CAR'07

Control Architectures of Robots 2007

2nd National Workshop on

Control Architectures of Robots:

from models to execution on distributed control architectures

- Paris - France
- May 31 and June 1st , 2007

Organized by LIP6, LIRMM and DGA

Coordinators: J. Malenfant, D. Andreu, A. Godin

Organization

The 2-day event is organized on May 31 and June 1st, 2007 in **Paris** on the **Jussieu campus**. It consists of 17 invited talks, and 1 round table.

Steering Committee

- Jacques Malenfant, LIP6, Univ. Pierre et Marie Curie, France - Jacques.Malenfant@lip6.fr
- David Andreu, LIRMM, Univ. of Montpellier II, France - andreu@lirmm.fr
- Aurélien Godin, ETAS, DGA, France – aurelien.godin@dga.defense.gouv.fr

Local Organisation

- Véronique Varenne, LIP6, Univ. Pierre et Marie Curie, France - Veronique.Varenne@lip6.fr
- Olena Rogovchenko, MoVe/LIP6, Univ. Pierre et Marie Curie, France
- Xavier Renault, MoVe/LIP6, Univ. Pierre et Marie Curie, France
- Lom Hillah, MoVe/LIP6, Univ. Pierre et Marie Curie, France

Web

- Jacques Malenfant, LIP6, Univ. Pierre et Marie Curie, France - Jacques.Malenfant@lip6.fr
- Stéphanie Belin, LIRMM, France - belin@lirmm.fr

Theme

This second national workshop is aimed at addressing important aspects of robot control architectures, with a specific emphasis on distribution, verification and validation, languages and modeling, and implementation of control architectures. It brings together researchers and practitioners from universities, institutions and industries, working in this field. It intends to be a meeting to expose and discuss gathered expertise, identified trends and issues, as well as new scientific results and applications around software control architectures related topics, through plenary invited papers.

Theme

Due to their increasing complexity, nowadays intervention robots, that to say those dedicated for instance to exploration, security or defence applications, definitely raise huge scientific and commercial issues. Whatever the considered environment, terrestrial, aerial, marine or even spatial, this complexity mainly derives from the integration of multiple functionalities: advanced perception, planification, navigation, autonomous behaviours, in parallel with teleoperation or robots coordination enable to tackle more and more difficult missions.

But robots can only be equipped with such functions if an appropriate hardware and software structure is embedded: the software architectures will hence be the main concern of this workshop.

As quoted above, the control architecture is thus a necessary element for the integration of a multitude of works; it also permits to cope with technological advances that continually offer new devices for communication, localisation, computing, etc. As a matter of fact, it should be modular, reusable, scalable and even readable (ability to analyse and understand it). Besides, such properties ease the sharing of competencies among the robotics community, but also with computer scientists and automatics specialists as the domain is inherently a multidisciplinary one.

Numerous solutions have been proposed, based on the "classical" three layers architecture or on more "modern" approaches such as object or component oriented programming. Actually, almost every robot integrates its own architecture; the workshop will thus be a real opportunity to share reflections on these solutions but also on related needs, especially standardisation ones, which are of particular importance in military applications for instance.

Hence, this second national workshop on control architectures of robots aims at gathering a large number of robotics actors (researchers, manufacturers as well as state institutions) in order to highlight the multiple issues, key difficulties and potential sources of advances.

Contents

Program

May 31, 2007

9:30-10:00	**Reception (coffee)**	
10:00- 10:30	**Welcome LIP6-DGA**	
10:30-12:00	**Session « Invited Speakers »**	
10:30-11:00	**Activities in the « GDR Robotique »**	F. Ingrand (LAAS)
11:00-12:00	**The EUROP Network**	P. Curlier (SAGEM)
12:00-13:30	**Lunch**	
13:30-15:30	**Session « Distributed Control Architectures»**	
13:30-14:00	**Mission Management System for Package of Unmanned Combat Aerial Vehicules** J.L. Farges (ONERA)	
14:00-14:30	**A generic architectural framework for the closed-loop control of a system** G. Verfaillie (ONERA), M.C. Charmeau (CNES)	
14:30-15:00	**Really Hard Time Developing Hard Real-Time: Research Activities on Component-based Distributed RT/E Systems** E. Borde (1, 2), G. Haik (1), L. Pautet (2) (1) THALES (2) Ecole Nationale Supérieure de Télécommunications	
15:00-15:30	**The RTMaps platform applied to distributed software development** N. du Lac (INTEMPORA S.A.)	
15:30-16:00	**Coffee Break**	
16:00-17:30	**Session « Validation and verification of control architectures »**	
16:00-16:30	**Formal assessment techniques for embedded safety critical system** C. Séguin (CERT)	
16:30-17:00	**Z and ProCoSa based specification of a distributed FDIR in a satellite formation** J.C. Chaudemar (SUPAERO), C. Tessier (Onera-Cert)	
17:00-17:30	**Incremental Construction and Verification of Robotic System using a Component-Based approach** A. Basu (2), M. Gallien (1), C. Lesire (1), T.H. Nguyen (2), S. Bensalem (2), F. Ingrand (1), J. Sifakis (2) (1) LAAS/CNRS (2) VERIMAG	
17:30-18:00	**Session « Implementation of control architectures »**	
17:30-18:00	**HS-Scale: a MP-SOC Architecture for Embbeded Systems** G. Sassatelli, L. Torres (LIRMM)	

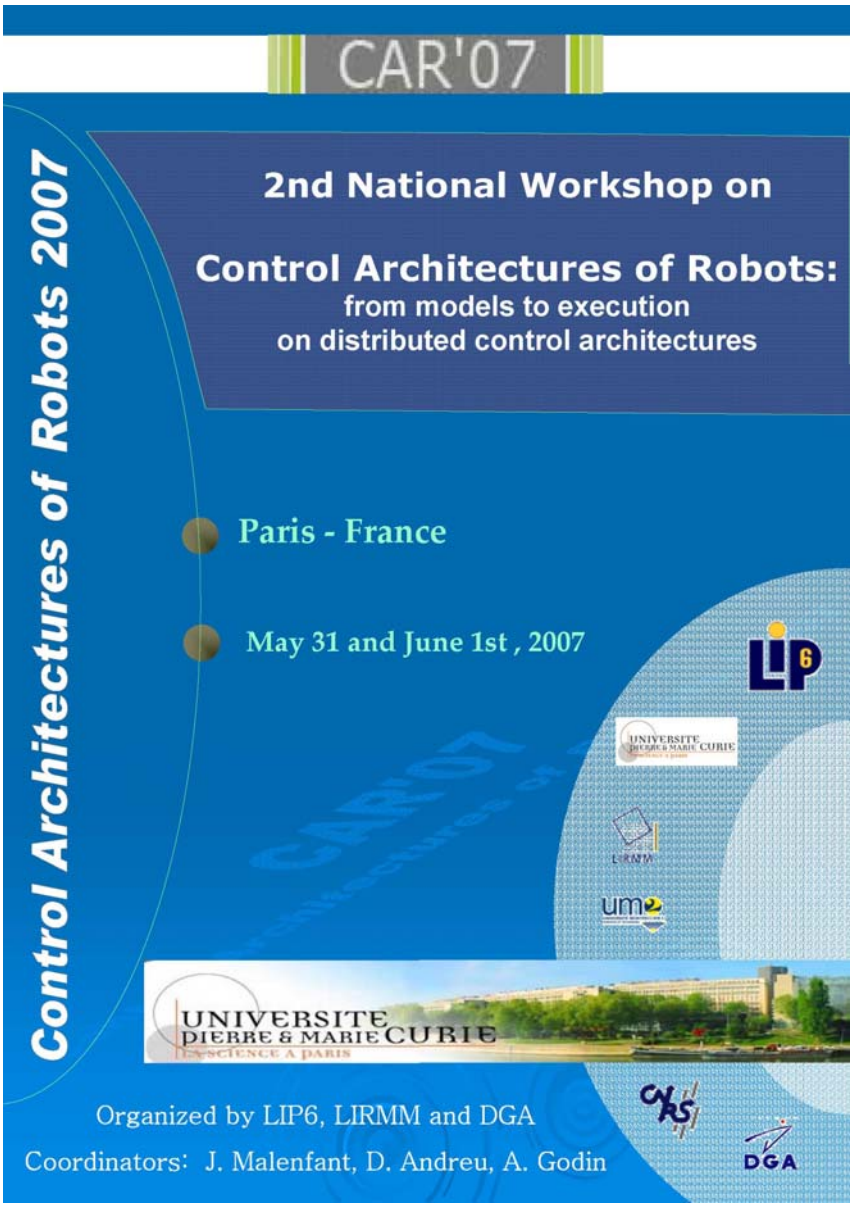
20:30	**Conference dinner (Altitude 95, Eiffel Tower)**

June 1st, 2007

9:30-12:00		**Session « Languages and MDA»**
	9:30-10:00	**A modelling language for communicating architectural solutions in the domain of robot control** R. Passama (OBASCO, École des mines de Nantes)
	10:00-10:30	**Benefits of the MDE approach for the development of embedded and robotic systems** X. Blanc (MoVe, LIP6), J. Delatour (ESEO)
10:30-11:00		**Coffee Break**
	11:00-11:30	**On Adaptive Robot Control Architectures** S. Stinckwich (GREYC - Université de Caen), N. Bouraqadi (École des Mines de Douai)
	11:30-12:00	**Design Principles for a Universal Robotic Software Platform and Application to URBI** J.C. Baillie (GOSTAI)
12:00-12:30		**Session « Validation and verification of control architectures » (late)**
	12:00-12:30	**A formal approach to designing autonomous systems: from Intelligent Transport Systems to Autonomous Robots** L. Petrucci (LIPN, Université de Paris XIII)
12:30-14:00		**Lunch**
14:00-16:00		**Round table**

Session

« Distributed Control Architectures »



MISSION MANAGEMENT SYSTEM FOR PACKAGE OF UNMANNED COMBAT AERIAL VEHICLES

J. Baltié*, E. Bensana, P. Fabiani**, J.L. Farges**, S. Millet***, P. Morignot*, B. Patin***, G. Petitjean*, G. Pitois* and J.C Poncet***

** Axlog Ingéniérie, ** ONERA, ***Dassault Aviation*

Abstract: This paper presents the development and the assessment of a mission management system (MMS) for a package (group of aircraft assuming together a common mission) of Unmanned Combat Aerial Vehicles (UCAV). The mission is carried out in an environment including a safe area and a dangerous area, no-flying-zones, a command and control centre, the terrain, threats and targets. The MMS architecture presents reactive and deliberative layers. It includes a planner that distributes the plan computation effort among the UCAV. This plan takes into account the co-ordination and collaboration necessary for the package and its associated assets to implement the mission. Simulation results indicate that the MMS is able to carry on missions, to activate contingent behaviours, to decide whether or not to plan and to re-plan.

Keywords: Aircraft operations, architectures, autonomous vehicles, constraint satisfaction problems, decomposable searching problems, management systems, distributed artificial intelligence, planning, robotics, simulation.

INTRODUCTION

Some early tests with uninhabited aerial vehicles carrying and delivering weapons were conducted in the sixties and the seventies. However, Unmanned Combat Aerial Vehicles (UCAV) have been widely studied only since 1980. UCAV can be used for locating, identifying and destroying enemy targets. Multiple operators remotely operate currently one UCAV. UCAV demonstrators such as X-45 (Wise, 2003), X-47A and NEURON are developed. Nowadays, such studies as the NEURON demonstrator project envision leveraging effects by mixing these UCAV platforms with inhabited ones like RAFALE. For example, the sketch on Figure 1 shows a RAFALE collaborating with a package of two NEURON.

Such a vision lead researches towards an increase of UCAV and package of UCAV autonomy. For instance, Mehra *et al*. (2000) design a control scheme that supports autonomous co-ordinated flight of multiple UCAV. Other examples are the proposition by Li *et al*. (2002) of a hierarchical control scheme for a package of UCAV that provides path planning, trajectory generation and formation keeping and the development by Beard *et al*. (2002) of a target assignment method for a package of UCAV.

Fig. 1. A RAFALE collaborating with a package of two NEURON

UCAV decisional autonomy implies not only path planning and target assignment functions but also functions for planning and executing target and weapon selection, system reconfiguration, synchronisation actions, etc. Moreover, all those functions have to be integrated inside an UCAV sub-system dedicated to the management of the mission.

Barrouil *et al.* (1999) defined the scope and goals of what could a Mission Management System (MMS) do inside a mixed air patrol. Grounded on this work and European studies such as MISURE (Avalle and Patin, 2007), this paper presents the development and the assessment of a MMS giving decisional autonomy to a package of UCAV. This MMS includes a planner that allows on-line distributed computation of a new plan when a disruptive event occurs.

The first section of this paper is devoted to the presentation of the requirements for the development of the architecture. It includes the description of the environment of the mission and the presentation of the UCAV equipment the software has to interact with. Then the problems linked with architectural choices are addressed in the second section and the global architecture of the MMS is provided. This architecture includes reactive and deliberative layers that are described in the third and fourth sections respectively. The deliberative layer aims at solving, a planning problem including target selection, target assignment, weapon selection, path planning, collaboration and co-ordination aspects for a package of UCAV. Moreover, the problem is solved through its distribution over the UCAV. The issue of control of MMS computation time is addressed in the fifth section. The sixth section gives some experimental results about the plan computation and the MMS behaviour. Finally some conclusions are presented.

1. ENVIRONMENT AND REQUIREMENTS

1.1 The system

The environment of the package is presented on Figure 2. It includes a friend (or safe) area where the normal traffic management rules apply and a foe (or dangerous) area where the military authorities provides the air orders, No Flying Zones (NFZ), a Command and Control (C2) centre that supervises the package and provides eventually new information or new orders. It also includes the terrain, known and unknown threats, primary targets to be processed by the mission and secondary targets that are processed on an opportunity basis. This environment is dynamical and the perception of the environment by the package is also dynamical: threats may be discovered during the course of the mission. Tactical assets like targets may also be redefined by the C2.

Each UCAV of the package get a specific payload that may include several types of equipment:

- Weapons include bombs, missiles (WEAP) and laser designators (LD). Some type of weapons need to be guided using LD assets either on the same or another vehicle. Some other need the LD only to achieve a precision level.
- Localisation devices include satellite (GPS), radio (RS) and inertial (IN) positioning systems. The localisation information should be merged to produce continuously an accurate localisation of the aircraft.
- Flight management devices include engine control (ENG), autopilot (AP), fuel level sensors (FUEL) and flight control systems (FCS). They are developed and should be organised in order to provide relevant level of safety for such aircraft.
- Communication devices implementing intra-formation (IF), low bandwidth (LBW) and high bandwidth (HBW) data-links. The IF data-link is grounded on the medium frequency band and provides highest discretion but presents a limited range. The LBW data-link is grounded on lowest frequency band and presents the highest range but provides poor discretion. Finally the HBW data-link is grounded on highest frequency band and provides medium discretion but is dedicated only to upload of image necessary to designate a target by C2. The use of these links are constrained by mission dependant characteristics such as the will not to be seen or the safety of the package.
- Auto-protection devices include missile approach warners (MAW), radar warning receivers (RWR), chaffs (CHAFF), flares (FLARE), jammers (JAMM.) and active electronic counter measures (ECM). There is also the capability to control the Radar Cross Section (RCS) of the aircraft.
- Sensors include synthetic aperture radars (SAR) and electro-optical sensors (EO).

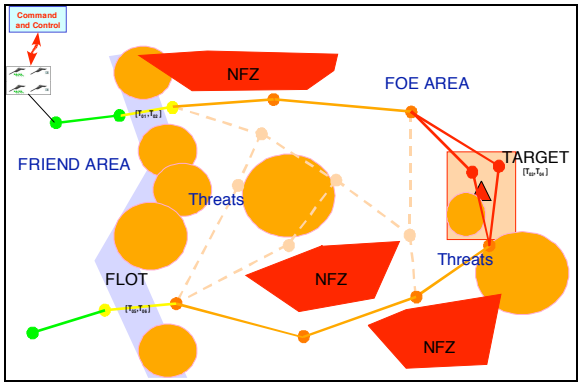


Fig. 2. Environment of the package of UCAV

Moreover, each UCAV is subject to flight dynamic constraints. Those constraints include:

- Maximum altitude,
- Speed limits,
- Load factor limits and
- Fuel consumption.

Finally, constraints appear due to the package work. Those constraints include:

- distance to respect in order to use discrete communications,
- relative positions of aircraft to gain benefit of the use of one auto-protection device,

- synchronisation issues and relative angles when laser designation is delivered by one aircraft and used by another aircraft to deliver weapons.

1.2 The problem

A mission plan is defined before take-off and the aims of the MMS are to follow the mission plan, to ensure safety, to ensure survivability and to ensure the success of the mission. Some requirements are deduced from those aims:

- the plan must be applied,
- disruptive events must be detected and analysed,
- if needed, reactive actions must be carried out and,
- if needed, the mission plan must be recomputed on-line.

Moreover, the flight dynamic constraints of the UCAV must be respected.

The package mission management problem includes more than the replication by the number of UCAV of a mission management problem for each agent. The set of actions that can be performed by a package of UCAV is larger than the one for a single UCAV. For instance the package can split; some UCAV fly to a convenient place to perform detection, identification and localisation of targets and other UCAV fly to another place to perform the strike itself. After the action the package can merge.

2. ARCHITECTURAL CHOICES

2.1 Approach

As pointed out by Findeisen and Lefkowitz (1969), there is two main ways of approaching control hierarchies:

- Control hierarchies with several levels: The system is decomposed in sub-systems. A controller is developed for each sub-system considering local criterion and local model. Higher level controllers integrate actions of lower level controllers in order to fulfil system objectives.
- Control hierarchies with several layers: The control problem is decomposed in sub-problems, for instance: reaction, optimisation, adaptation, auto organisation. Each sub problem is treated by a suited technique and the integration is made by the higher layers in order to solve the global problem.

Figure 3 present the application of the two approaches to the mission of a package UCAV. The decomposition of the system considered here could consist in splitting the mission in package and environment. The package is decomposed in different UCAV and the UCAV in different types of equipment. The decomposition of the problem in sub-problems highlights execution, disruptive event detection, disruptive event analysis, reaction and planning.

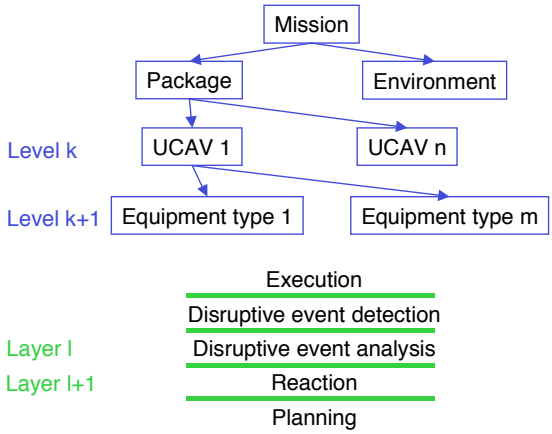


Fig. 3. Possible approaches to control hierarchies for the mission of a package of UCAV

The approach considered is grounded on both decompositions and also takes into account the fact that each UCAV has its own on-board computer and that there are communication constraints between UCAV. This approach stems from preceding studies (Riedel *et al.* , 2006; Avalle and Patin, 2007) where its efficiency was shown.

2.2 Levels of the hierarchy

As shown on Figure 4, the control hierarchy is organised in three levels that can be found on each UCAV. Each level is able to work with the corresponding level in the other UCAV. At the higher level, the MMS component is distributed among the UCAV and interacts with the other architecture components. The group of UCAV elaborates the different coupled plans using this distributed component. At an intermediate level, a component (TACSIT) is devoted to elaboration of a tactical situation. It is also distributed among the UCAV and merges different source of information to create the shared tactical situation. MMS and TACSIT have only information processing capability.

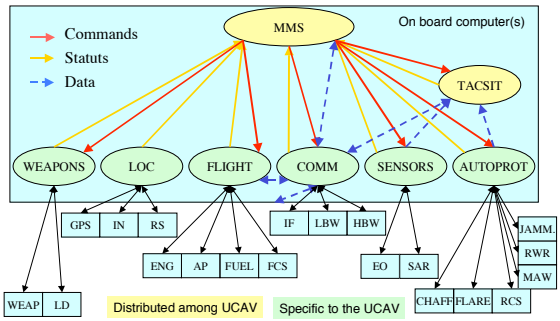


Fig. 4. Control hierarchy with three levels for the mission of a package of UCAV

At the lower level architecture components are able to control a specific part of the hardware of a specific UCAV. The functions of lower level components are localisation (LOC), flight management (FLIGHT),

communication (COMM), self-defence (AUTOPROT), sensor management (SENSORS) and weapon management (WEAPONS). The details of the equipment controlled by each component are given in section 1.1. Equipment are only controlled by these components. There is no direct control by the MMS on the equipment. The rationale behind this decomposition is linked to the reality of the industrial organisation where aeronautic companies delegate components realisations to other companies. It also must be stated that the functional description of the lower level of the architecture is very similar to real military aircraft functional description.

From a practical point of view components are connected inside each UCAV through a software bus. Flows of information required to be able to implement a control of the package are numerous and of different types:

- Commands from the MMS to the other aircraft components,
- Status from the other aircraft components to the MMS,
- Data exchanged between components and,
- One very important asset, data between one UCAV communication component to another UCAV one.

2.3 Layers of the MMS

Existing architectures for MMS of autonomous vehicles are reactive or deliberative or both. Reactive architectures are not able to support problem solving but react quickly to events while deliberative architectures are fully based on problem solving and usually react slowly. The use of at least two layers to allow both behaviours is common in the literature since the work of Bresina and Drummond (1990). Practical intelligence of an UCAV consists in a mix of reactive and deliberative behaviours. The architecture is designed according to three principles:

1. The on-line planning shall be activated only when the current plan is invalidated by the current situation.
2. The reactive level shall not only execute the plan but also handle emergency situations.
3. Sensor inaccuracy is managed through pre-defined behavioural procedures for inaccuracy reduction.

Figure 5 presents the actual MMS architecture with reactive and deliberative layers. This architecture is organised using an underlying database that stores information about:

- the vehicle and the other vehicles,
- the known threats in the environment,
- the targets of the mission,
- the plan to carry out the mission and
- the configuration parameters and thresholds.

When considering the MMS database, it must not be confused with the TACSIT component. In fact, there is no semantics linked to the tactical situation described. It is influence map, list of objects, list of path (foe and friends) and the like. The MMS database is filled by its different parts interpreting data issued of the TACSIT component and giving sense to these data with respect to the concept manipulated by the MMS and its planning part.

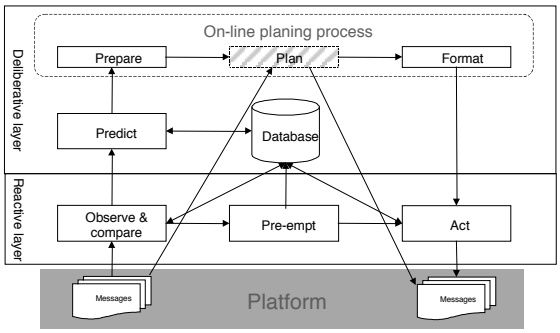


Fig. 5. Mission management system architecture with two layers

In this architecture, the purpose of the reactive layer is to execute the plan and to implement contingent behaviours for disruptive events that need quick reaction. The purpose of the deliberative layer is to determine if a new plan is needed and, in that case, to compute it.

3. REACTIVE LAYER

The reactive layer includes the following functions: “observe and compare”, “pre-empt” and “act”.

3.1 Observe and compare

The “observe and compare” function receives the messages from the platform, from other platforms and from C2 through the communication component. It updates the database information according to those messages and performs tests about the short-term situation. It tests the possible disruptive events such as unexpected exposure to a threat, approaching missile, presence in a NFZ, loss of communication and failure of a vehicle component. Most of the time, this function does not send messages in direction to “pre-empt” and “predict”. For the failure of a vehicle component a message is sent to “predict”, for other disruptive events a message is sent to “pre-empt”.

3.2 Pre-empt

The “pre-empt” function determines if necessary contingent behaviours and their unit actions. This function verifies the UCAV situation and uses a set of rules to determine the suited contingent behavior. If necessary, it computes the parameters of a contingent behaviour and activates it. Finally, the function computes the actions associated with the active contingent behaviours. Four behaviours may be activated by the function:

- The behaviour for new radar threat detection implements update of the radar list and radar locations, active electronic counter measure under specified conditions, manoeuvre for

avoidance or information gathering under other specified conditions.
- The behaviour for approaching missile detection implements unconditional use of active electronic counter measures, chaff and flare decoy and evasive manoeuvres.
- The behaviour for loss of communication between neighbouring vehicles consists in commanding the altitude of the vehicles to different pre-determined flight levels in order to ensure the absence of collision between them. The secured altitude slots are attributed to each aircraft at plan generation time, ensuring their uniqueness for each aircraft of a formation.
- The behaviour for NFZ violation avoidance is implemented in two parts. First, a modification of current trajectory is computed, attempting to avoid incoming NFZ, going round it by the shortest way. If this fails or if the situation evaluation reports that aircraft suddenly appears to be inside a NFZ, solution consists in a fast trajectory computing that will attempt to exit NFZ by crossing the closest frontier point.

All behaviours have a date parameter, a timeout and a homing way-point parameter. They are updated each time the "pre-empt" component is activated. This update can be null according to the current aircraft situation, meaning that the system can return to nominal plan execution. Otherwise, the contingency ends once its associated timeout has been passed out or when homing point is reached. All behaviours, except loss of communication, have a Boolean parameter indicating if the global path can be modified. Finally, the behaviour for new radar threat detection has an additional parameter indicating the origin way-point.

Four types of information represent the behaviours:
- The warning processing information indicates what should be done in terms of knowledge management.
- The system management information indicates what should be done in terms of auto-protection actions.
- The flight plan modification information indicates what should be done in terms of navigation actions.
- The end of contingency information indicates conditions for terminating the behaviour.

3.3 Act

The "act" function carries out the unit actions of the "format" or of the "pre-empt" function. It also determines if an action is correctly performed or not and if it is performed in time. For this, the function analyses the status of the sub-systems stored in the database. When an action is assumed finished, it triggers the next actions. In order to allow the flight management component to perform trajectories with appropriate turns, navigation actions are not treated individually, but by couples of successive actions.

Whenever actions of the UCAV are linked to actions of other UCAV they are to be done on conditions. This is the way to implement collaborative actions such as fire to shooter sequences.

Actions are scheduled according to their execution criteria:
- At a given instant,
- At a given location or,
- At an instant defined by the execution of another action:
  - Simultaneously,
  - Immediately after or,
  - After a given interval of time.

When actions of the plan and from "pre-empt" are conflicting, it always gives the priority to pre-emption actions. This mechanism corresponds to a suppressor function in the sense of Brooks (1986). Finally the function sends messages to the platform sub-systems.

4. DELIBERATIVE LAYER

The deliberative layer includes the following functions: "predict", "prepare", "plan" and "format".

4.1 Predict

The "predict" function assesses the feasibility of the on going plan and decides to compute a new plan or not. Most of the time, this function does not send message in direction to "prepare". The feasibility of the on going plan is assessed with respect to several criteria:
- Probabilities of UCAV survival and of target killing are updated and checked against a threshold.
- The possibilities of fulfilling time constraints at some waypoints and of having enough fuel to finish the mission are checked.

It should be noted that the "predict" component faces a dilemma. On one hand, deciding to re plan all the time leads the agent to an erratic behaviour, always starting the beginning of new unrelated plans, and therefore not leading to any goal at all (too often replanning). On the other hand, deciding to plan too scarcely leads the agent to follow unusable plans, since the behaviour of the agent does not adapt to what actually happens in the environment (too scarce replanning). The solution we propose for this "predict" component is a medium term on the previous spectrum, by using variables representing states of the agent. When the mean of these variables is above some threshold, then replanning decision is taken and replanning occurs. This solution is not satisfactory in principle, since it does not solves the problem of the continuity of behaviour of the agent over successive replanning activities. But at least it provides a practical and simple (but not elegant) solution, even if these variables and thresholds need careful tuning for realistic replanning frequency to be

adopted. Moreover, the occurrence of a replanning request while replanning has to be managed. This management is performed using priorities on replanning reasons. If the priority of the reason of the present replanning request is lower than the one of the on going replanning, the request is ignored. Otherwise, the on going replanning activity is stopped and the replanning is started with a context including the present request.

4.2 Prepare

The "prepare" function gathers and generates data for the "plan" function. The data describes a planning context for a package and includes:

- The time at which the plan should began because problems are not stationary.
- The vehicles participating to be considered and their predicted state in terms of geometry and resources at the time at which the plan should began. Three classes of vehicles are to be distinguished:
  - The vehicles participating to the communication cluster in which the planning is carried on.
  - The vehicles not participating to the communication cluster but presenting a plan assumption.
  - The vehicles not participating to the communication cluster and assumed out of order.
- Relevant characteristics of the environment including threats, targets and NFZ. It includes the description of the airspace threatened by each threat.
- A graph, with nodes and edges, including possible paths for acquisition, attack and return to base. This graph is built in two steps. An initial graph is deduced from the initial mission plan by associating mission waypoints to graph nodes and transitions between these waypoints to graph edges. Nodes and edges are tagged according to their strategic properties for acquisition, shooting, etc. The second step is done each time the component is activated. It consists in the generation of different alternative paths for each strategic action, including Return To Base. These paths are generated using a potential field algorithm in which threats and NFZ are associated to repulsing potential while targets and base airport are associated with attractive potential.
- Time intervals and altitude constraints at waypoints.
- The goal-action prototypes in terms of resources to be used by the vehicles at specified places in the space and at specified times in order to achieve each goal.

4.3 Plan

The "plan" function allows a multi-UCAV distributed planning. It can exchange directly messages with other UCAV. It takes a planning context given by "prepare" and provides a plan to "format".

The resulting plan includes for each vehicle participating to the communication cluster a timed sequence of actions. There are two kinds of actions:

- Navigation actions and
- Use of resources by vehicles in order to fill a goal-action prototype.

Constraint programming approach: The planning problem for the package involves different aspects: selection of goals, selection of an action mode for each goal, assignment of UCAV and their resources to each selected action mode, path planning and scheduling for each UCAV. Constraint programming is a powerful approach for integrating those different aspects. Indeed this approach is efficient for path planning (Allo *et al.*, 2002; Strady-Lécubin and Poncet, 2003) as well as for planning problems expressed by means of a propositional representation (van Beek and Chen, 1999). Thus, the problem is modelled using a constraint programming approach. Variables are associated to UCAV and nodes. For instance the Boolean variables $P_{i,j}$ and $I_{i,j}$ indicate respectively that UCAV i will pass by node j and is involved in a target attack at that node, $Q_{k,i,j}$ indicates that it uses resource k. The integer variable $T_{i,j}$ gives the arrival time of the UCAV at the node. Other variables are associated to UCAV and edges and to goals. For instance, for a target o, the Boolean variables A_o, $A_{o,j}$ and $A_{o,j,m}$ indicates respectively that the target will be attacked, that the attack takes place at node j and that it is done in mode m. The integer variables $Tobj_o$ and Eff_o indicate the attack time and efficiency respectively. Constraints describe navigation possibilities and conditions for goal achievement. In the model, the link between the path planning and the propositional planning parts of the problem is ensured by constraints of the type:

$$I_{i,j} \le P_{i,j} \quad (1)$$

$$I_{i,j} \le \sum_k Q_{k,i,j} \quad (2)$$

$$Q_{k,i,j} \le K_{k,i} I_{i,j} \quad (3)$$

$$\sum_i Q_{k,i,j} \ge R_{k,m} A_{o,j,m} \quad (4)$$

$$\sum_m A_{o,j,m} = A_{o,j} \quad (5)$$

$$\sum_j A_{o,j} = A_o \quad (6)$$

Equations 1 and 2 indicate that pre-conditions for an UCAV to participate to an attack at a node are to pass by that node and to have some resource to use at that node. Equation 3 bound the resources usage by zero if the UCAV does not participate and by the available quantity, $K_{k,i}$, otherwise. Equation 4 indicates that the pre-condition for the package to attack a target in a given mode is to have at least the resource amount requested for that mode, $R_{k,m}$. Equations 5 and 6 indicates that a single mode and a single node are selected for the attack of the target.

Distribution of the computation: The graph of variables and constraints associated to a multi-vehicle mission presents a star structure: the variables associated to goal achievement are connected by constraints to the variables associated to the different UCAV but there is no direct constraint between the variables of two UCAV. This structure allows the decomposition of the initial problem into a problem associated to each UCAV and a goal achievement problem. The decomposition of the initial problem has the advantage of permitting the use of the computing resources of all UCAV. Several techniques are available to conduct the distributed search of a solution. The technique used works in three steps:

1. Sets of solutions are searched for the problems associated to each UCAV.
2. A co-ordination problem, including the goal achievement problem and the selection of one solution per set, is solved.
3. The solution is refined for each UCAV.

Implementation: This technique is implemented using JADE (Bellifemine *et al.*, 1999), a FIPA compliant agent framework, and the CHOCO (Laburthe, 2000) tree searching constraint solver. Special attention has been given to the problem of the control of the time spent by the solver to solve the sub-problems; selection of variables to be assigned, selection of values for those variables, interruption of a tree search and time assignment to each step of the resolution.

4.4 Format

The "format" function refines the macro actions of the plan into sequences of unit actions. For instance the macro action "launch bomb 1 on target 101 at time t" is refined in the sequence of unit actions:

- "select resource type bomb 1 at time t-d" then
- "initialise selected resource with target 101 features at time t-e" then
- "ask to C2 go/no go at time t-f" then
- "if C2 answer is go fire bomb 1 at time t-g".

Finally, "format" send a message to "act" to inform it about the existence of a new plan.

5. EXECUTION AND COMPUTATION TIME

5.1 Activation logic

The activation logic of the MMS modules is presented on figure 6. The MMS is activated every 0.1 seconds. For most of the cycles only three functions are activated: "observe and compare", "predict" and "act". For those cycles, the result of the analysis of messages from the platform by "observe and compare" and "predict" indicates that no pre-emptive behaviour has to be activated and no new plan has to be computed. The "act" function continues carrying on actions of the current plan.

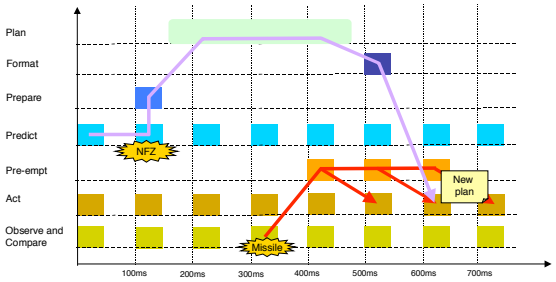


Fig. 6. Activation logic of the modules of the MMS

For cycles where "observe and compare" indicates that a pre-emptive behaviour has to be activated, the "pre-empt" function is additionally activated. This function may remain activated for several consecutive cycles until the behaviour is finished. Meanwhile the "act" function applies the actions issued from the behaviour. For instance, if the disruptive event is the detection of a missile firing, the function remains activated until the end of the escape manoeuvre.

For cycles where "predict" indicates that a new-plan has to be computed, the "prepare" function is additionally activated. In the same cycle the "plan" function is activated in the background. Several cycles after the "plan" function provides a plan and activates the "format" function just for a single cycle.

5.2 Taking into account constraints

Special attention is given to the way the computation time constraints are taken into account. First, different priorities are assigned to the input messages and only a bounded number of messages are processed at each cycle. Second, long processing is performed over several cycles, for "predict", or in a separate thread, for "plan". Moreover, "plan" solves the co-ordination problem by searching first a solution without attack. This solution is obtained very quickly providing an almost anytime feature for "plan". The conjunction of those techniques ensures that a computation time bound for a time cycle of the mission management system can be predetermined.

6. EXPERIMENTAL RESULTS

Experimental results are given in the context of suppression of enemy air defence and strike scenarios.

6.1 Unit test of the planning function

In order to assess the performance of the planning function, tests are conducted on a single Sun Blade 1500 computer. A planning request for four UCAV with 4 targets conducts to the generation of 46 solutions for the first UCAV after 1.0 seconds, 46 solutions for the second UCAV after 1.6 seconds, 46 solutions for the third UCAV after 1.7 seconds and 138 solutions for the fourth UCAV in 2.4 seconds.

Figure 7 illustrates the second step of the planning method. A solution with no target attacked is found in about 1.0 seconds. Then as resolution time increases the efficiency of the solution, in terms of average number of targets destroyed, is improved and more targets are attacked. Finally the refinement of the solution is given in 0.05 seconds for the first UCAV, 0.09 seconds for the second UCAV, 0.14 seconds for the third UCAV and 0.24 seconds for the fourth UCAV. Note that the resolution times for the first and third step of the method are over estimated because the tests are conducted using a single computer. Finally, it can be observed that if optimality is not required the computation time for the co-ordination step can be reduced to few seconds.

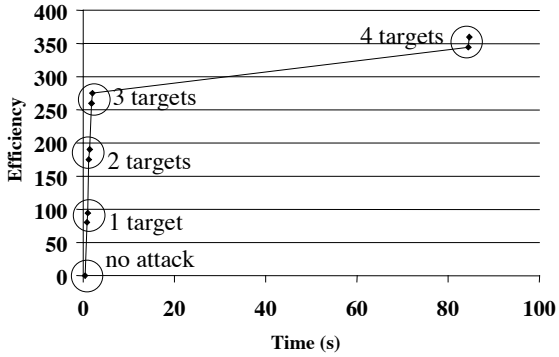


Fig. 7. Efficiency (%) of the solution of the co-ordination problem in function of the computation time

6.2 Simulation experiments

A simulation program was developed in order to test the behaviour of the MMS for a complete mission execution. In order to run these assessments there was an incremental complication of the scenarios starting from only one aircraft doing a flight path without any action and ending with a complete eight UCAV scenario including all the sequence of events possible.

Figure 8 presents the result of an experiment conducted with this program. It shows how the package, here two UCAV, reacts to the introduction of a new threat and a simultaneous failure in its jamming capabilities.

The first step is the detection by different UCAV of the package of the presence of the new threat. After having localised this threat the package decides to replan because of a too high probability of being killed if it follows the original path. The second step is then to prepare new segments on which the package will be able to find a new path and eventually assign actions. As there is no more jammers usable and as the fuel consumption used to avoid the new threat is compatible with the goal of the mission without endangering more the package, the solution kept is to go around this new threat changing flight level in order to use the terrain as a natural mask. This adaptation of the flight path to avoid the threat leads to the adaptation of how to attack the target because of the cooperation needs of the two UCAV. The new plan is then applied at a previously defined waypoint of application.

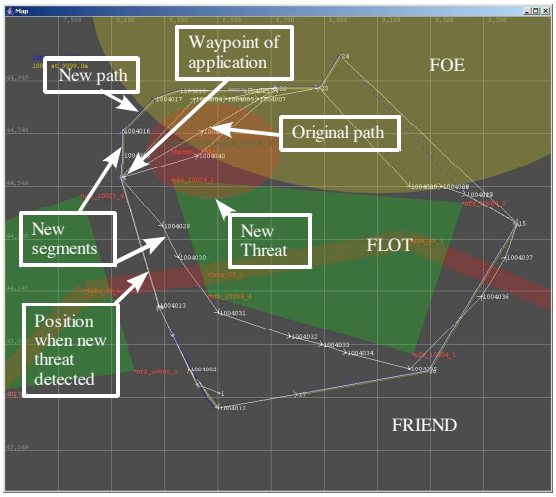


Fig. 8. Introduction of a new threat and package reaction

More experiments were done during the project in order to increase the complexity of the planning to be done and their results demonstrate the capability of the package integrating a distributed planing function through a reactive and deliberative architecture to achieve the goals specified. That is, the package is able to carry on nominal missions as specified, to activate the contingent behaviours on disruptive events, to decide whether or not to plan and, if necessary, to plan and to run in a bounded time. Moreover data link requirements for the functions of the MMS and performance of distributed planning are assessed.

Figure 9 shows how complex a tactical situation can become and what the planner had finally to manage as well in the nominal situation than in the presence of events.

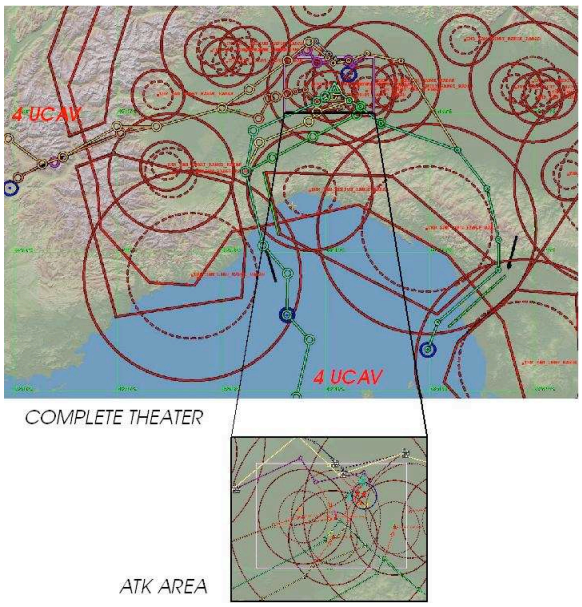


Fig. 9. Complexity of a tactical situation

CONCLUSION

The proposition of this architecture and a distributed planning method for package missions contributes to demonstrate the feasibility of vehicle intelligence and autonomy. Indeed, with the integration of on-line planning, disruptive events in absence of human intervention do not conduct necessary to the abortion of the mission.

However this area of application of robotic architectures is not fully covered. Some research directions are:

- Study of the link between the geometry of the flight path and the actions.
- Study of the way of taking into account uncertainty about the state of the vehicles and the environment as distance from current date increases.
- Study of the efficiency of other distributed methods.
- Study of mixed initiative planning for fleets with manned and unmanned vehicles.
- Study of the sustainability of the mission consistency despite the ability of computing several new plans while performing the mission.
- Study of the possibility of deriving proofs about the frequencies of pre-emptive behaves and re-planning activity.
- Study of the possibility of deriving proofs about the architecture safety and about the planner safety as a preliminary step to certification.

REFERENCES

Allo, B., C. Guettier, V. Legendre, J.C. Poncet and N. Strady-Lecubin (2002) Constraint model-based planning and scheduling with multiple resources and complex collaboration schema. In: *Proceedings AIPS'02*.

Avalle, M. and B. Patin (2007) Mission Management System for UCAV. Presented in: RTO meeting, AVT146, Platform Innovations and System Integration for unmanned Air, Land and Sea vehicles.

Barrouil, C., R. Demolombe, P. Fabiani, C. Tessier, P. Da Silva Passos, Y. Davaine, B. Patin and N. Prego (1999) TANDEM: an agent-oriented apoach for mixed system management in air operations. In: *RTO meeting proceedings 46, Advanced Mission Management and System Integration Technologies for improved Tactical Operations.*

Beard, R.W., T.W. McLain, M.A. Goodrich and E.P. Anderson (2002) Coordinated target assignment and intercept for unmanned air vehicles. *IEEE Transactions on Robotics and Automation*, **18(6),** 911-922.

van Beek, P. and X. Chen (1999) CPlan: A Constraint Programming Approach to Planning. Proceedings of AAAI, 585-590.

Bellifemine, F., A. Poggi and G. Rimassa (1999) JADE – A FIPA-compliant agent framework. In: *Proceedings of PAAM.*

Bresina, J.L. and M. Drummond (1990) Integrating planning and reaction. A preliminary report. Proceedings of the American Association of Artificial Intelligence Spring Symposium.

Brooks, R. (1986) A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, **2(1),** 14-23.

Findeisen, W. and I. Lefkowitz (1969) Design and Applications of Multilayer Control, 4th Congess of the International Federation of Automatic Control, Technical session 42, 3-22.

Laburthe, F. (2000) CHOCO: implementing a CP kernel. In: *CP'00 Post Conference on Techniques for Implementing Constraint programming Systems*. TRICS. Singapour.

Li, S., J.D. Boskovic, S. Seereeram, R. Prasanth, J. Amin, R.K. Mehra, R.W. Beard and T.W. McLain (2002) Autonomous Hierarchical Control of Multiple Unmanned Combat Air Vehicules (UCAVs). In: *Proceedings of the American Control Conference*. Anchorage.

Mehra, R.K., J.D. Boskovic, and S. Li (2000) Autonomous Formation Flying of Multiple UCAVs under Communication Failure. In: *Position Location and Navigation Symposium IEEE 2000*, 371-378, San Diego.

Riedel, J.E., S. Bhaskaran, S. Desai, D. Han, B. Kennedy, G.W. Null, S.P. Synnott, T.C. Wang, R.A. Werner and E.B. Zamani (2006) Autonomous Optical Navigation of Deep Space I - http://nmp-techval-reports.jpl.nasa.gov/.

Strady-Lécubin,N. and J.C. Poncet (2003) Mission Management System High Level Architecture, Report 4.3. MISURE/TR/4-4.3/AX/01, EUCLID RTP 15.5.

Wise, K.A. (2003) X-45 Program Overview and Flight Test Status. In: *2nd AIAA "Unmanned Unlimited" Systems, Technologies and Operations – Aerospac*. San Diego.

ACKNOWLEDGEMENT

The French "Délégation générale de l'armement", a part of the ministry of defence, has funded this work from 2003 to 2006 in the scope of the ARTEMIS project: thanks to this institution. The authors acknowledge the ARTEMIS partners for their contribution: thanks to Jean-Francois Gabard and Catherine Tessier.

A generic architectural framework for the closed-loop control of a system

Gérard Verfaillie and Michel Lemaître
ONERA
2 av. Édouard Belin, BP 74025, F-31055 Toulouse Cédex 4
Gerard.Verfaillie@onera.fr, Michel.Lemaitre@onera.fr

Marie-Claire Charmeau
CNES
18 av. Édouard Belin, F-31401 Toulouse Cédex 9
Marie-Claire.Charmeau@cnes.fr

Abstract

In this article, we present a generic framework for the functional architecture of the closed-loop control of an engine or a system. Besides its genericity, its main features are (1) a decomposition of the system control into hierarchically organized modules, (2) an encapsulation of control and data inside each module, (3) standardized communications between modules via requests, reports about request execution, and information about the system state, (4) a standardized organization of each module around the following four components: tracking of the received requests, tracking of the emitted requests, tracking of the system state, and decision-making upon request emission, and (5) a common framework for the interaction between reactive and deliberative tasks inside the module components and especially inside the state tracking and decision-making ones.

We show how this framework can be applied to the control of an autonomous satellite dedicated to Earth watching and observation.

1 The AGATA project

The architectural framework presented in this paper is one of the first results of the AGATA project (Autonomy Generic Architecture: Tests and Applications, http://agata.cnes.fr). From mid-2004, this project brings engineers

and researchers from CNES[1], ONERA[2], and LAAS-CNRS[3] together around the global objective of increasing spacecraft autonomy [CB05].

Most of the satellites and space probes that are today in operation are permanently and tightly controlled by human operators in ground control centers. Except for specific tasks, such as thermal, energy, attitude, telemetry, or telecommand control, for which a reactive control loop is necessary onboard, they have no autonomous control capability. They cannot reconfigure themselves autonomously after a subsystem failure. They cannot decide and control autonomously orbital manoeuvres in case of a too large drift from their reference orbit. In case of satellites or probes dedicated to observation of Earth, of other planets, comets, or asteroids, or of the universe outside the solar system, they cannot decide autonomously on the observations they perform. In case of rovers at the surface of planets, they cannot decide on the areas they explore.

For all these tasks, they must wait for decisions made on the ground by human operators or at least under their supervision. The first difficulty is that communication may not be permanently possible between satellites and space probes, on the one hand, and their ground control centers, on the other hand. This is the case with Earth observation satellites for which visibility windows may be rare (about 10% of time) due to their low orbit altitude. This is also the case with planet exploration probes or rovers when they are hidden by the planet. The second difficulty is that communication and normal mission execution may be incompatible. This is the case with planet exploration probes for which communication with Earth, on the one hand, and planet observation, on the other hand, require incompatible spacecraft orientations. The third difficulty is that the communication time may be incompatible with the requirements in terms of onboard reactivity. This is the case with planet exploration probes or rovers. For example, communication takes some tens of minutes between Mars and Earth at the light speed.

The result is a loss in terms of reactivity. In case of subsystem failure, the whole system is unavailable until a communication be possible with the ground control center, human operators make decisions, and send them to the satellite or to the probe. In case of observation systems, observation

[1]CNES: Centre National d'Études Spatiales, French Space Agency, http://www.cnes.fr

[2]ONERA: Office National d'Études et de Recherches Aérospatiales, French Aerospace Lab, http://www.onera.fr

[3]LAAS-CNRS: Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique, Laboratory for Analysis and Architecture of Systems of the French Research Center, http://www.laas.fr

opportunities may be missed. This is the case with the satellites that are currently under consideration for the surveillance of phenomena at the Earth surface, such as forest fires, volcanic eruptions, pollutions, or floods. If they are not equipped with autonomous decision-making capabilities in terms of observation, they may miss important observation opportunities immediately after detection. This is also the case with planet exploration rovers. If they are not equipped with autonomous decision-making capabilities in terms of movement and observation, they may miss scientifically important observation opportunities. To take another example, most of the images taken today by optical Earth observation satellites (about 80%) are lost because of the presence of clouds. To put this right, cloud detection systems in front of these satellites are currently under consideration, but autonomous decision-making capabilities in terms of observation are necessary to take into account onboard the information provided by these systems.

In all cases, autonomy can improve system dependability and global mission return in terms of quantity, quality, and quick delivery of collected data.

Studies about satellite and space probe autonomy have been active since the nineties. One can cite the generic seminal work performed at NASA Ames and JPL in the context of the DS-1 technological space probe [MNPW98] and works performed at JPL in the context of the technological Earth surveillance and observation EO-1 satellite [CST+05, CCD+05], which demonstrated operationally the feasibility of onboard ground phenomena detection and autonomous observation planning and replanning. One can also cite works performed at MIT in the domain of autonomous failure diagnosis and reconfiguration [WICE03]. In Europe, one can cite works in the domain of autonomous Earth surveillance and observation [DVC05] and in the domain of autonomous orbital manoeuvres [LCL+04] whose feasibility has been operationally demonstrated in the context of the Demeter satellite.

On this basis, the goal of the AGATA project is to check off, to understand, to adapt, to develop, and to combine all the technological pieces that are necessary to the development of spacecraft autonomy. To progress in this direction, its short-term objective is to develop a ground simulator of an autonomous spacecraft which will demonstrate, at least in the context of some specific missions, that the current technology allows a spacecraft to be autonomously correctly controlled.

2 A generic architectural framework

One of the first works in the AGATA project was to define what should be the architecture of the software responsible for the control of an autonomous satellite. For that, classical architectures used in robotics were considered. Among them, one can cite the classical three-level planning-centered architectures independently developed at NASA-Ames [MNPW98] and at LAAS-CNRS [ACF+98] and the execution-centered architecture developed at ONERA [BL99]. But, we were especially interested in modular architectures such as the GENOM architecture developed at LAAS-CNRS [FHC94] for the hardware-software interface and the IDEA architecture developed at NASA-Ames [MDF+02]. For the AGATA project, the result has been the definition of a generic architectural framework whose main features are:

1. a decomposition of the system control into hierarchically organized modules;
2. an encapsulation of control and data inside each module;
3. standardized communications between modules via requests, reports about request execution, and information about the system state;
4. a standardized organization of each module around the following four components: tracking of the received requests, tracking of the emitted requests, tracking of the system state, and decision-making upon request emission;
5. a common framework for the interaction between reactive and deliberative tasks inside the module components and especially inside the state tracking and decision-making ones.

One must stress that, although this architectural framework has been developed in the context of the specific AGATA project, its principles are applicable far beyond the space domain, in fact for the architectural design of any autonomous system.

It may be also important to stress that what is discussed in this paper is a functional control architecture and not a software architecture. There are certainly many ways of implementing such an architectural framework.

3 Control decomposition into hierarchically organized modules

The first very simple idea is that, due to the increasing complexity of satellites, it is not reasonable to try and build a unique software module responsible for the control of the whole satellite. As far as possible, independencies must be exploited. This led us to the decomposition of the satellite control into a hierarchy of control modules, each one being responsible for the control of a subsystem.

One can see in Figure 1 a possible decomposition of the control of a satellite dedicated to Earth surveillance and observation. This satellite is equipped with a permanently active detection instrument of wide swath, able to detect phenomena at the Earth surface in front of the satellite, such as forest fires, volcanic eruptions, ... In case of detection, it is able to send an alarm to the ground using the relay of geostationary satellites. Moreover, the satellite is equipped with an observation instrument of narrow swath, active on request and able to take images of the areas where phenomena have been detected. Data produced by this instrument can be downloaded to users on the ground when the satellite is within the visibility of a ground station.

Starting from the top, one can see that the satellite control is, as usually in the space domain, decomposed into a platform control module and a payload control module. Both modules are then decomposed into lower level control modules, each one being responsible for the management of one of the main functionalities in the satellite: orbit, attitude, energy, thermal, telecommand, and telemetry control for the platform, and detection, observation, and data downloading control for the payload. Going deeper, each of these modules is itself decomposed into lower level control modules, called monitors, each monitor being in charge of handling a set of hardware equipments. For example, the GPS monitor is in charge of handling the two GPS receivers present onboard and the thruster monitor in charge of controlling the pool of thrusters that can be activated when one wants to correct the satellite orbital trajectory. Following the ideas of the GENOM architecture [FHC94], the monitors allow the control software to access the hardware via a software interface which is independent from the precise hardware configuration, for example independent from the number of redundant equipments and from the one that is currently used.

In Figure 1, the arcs between modules represent possible requests emitted from one module to another one. For example, the observation module

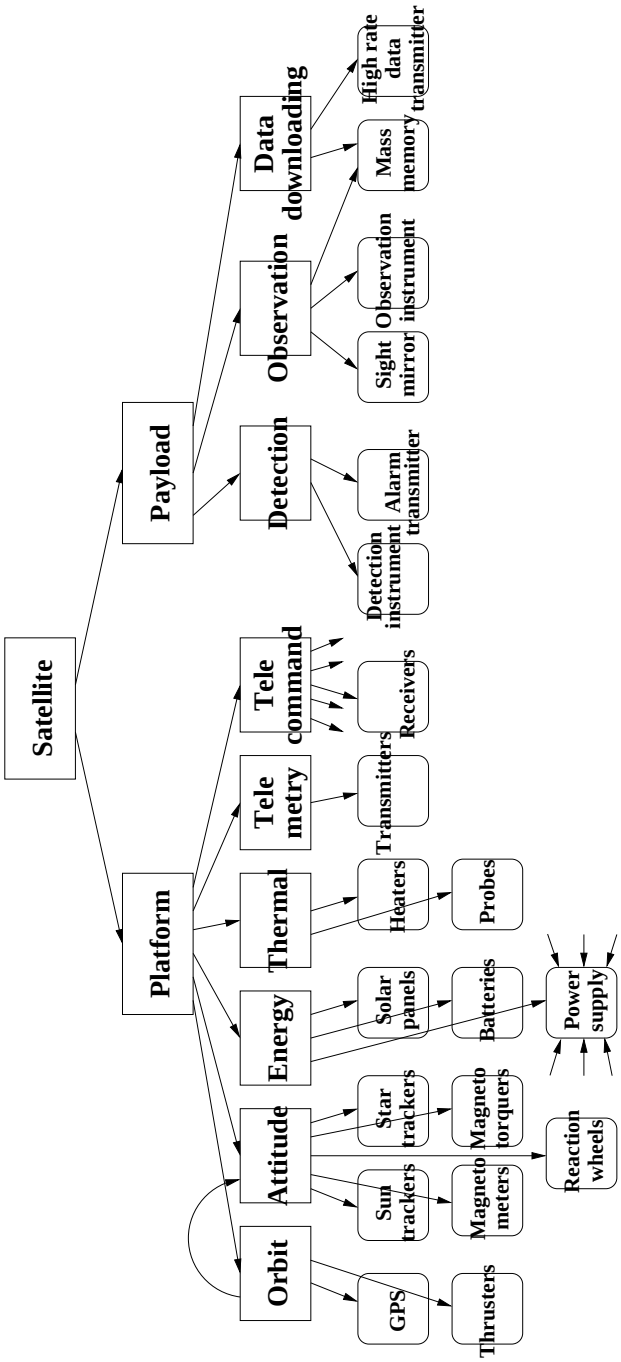


Figure 1: Possible architecture of the control of an Earth surveillance and observation satellite.

can send requests to the sight mirror monitor responsible for directing observation to the right area on the ground, to the observation instrument monitor responsible for triggering observation, and to the mass memory monitor responsible for recording observation data. These arcs result in a directed acyclic graph: no request loop.

But possible dependencies cannot be forgotten. One can see for example that the observation and data downloading modules can both emit requests to the mass memory monitor. These requests must be coordinated by the higher level payload module in order to guarantee that onboard memory be never overflown.

In Figure 1, we only represent the top-down flow of requests from high to low-level modules and do not represent the opposite bottom-up flow of information (reports about request execution and information about the system state) from low to high-level modules. For example, the attitude module gathers information coming from the sun tracker, star tracker, and magnetometer monitors

As dependencies between emitted requests cannot be ignored, dependencies between received information cannot be ignored too. For example, conflicts between information coming from the sun tracker, star tracker, and magnetometer monitors are managed by the higher level attitude module in order to build an estimate of the satellite attitude.

4 Encapsulation of control and data in each module

The second very simple idea is to reuse the principles of encapsulation that are at the basis of *object programming*. In terms of control, that means that, if a module M is in charge of the control of a satellite subsystem S, S cannot be controlled from any other control module M' without a request to M. Moreover, information about the state of S cannot be obtained without an access to the data that are maintained by S.

For example, any request for a change in the satellite attitude must be sent to the attitude module and to no other module, and any information about the satellite attitude must be obtained from it and from no other module.

We think that these principles, although they do not remove all the possible conflicts in terms of requests or information, can greatly help to limit and to manage them.

5 Standardized communications between modules

On this basis, we think that it is possible to standardize the communications between modules, taking into account the main three kinds of exchange that are necessary between them:

1. *control requests* emitted from a module to a lower level one;
2. *request reports* emitted in the opposite direction from a module to a higher level one;
3. *information* about the *system state* from a module to a higher level one.

About requests, one must stress that they are not limited to basic commands immediately and compulsorily executed. Some requests may be complex, such as the regular observation of a ground area. Some are not immediate, such as the observation of a ground area when the satellite will be within its visibility. Some are not mandatory and must executed if possible, such as observations which may conflict with each other. In this case, it may be useful to associate with each request a priority degree which will guide decision-making towards good choices.

About information, one must stress that actual communication mechanisms (systematic information, information on request, ...) and means (message passing, shared memory, ...) depend on implementation choices.

6 Generic organization of each module

Beyond the communications between modules, we think that it is also possible to standardize the organization of each module and to propose a generic organization built around four main components:

1. a *received request tracking* component responsible for receiving requests from higher level modules and for tracking and reporting their execution;
2. an *emitted request tracking* component responsible, in the opposite direction, for emitting requests to lower level modules and for tracking and reporting their execution;
3. a *system state tracking* component responsible for the tracking of the state of the subsystem the module is responsible for;

4. a *decision-making* component in charge of deciding upon the emission of requests to lower level modules in order to answer requests received from higher level modules.

To these main four components, it may be useful to add:

1. a *supervision* component in charge of initializing the module and of managing its possibly different control modes;
2. a *model* component in charge of managing the data that represents the model of the subsystem the module is responsible for; differently from the system state which evolves over time, this model is assumed not to change or to change at a far lower rate;
3. an *information processing service* component which gathers all the data processing services naturally associated with the module, for example the software responsible for orbit, eclipse, and visibility prediction inside the orbit module.

Figure 2 shows the generic scheme of a control module at any level in the module hierarchy.

7 Generic scheme of interaction between reactive and deliberative tasks

An autonomous system must be always correctly controlled in a dynamic environment, with possible changes in the system itself due for example to subsystem failures or in its environment due for example to new observation conditions or new observations to perform. As a consequence, its control must be globally reactive: the control system must be able to react immediately to any event.

Some of the tasks we identified in each control module can be considered as *reactive*, such as received request tracking, emitted request tracking and, in some cases, system state tracking and decision-making. By knowing the maximum event rhythm or by imposing event buffering, we can guarantee that each set of instantaneous events be managed before the following one.

However, some of these tasks cannot be considered as reactive, such as, in some cases, system state tracking and decision-making, or any other complex data processing task. For example, building a failure diagnosis, a predictive resource profile, or an activity plan over a given temporal horizon

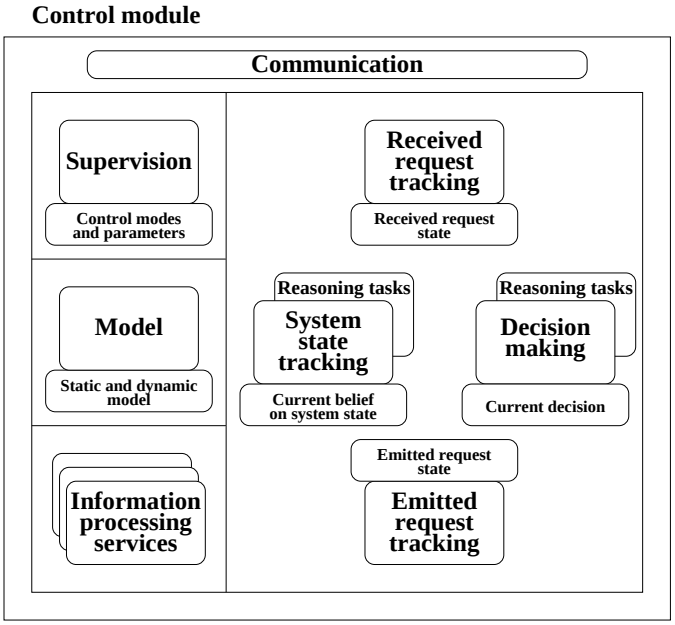


Figure 2: Generic scheme of a control module.

may take a time much greater than the maximum event rhythm or than the buffering rhythm. We refer to these tasks as *deliberative*.

The problem is to define what must be the temporal behavior of the deliberative tasks and what must be their interaction with the reactive ones, if we want them to be useful to the reactive control.

The generic scheme of interaction between reactive and deliberative tasks we propose is summarized in Figure 3. See [LV07] for more details.

According to this scheme, reactive control tasks are in charge of the interaction between the environment, on the one hand, and deliberative reasoning tasks, on the other hand. The latter are never in direct interaction with the environment. Reactive control tasks receive changes from the environment. They may react to them by immediately committing to actions. Note that waiting may be a candidate action. Concurrently, they may compute a deadline for deciding latter on the next action to perform. Then, they may run deliberative reasoning tasks, by providing them with relevant information about changes. On their side, deliberative tasks use this information to produce what we call deliberations, which can be state estimates, failure diagnoses, action proposals, or any other result useful for

decision-making. We assume that deliberative tasks are designed to have an anytime behavior, that is the ability to produce quickly a first result and to improve on it as long as time is available for reasoning. When the deadline occurs, reactive control tasks use the successive deliberations they received from deliberative tasks to make the right decision. If they received no deliberation, they make a reactive default decision.

This scheme requires only that reactive control tasks be able to compute a deadline, to check deliberations before making decisions, to make decisions even when no deliberation has been received, and to perform all of this reactively.

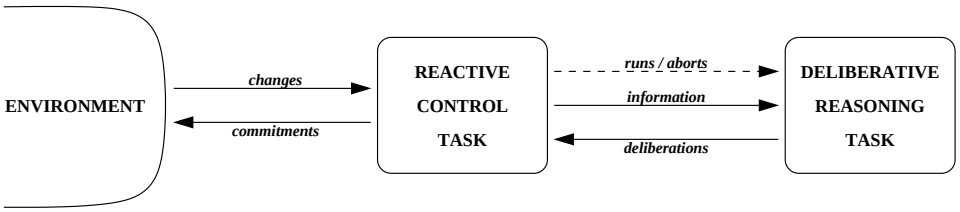


Figure 3: Generic scheme of interaction between reactive and deliberative tasks.

8 Conclusion

We are currently working on applying all these architectural principles to the design and the implementation of a control architecture for an autonomous satellite dedicated to Earth surveillance and observation, such as the one that has been roughly described in Section 3.

The following scenario we will consider is an autonomous agile satellite dedicated to Earth observation, equipped with an optical observation instrument and a cloud detection instrument in front of the satellite, able to provide the module in charge of deciding upon observations with information about the actual cloud cover, in order to avoid imaging clouds, as it is too often the case with currently operational satellites.

Beyond these experiments, we hope that the architectural principles we presented in this paper be applicable for the closed-loop control of many other autonomous systems.

9 Acknowledgements

We would like to thank the people who took part in most of the AGATA brainstorming meetings about architecture, especially Solange Lemai-Chenevier from CNES and Félix Ingrand from LAAS-CNRS.

References

[ACF+98] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An Architecture for Autonomy. *The International Journal of Robotics Research*, 17(4):315–337, 1998.

[BL99] C. Barrouil and J. Lemaire. Advanced Real-time Mission Management for an AUV. In *Proc. of the SCI NATO Symposium on Advanced Mission Management and System Integration Technologies for Improved Tactical Operations*, Florence, Italy, 1999.

[CB05] M.-C. Charmeau and E. Bensana. AGATA: A Lab Bench Project for Spacecraft Autonomy. In *Proc. of the 8th International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS-05)*, Munich, Germany, 2005.

[CCD+05] S. Chien, B. Cichy, A. Davies, D. Tran, G. Rabideau, R. Castano, R. Sherwood, D. Mandl, S. Frye, S. Shulman, J. Jones, and S. Grosvenor. An Autonomous Earth-Observing Sensorweb. *IEEE Intelligent Systems*, 2005.

[CST+05] S. Chien, R. Sherwood, D. Tran, B. Cichy, G. Rabideau, R. Castano, A. Davies, D. Mandl, S. Frye, B. Trout, S. Shulman, and D. Boyer. Using Autonomy Flight Software to Improve Science Return on Earth Observing One. *Journal of Aerospace Computing, Information, and Communication*, 2005.

[DVC05] S. Damiani, G. Verfaillie, and M.-C. Charmeau. Cooperating On-board and On the ground Decision Modules for the Management of an Earth Watching Constellation. In *Proc. of the 8th International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS-05)*, Munich, Germany, 2005.

[FHC94] S. Fleury, M. Herbb, and R. Chatila. Design of a modular architecture for autonomous robot. In *Proc. of the IEEE Inter-*

national Conference on Robotics and Automation (ICRA-94), San Diego, CA, USA, 1994.

[LCL+04] A. Lamy, M.-C. Charmeau, D. Laurichesse, M. Grondin, and R. Bertrand. Experiment of Autonomous Orbit Control on the DEMETER Satellite. In *Proc. of the 18th International Symposium on Space Flight Dynamics (ISSFD-04)*, Münich, Germany, 2004.

[LV07] M. Lemaître and G. Verfaillie. Interaction entre tâches réactives et délibératives pour la décision en ligne. In *Actes des Journées Françaises sur la Planification, la Décision et l'Apprentissage pour la Conduite de Systèmes (JFPDA-07)*, Grenoble, France, 2007.

[MDF+02] N. Muscettola, G. Dorais, C. Fry, R. Levinson, and C. Plaunt. IDEA: Planning at the Core of Autonomous Reactive Agents. In *Proc. of the 3rd NASA International Workshop on Planning and Scheduling for Space*, Houston, TX, USA, 2002.

[MNPW98] N. Muscettola, P. Nayak, B. Pell, and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, 1998.

[WICE03] B. Williams, M. Ingham, S. Chung, and P. Elliott. Model-Based Programming of Intelligent Embedded Systems and Robotic Space Explorers. *Proc. of the IEEE*, 91(1):212–237, 2003.

Really Hard Time developing Hard Real Time
Research Activities on Component-based Distributed RT/E Systems

Etienne BORDE[1,2], Grégory HAIK[1], Virginie WATINE[1], Laurent PAUTET[2]

1 : THALES Land and Joint Systems, 5 avenue CARNOT 91883 Massy cedex France
{firstname.lastname@fr.thalesgroup.com}

2 : Ecole Nationale Supérieure de Télécommunications, 46 rue BARRAULT, 75634 Paris cedex France
{firstname.lastname@enst.fr}

***Abstract**: The development process of distributed real-time embedded systems (DRES) suffers significant limitations when addressing the antagonistic concerns of systems interoperability, flexibility, and reliability. In this paper, we first present a component-based development process and related architecture designed to enable DRES interoperability while improving developer productivity. We then describe the techniques this process uses in order to improve reliability of these systems. The component-based framework is illustrated by a practical use case. Lastly, we present research orientations addressing verification, validation, and certifiability on the one hand, and their ability to tackle with the always-growing flexibility requirements on the other hand.*

As real time embedded systems complexity is growing every year, industry software architects are facing major challenges in terms of development productivity. This issue is addressed by component-based development methodologies, as proposed by the software engineering academic community [1]. Automatic deployment and configuration, code generation, code reuse, better testability and early validation, reduced time to market, easy tuning and mitigating integration risks are some of the benefits of component-based development (CBD). In mid 1990, industry standards have been issued for applying CBD to large scale information systems (Sun's Enterprise Java Beans – EJB [2], OMG's CORBA Component Model – CCM [3]). Unfortunately, CCM and EJB frameworks are not directly applicable to distributed real time embedded systems: scarce computing and memory resources, hardware heterogeneity, real time constraints, performances and assurance issues make CCM and EJB irrelevant for development of DRE systems. Consequently, DRES software engineering community has performed extensive research for adapting component-based development to DRE systems. THALES has been involved at the OMG for defining a variant of CCM that may cover the technical requirements of embedded real time component-based frameworks. The reason of choosing CCM as a starting point was the suitability of OMG's philosophy with the requirements of THALES' clients in terms of interoperability and standard openness. Thus, the main challenge was to make CCM efficient, predictable and low footprint. Moreover, since non functional requirements of DRE systems are very versatile from one system to the other, it is crucial for a DRES component-based framework to be highly adaptable and configurable. Note that a naïve approach to adaptability may deeply impede interoperability.

Research conducted by THALES during the last four years has led to a component framework that is usable and beneficial for industry-standard real time embedded systems. Ongoing research is performed for addressing more application domains, particularly safety-critical, mission-critical or security-critical certified applications requiring deep verification and validation on the one hand, and reconfigurable and multi-mission systems on the other hand.

This paper will first present the outcome of THALES' collaborative research effort performed during the last four years (section 1). The benefits of the resulting component-based framework, namely MyCCM, are illustrated on a real world program MyCCM is being applied to (section 2). Finally, an overview of current research activities is provided in section 3.

1. Component-based Framework for Distributed RT/E Systems

This section will first present the authors' viewpoint on component-orientation, and then describe the principles and the outcome of adapting component-based development to real time embedded systems as performed in the scope of collaborative projects.

1.1. Component-Orientation

Principles

Component-orientation has been introduced to help managing the increasing complexity that Information Systems were facing and to increase the productivity of their development. A component is basically a piece of functionality that can be assembled with others in order to provide the full functional coverage of the system. Allowing to break down the whole system in smaller really independently manageable pieces, easier to develop and to reuse, makes it much cheaper to develop and integrate.

To achieve that goal, components come with three main characteristics:

- A self-described packaging format, so that components can be deployed and configured externally from the application.
- The explicit description, by means of ports, of not only the services that the component is providing (as an object does), but also of the ones it requires for functioning, to be provided by other components; this allows the components being connected externally from the application.
- The separation of the business logics (the components themselves) from the relevant[1] technical support that the infrastructure is providing to them and which is under the responsibility of the containers, parts of the infrastructure aiming at hosting the components.

Even not always put forward, this "separation of concerns" is a key factor to master complexity and to allow an effective reuse, hardly achievable with only object orientation. Actually the two more important factors that prevent reuse are:

- Inability to master the dependencies between a piece of software and others: component model – making this explicit – provides a way to tackle this.
- Inability to master dependencies between a piece of software and its underlying infrastructure: Component/Container model offers a way of structuring this. As a side effect, this model allows to guarantee that the supported technical properties are enforced consistently across the application (e.g., the access control policy is guaranteed to be applied to all components)

Of course, the work to build the appropriate containers is no more of the application developer's responsibility. Component-based infrastructure (e.g., EJB – see below) are thus coming with the tooling allowing to automatically generate the containers, based on standardised descriptions.

This model has been implemented as support for Information Systems. The first consistent implementation has been Sun's EJB (Enterprise Java Beans) which is a real success in this specific application domain. EJB offers support for technical services that are meaningful in this area (Persistence support, Transactional support and Security – to be understood as access control) and is built on top the Java infrastructure.

CORBA CCM (CORBA Component Model) is a more generic and distributed specification of this model, even if still fully dedicated to Information Systems (it offers support for the same technical services as EJB). However, as the general trend of CORBA is to focus more on technical systems[2], CCM is moving in the same direction. This move has already started at OMG, with newly adopted Lightweight CCM specification that defines a CCM profile compatible with embedded targets.

With .NET, Microsoft has also adopted this model, even if the proprietary solution it proposes allows a slightly different, more intrusive and less structured, way of organising the software.

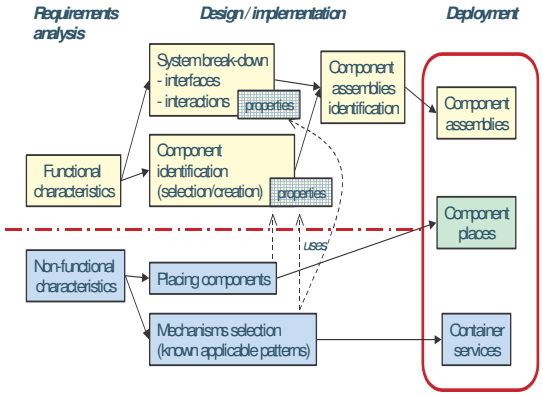


Figure 1: Impact of separation of concerns on development process

Impacts on the Development Process

Separating the functional properties from the non-functional ones and building the whole system by means of assembly, deployment and configuration of components have huge positive impacts on the development process itself, whose main phases are featured on Figure 1:

[1] What is the relevant technical support depends on the application domain.

[2] Cf. all the new specification in the area: Real Time CORBA, Minimum CORBA – special profile for embedded systems, Fault Tolerant CORBA, Data Parallel CORBA...

- It allows a clean role separation: domain experts may focus their functional expertise while platform specialists may define and implement the best technical support.
- Component reuse is made possible.
- As technical support realisation is mainly achieved based on component configuration, main design choices (such as component localisation) can be delayed until the latest and therefore adjusted during the integration, making that costly phase much easier.

1.2. Adaptation to Real-time and Embedded Systems

All the positive impacts of component-orientation would be also very desirable for RT/E systems. However current implementations are not suitable for those systems since they don't provide the relevant non-functional support nor the adequate interaction modes between components and are not meant to accommodate resource constraints, which are here of prime importance.

THALES has therefore decided to adapt an existing standard component model to specific constraints of RT/E systems and to define a dedicated framework. As basis, OMG's Lw-CCM has been selected and its adaptation initiated in the scope of two research projects (IST/COMPARE [7] and ITEA/MERCED [8]).

One important characteristic of the RT/E area is the huge variety of its systems. Trying to provide in a unique framework a solution that would fit all the RT/E systems is not achievable, nor desirable. Therefore, rather than just adding the technical support that would allow to develop the use-cases parts of those projects, it has been decided to focus on extensibility and usability.

Extensibility is achieved by providing the ability to plug within the container, technical support providers, called container services. For that purpose, has been defined an open architecture for the container with specification of dedicated interfaces to insert and configure the container services as well as a packaging format to enable their deployment.

Usability for RT/E systems is primarily conditioned by the ability to get adequate interaction between components. Actually proper interaction support is crucial for it conditions the ability of designing components and the existing one was far from being enough. A new construct, named connector, has been introduced to capture the interaction style. Connectors can then be developed for as many interaction styles as needed. Connectors are actually made of fragments, which are considered as specific container services and can be deployed and configured as such.

Usability also depends on the ability of the framework to accommodate stringent resource requirements. Key elements for that purpose are (i) the ability to tailor the containers at their minimum functional coverage by plugging in only what is strictly needed, (ii) the platform isolation provided by the containers, which allows the use of a huge variety of technical services (including small ones if needed) as well as (iii) the implementation of specific trade-offs to get the better use of the scarce resources. As a proof of concept of the ability to accommodate various platforms, two implementations of the framework have been developed during those research projects: the first one on top of RT-CORBA, the second one on top of OSEK-VDX.

In the scope of these two projects, interaction support for the use-cases has been implemented, as well as some container services providing basic development facilities (tracing...) and timing properties enforcement (locking protocols and appropriate setting of POSIX scheduling parameters).

1.3. MyCCM

Based on these research results, THALES has started to industrialise a framework, called MyCCM (Make Your CCM). MyCCM building blocks are represented on **Figure 2**.

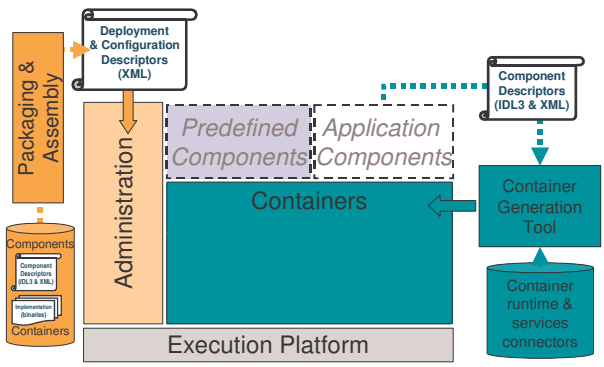


Figure 2: MyCCM building blocks

The central part represents MyCCM runtime, that is mainly two folds: (i) infrastructure and its services mediated by the containers, which in return are hosting structures for the components, and (ii) a tool called Administration, dedicated to deployment, configuration and connection of components – and later on, monitoring and control. On the right side is represented the container generation tool, which generates the containers and

component envelopes[3] based on component description (IDL3 and XML files); on the left side, the packaging and assembly tool which prepares what the Administration needs as input, namely the component packages and the deployment plan.

Adaptation to a particular domain

As it can be seen, the framework is a combination of runtime support and associated tooling. Porting it to a given platform will affect not only the runtime support, but the generated code as well. Therefore attention has been paid to make the code generation mechanisms as flexible as possible.

Besides porting to the target platform, adapting the framework to a specific domain consists just in ***selecting or defining and implementing proper container services and connectors***.

Real Time Issues

Container services are used for configuring time and scheduling parameters of the application, while other services provide real time locking mechanisms. This way, the software architect addresses all the scheduling issues in configuration files. The corresponding activation model is illustrated on **Figure 3**: threads, that may be either ***periodic*** or "***one shot***" are associated to components entry points. One shot threads are meant for interrupt handling as well as any input I/O functions: where necessary, the architect connects one shot threads to a never ending incoming event handler encapsulated in the receiving component.

MyCCM also enables the software architect to set the scheduling parameters of the threads handling the framework-supported communication mechanisms. Such communication threads may have either a (i) ***static*** scheduling configuration as given by the architect, a (ii) ***dynamic*** configuration where scheduling parameters are inherited from the calling activity or finally (iii) ***no-configuration*** – applicable only to components in the same address space – where the calling thread performs the local method invocation. MyCCM uses the underlying OS and middleware (POSIX and RT-CORBA for instance) for implementing these policies.

[3] A component envelope is a generated piece of code allowing a user-written piece of code to be hosted by a container. What is generally called 'component' is the business code (written by the application-developer) plus its envelope.

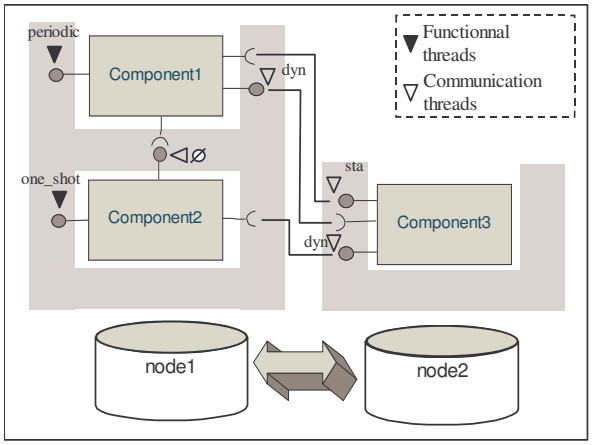


Figure 3 : Threads configurations

2. A Practical Use Case

Beyond experimentations and use case developments performed in the scope of research projects, MyCCM has been recently adopted by THALES for the development of an Infra-Red Search and Track (IRST) Signal Processor called ARTEMIS.

2.1. The ARTEMIS Program

ARTEMIS IRST system has been selected to equip the Future European Multi-Role Frigates (FREMM). It is mainly composed of three infra-red sensors – located around a mast – and a Signal Processor. Each sensor covers a third of the frigate horizon (120°). The three video streams are sent to the IRST Signal Processor (ISP) by a specific protocol on top of UDP. The ISP applies visualisation and tracking algorithms to incoming streams. Data processing is distributed on different single board computers. Each algorithm is encapsulated in a component that communicates either locally or remotely with its counterparts via the communication support of the component framework. For controlling and configuring the components, the framework relies on a free RT-CORBA middleware, while high bandwidth data streams are handled by a fast, unreliable implementation of CCM event support. Finally, processed video and tracks are sent to the frigate's Combat Management System using a dedicated protocol.

A module in a ***UML modeller*** enables the ISP software architect to design the application in a top down approach. ***Scheduling settings*** (priorities and periods) and parameters of components are defined ***in UML models as well***. The software architect may also define functional validation scenarios, or target performance measurement scenarios in the UML modeller. CCM deployment and configuration descriptors are then produced for

each validation scenario. On-target execution of these scenarios is handled automatically by MyCCM deployment mechanisms.

2.2. Benefits for the Software Architect

Usage of a component framework such as MyCCM has good consequences on ***productivity*** of software development. Several factors contribute to increasing productivity. First, MyCCM development process is based on ***intensive code generation***: configuration of scheduling parameters such as POSIX priorities and thread periods, priorities of RT-CORBA communication threads, data marshalling code and method dispatch, and configuration, deployment and connection of application components are simply generated from the architecture description provided as a set of UML models. With MyCCM, the program manager doesn't have to provision development efforts for these tasks. Additional cost reduction resides in the dramatic ***simplification of internal communication protocols***: with handcrafted communications, detailed specifications of datagram/stream formats are necessary; with a communication middleware such as CORBA, equivalent specifications are much more abstract, and consequently simpler. Specification of interfaces is done at operation level, with in/out parameters and exceptions, not at socket level. Moreover, ***generation of threading artefacts*** and ***deployment code*** makes a program relying on MyCCM clearly more productive than a regular CORBA or RT-CORBA application, for which only communication support is generated.

Integration of MyCCM with modelling tools is another productivity factor. Indeed, the software architect is not meant to write complex architectural descriptors by hand: definition of application architecture, design, configuration and deployment is rather done in a UML-based GUI. The first benefit is that graphical models constitute a ***key communication support*** between team members. Moreover, model-to-descriptor generators may come with potentially ***intensive model verification***, which leads to early discovery of model inconsistencies.

Another productivity factor is ***easier testability***: because MyCCM architecture enables the component developer to avoid platform dependencies, the functionalities of such platform-independent components can be validated not only on target, but also on development host. Components might be specifically enveloped and compiled for either the host or the target by just modifying few parameters of the UML architectural description, so that a ***complete functional validation*** can be performed on host.

Finally, integration costs and risks are significantly reduced by the ***late binding*** between application and the platform: when all software components are finally put together for final integration, it is almost inevitable that integration engineers have to fine tune the thread number and their associated parameters (period, priority…). With MyCCM, since all these parameters are in the models and not in the application code, engineers don't have to deeply analyse component code for finding out where threads are created and configured. Another typical situation where this late binding property mitigates integration risks is when hardware happens to be ill dimensioned with regard to software execution times and system timing requirements. Application components only have to be re-targeted (in UML models) to the new hardware. MyCCM will take into account the new hardware environment by generating some other technical code, with no impact on component implementations.

To sum up, component-based development might have very positive impact on productivity of software development. Still, our real-world experimentation with ARTEMIS program has not yet reached the point where productivity benefits might be factually assessed. This point will be reached by the end of all development and integration activities.

2.3. Current Limitations

There are key issues in DRES development that are not currently addressed by MyCCM. For instance, although MyCCM enables software architect to implement priority-based real time systems, no support is provided yet for taking advantage of RT-CORBA support for ***real time network protocols***. Network scheduling must be performed by other means. In today's MyCCM, priority inversion may occur in network stacks.

No support is provided either for proving that the system will meet its end-to-end requirements. This issue is generally referred to as schedulability. Another key issue is proving that the system is deadlock-free. Those two properties (schedulability and deadlock verification) are mandatory for ***high assurance systems***, might they be mission-critical, safety-critical or security-critical.

Yet another issue is that MyCCM only offers a programmatic support for reconfiguration: if needed, ***reconfiguration must be programmed by hand***. A descriptive, framework-supported reconfiguration mechanism would contribute to further increase productivity. Last but not least, ***multi-mission systems*** have specific requirements on software architecture that are not currently addressed by MyCCM.

Next section will discuss these issues and provide the reader with insights about ongoing research activities on these topics.

3. Ongoing Research Activities

The two directions followed by our research activities are verification and validation on the one hand, and support for greater flexibility on the other hand. While verification and validation address safety-critical, mission-critical or security-critical systems, research on flexibility addresses context-aware systems, fault tolerant systems, load balanced systems, autonomous systems and multi mission systems.

3.1. Verification & Validation

Worldwide academic community has performed extensive research towards verification of DRE systems. During the last decades, formal methods such as model checking, computational logics, schedulability algorithms and static code analysis have been foreseen to be able to provide the software architect with advanced tools and methods for verifying and validating DRE systems. Yet, ***these formal methods lack proper integration with the actual execution infrastructure***, making them more difficult to use efficiently. Numbers of properties might be verified, although resource dimensioning is under particular focus in the DRE domain.

Typical resources to be dimensioned are time, memory and energy. The following will focus on time.

Several tools are available to analyse real time properties of the system. Some are based on model checking techniques (UPAAL [14] for instance), others on algebraic methods inherited from Rate Monotonic Analysis, such as MAST [15] and Cheddar [17]. Not surprisingly, both methods have their respective limitations: model checking is subject to exponential state space explosion, while algebraic methods are not able to analyse all possible situations.

Research has been conducted to integrate model checking techniques with component-based development [11, 12, 13]. University of Cantabria (Spain), THALES and ENST (among others) are currently performing research towards integration of CBD and algebraic methods. This requires the user to provide a ***characterisation of the temporal properties of each component***, so that end-to-end execution times can be synthesised. Such end-to-end execution times can finally be provided, together with other parameters such as periods, priorities and deadlines, to the scheduling analysis tool.

As a matter of fact, these temporal characterisations might even be automatically generated by an ***on-target performance measurement*** setting of the component framework, provided that a more abstract behavioural *profile* – omitting numbers – is available. For complex components, execution time measurement scenarios could be generated by ***static analysis of component code*** as developed at CEA-List.

Finally, on-target measurements may also be performed for OS, middleware and network ***traversal timings***, provided that quantitative data flow descriptions are available.

Other verifications might be performed besides schedulability analysis. Most notably, Ocarina [19] has been developed in order to enable the configuration of the middleware from an AADL [16] description. In parallel with the generation of the middleware code, a behavioural model of this middleware is generated into a well formed coloured Petri net in order to verify behavioural properties of the middleware thanks to model checking techniques.

As another verification example, the AADL Error annex is used [18] to model possible faults, their probability of occurrence, and their propagation through the system. Automatically mapped into a stochastic Petri net, this allows analysing fault occurrence and propagation in terms of probability.

THALES is currently involved in ITEA-SPICES[4] project, whose objective is to integrate such verification tools with component-based frameworks. SPICES strategy is to make all descriptors and tools input to be AADL models, so that users may apply various verification and simulation tools to their own AADL application models. The target outcome is a featured component framework connected the relevant verification tools for addressing ***safety-critical avionics applications***.

3.2. Flexibility

Antagonistic with enforcing safety critical requirements, enhanced flexibility of the component framework will ease application developers and – as we will see – users of multi-mission systems to handle variability of the environment, as well as variability of the system itself.

[4] SPICES is an ongoing ITEA funded collaborative project.

Dynamic reconfiguration

Many research activities have been performed to provide dynamic reconfiguration mechanisms, and use case examples are numerous: reduction of functionalities due to a battery or bandwidth limitation, recuperation of functionalities in case of fault recovery, fine tuning and debug of the application, load balancing, user requests, are typical examples that require reconfiguration mechanisms.

A proper reconfiguration support in a component framework is made of both runtime mechanisms and description language. Regarding runtime mechanisms, component frameworks following Lw-CCM specification implement component instantiation and removal, component connection and disconnection. Openness of MyCCM container implies instantiation, removal and (dis-)connection capabilities of container services as well.

Besides runtime support, ***expression of dynamic reconfiguration policies is the real issue***. A distinction might be done between two reconfiguration categories, depending on whether the different configurations are ***statically enumerated*** and fully specified, or if on the contrary configurations are unknown or ***underspecified before actual occurrence*** of reconfiguration. The former case is referred to as pseudo-static reconfiguration, while the latter is simply called dynamic configuration. Each category might be better addressed by a specific description language for reconfiguration policies. Guarded state automaton would be suitable for pseudo-static reconfiguration, while more expressive language would be necessary for dynamic reconfiguration. Note that *programmatic* expression of reconfiguration policies is already available in MyCCM. Reconfiguration logics can be implemented in dedicated components behaving like deployment agents. A key benefit in stepping beyond this programmatic support is to verify and control reconfiguration policies, since reconfiguration is often subject to timing requirements as well.

In [22], for example, the reconfiguration may be delayed depending on the time of service interruption that the reconfiguration process requires. Reconfiguration priority is another key parameter. For instance, the reconfiguration of a cell phone due to a low-battery signal should not impede the end user to receive or emit calls. At the opposite, if reconfiguration is due to the breakdown of a computing resource, the reconfiguration may be more important than the execution of some functional parts of the application. In the scope of Flex-eWare[5] project, reconfiguration capabilities will be added to MyCCM framework.

Multi-mission support

The objective of a framework-based support to multi-mission systems is to ease the configuration of a multi-mission platform system by an end user. Typical example is an Unmanned Aerial Vehicle (UAV) usually used to monitor forest fires that could be reconfigured in emergency for supervision of evacuation operations after an earthquake. In this case, the infrared camera must be replaced by a standard camera, and the system must be reconfigured.

As part of Flex-eWare project, we will design a tool dedicated to ease configuration of multi-mission systems, based on ontology of the application domain. This tool helps the end user to select components providing and using consistent services. The correspondence of services is evaluated within the ontology, relying on ***semantic annotations*** attached to the services. This tool also supports the assembly of those components by ***generating the connectors*** for two components that are semantically equivalent but syntactically different.

The following example constitutes a use case of such a tool: supposing that the two formerly mentioned cameras drivers have different interfaces (different operation names, parameters type, etc…). Note that if the cameras are done by different vendors, the probability of this scenario is high. Still, the platform integrator might have attached to these two camera drivers a common goal in the ontology, and equivalent concepts attached to interfaces as well. Then a mechanism of data representation may be generated to enforce the compatibility of those interfaces.

Very few research works have been undertaken in order to tackle the flexibility issue in the field of DRES. These techniques, based on semantic representation of application domains have mainly been studied for information systems and web services. Indeed, a major issue when using such a technique is to guarantee that the behavioural properties that have been verified on the previous system will still be verified in its new configuration. In the case of critical systems, in which properties are very difficult to check, this issue constitutes an open issue.

[5] Flex-eWare is a ANR (*Agence Nationale de la Recherche*) project focusing on flexibility of component-based systems

4. Conclusion

As shown by the description of FREMM IRST use case, component-based frameworks are already applicable to industry-standard distributed real time embedded systems. Moreover, MyCCM internal design makes it ***easy to adapt*** to the non functional requirements of a particular application domain.

State-of-the-art real time operating systems and communication middleware provide a level of performance that enable component frameworks to address ***highly constrained systems***, such as vetronics and robotics.

Although with positive impact, component frameworks will not solve all the issues of developing hard real time. No matter whether a component framework is used or not: what is hard is to come out with an application design globally satisfying the antagonistic constraints of predictability and resource usage optimisation. However, ***component frameworks have good consequences on productivity*** and potentially integrate verification tools with simulation capabilities in a consistent manner. In other words, component frameworks might become the working environment of real time embedded software architects.

5. References

[1] : C. Szyperski. Component Software: Beyond Object-Oriented Programming. *Adison-Wesley, 1998.*

[2] : T. Valesky, Enterprise JavaBeans: Developing Component-Based Distributed Applications. *Addison-Wesley 1999.*

[3] : Object Management Group. CORBA Component Specification. In: *OMG Document formal/02-06-65, Juin 2002.*

[4] : R. Marvie, P. Merle, and J.M. Geib. Separation of concerns in modeling distributed component-based architectures. In: *proc. of Enterprise Distributed Object Computing Conference, 2002 (EDOC '02).*

[5] : A.D. Birrell, and B.J. Nelson. Implementing Remote Procedure Calls. In: *ACM Transactions on Computer Systems 2, 1 (February 1984): 39-59.*

[6] : Object Management Group. Light Weight CORBA Component Model Revised Submission. In: *OMG Document realtime/03-05-05 edn. (2003).*

[7] : S. Robert, A. Radermacher, V. Seignole and S. Gérard. The CORBA connector model. In: *Proceedings of adaptive and reflexive middleware, 2005 (ARM 05).*

[8] : S. Robert, A. Radermacher, V. Seignole, S. Gérard, V. Watine, and F. Terrier. The CORBA Connector Model. In: *proceedings of Software Engineering and Middleware, ACM digital library, September 2005, Lisbon, Portugal.*

[9] : Object Management Group. Deployment and Configuration Adopted Submission. In: *OMG Document ptc/03-07-08 edn. (2003)*

[10] : A. Bailly. Test et validation de composants logiciels. In: *PhD Thesis in computer science, at Université des Sciences et Techniques de Lille (USTL/LIFL).France, 2005.*

[11] : G. Madl, S. Abdelwahed, and D.C. Schmidt. Verifying distributed real-time properties of embedded systems via graph transformations and model checking (invited paper, submitted).. In: *The International Journal of Time-Critical Computing, 2005.*

[12] : J. Carlson, J. Hakansson, and P. Pettersson.. SaveCCM: An Analysable Component Model for Real-Time Systems. In: *Proceedings of Formal Aspects of Component Software, 2005.*

[13] : J.P. Etienne, and S. Bouzefrane. Vers une approche par composants pour la modélisation d'applications temps réel. In: *Proceedings of sixth francophone conference on Modeling and verification, 2006.*

[14] : K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. In: *Journal on Software Tools for Technology transfer, 1(1-2):134-152, October 1997.*

[15] : M. González Harbour, J.J. Gutiérrez, J.C. Palencia and J.M. Drake. MAST: Modeling and Analysis Suite for Real-Time Applications. In: *Proceedings of the Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001.*

[16] : SAE – Society of Automative Engineers. SAES AS5506. In: *Embedded Computing Systems Committee, SAE, November, 2004*

[17] : F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and Memory requirements analysis with AADL. In: *Proceedings of the 2005 annual ACM SIGAda international conference on Ada, Atlanta, GA, USA, 2005.*

[18] : A.E. Rugina, K. Kanoun and M. Kaâniche. AADL-based Dependability Modelling. In : *LAAS report N°06209, 2006.*

[19] : T. Vergnaud. Modélisation des systèmes temps-réel embarqués pour la génération automatique d'applications formellement vérifiées. In: *PhD thesis, at Ecole Nationale Supérieur des Télécommunications de Paris, France, 2006*

[20] : K. Fujii and T. Suda. Component Service Model with Semantics (CoSMoS) : A new Component Model for Dynamic Service Composition. In : *Proceedings of Applications and the Internet Workshops (SAINTW'04), Tokyo, Japan, 2004, p. 348-355..*

[21] : L. Apvrille. Contribution à la reconfiguration dynamique de logiciels embarqués temps-réel : Application à un environnement de télécommunication par satellite. In : *PhD thesis, Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS (LAAS),Toulouse, France, 2002.*

[22] : E. Schneider. A Middleware Approach for Dynamic Real-Time Software Reconfiguration on Distributed Embedded Systems. In: *PhD thesis, at Université Louis Pasteur, Strasbourg, France, 2004.*

WORKSHOP CAR'07

May 31rst – June 1rst 2007

RTMAPS V3.2

APPLIED TO DISTRIBUTED APPLICATIONS DEVELOPMENT

INTEMPORA S.A.

Introduction

The increasing complexity of robotics systems and the developments of standard wireless communication technologies have opened a new field for the development of distributed applications : autonomous system will rely more and more on cooperative and communicating agents.
Such distributed systems require new development techniques that have to be handled by engineers and researchers.
In the software development field, some of the problems that have to be handled are the following :

- Quality of service for the communication channels
- Software maintainability concerning the communication protocols
- Localization of the agents
- Synchronization and timestamping of the data
- Data exchange latencies
- Distributed intelligence

RTMaps 3 introduced functionalities that simplify the development of distributed real-time and multi-sensor applications. These functionalities have been successfully demonstrated within several projects such as the PUVAME project (a PREDIT project leaded by INRIA Rhône-Alpes : a project for pedestrian detection and collision avoidance on public transport vehicules such as buses or tramways) and the SACARI interfaces from the LIMSI (an immersive 3D environment for remotely controlling a semi-autonomous vehicle).

What is RTMaps ?

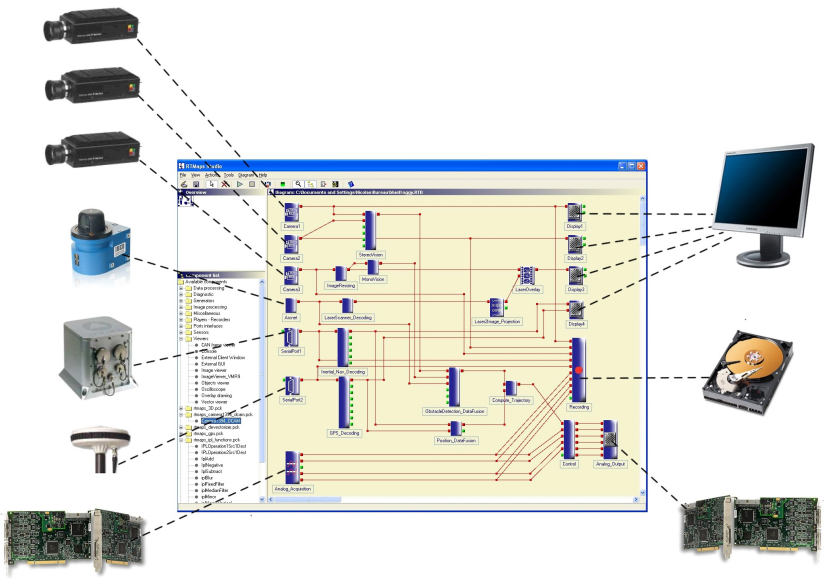


Figure 1 - The RTMaps Studio interface for modular graphical programming

The RTMaps software offers an easy connexion to any type of sensors and actuators, and also the capability to acquire, record, process and transmit data, in real time.

In 2000, the Carsense European project, gathering industries (FIAT, BMW, Renault, Thales, Ibeo) and research labs (INRIA, LIVIC...) looked for a digital data logger and chose a solution developed by the «Centre de robotique de l'Ecole des Mines de Paris»: RTMaps. The objective was the perception of the objects around a moving vehicle. It was the first use of RTMaps. RTMaps is now adopted by important industrial groups (Renault, PSA, Valeo...) and by national and European projects (Arcos, Puvame, REACT...).

Time and measurement play an essential part in the industrial and robotics applications: hence RTMaps precisely timestamps every piece of data at its time of acquisition. This timestamping process provides a complete data flow control during the data processing and replaying. During the tests, situations and behaviours can be recorded and analyzed later. Reproducing a situation is thus possible. RTMaps makes easy the link between real and digital worlds.

Examples of supported sensors: Webcams, DV camcorders, FireWire DCAM digital cameras, analog and digital cameras, stereo-vision devices, GPS, inertial measurements units, radars, laser telemeters, CAN bus devices, analog and digital input/output devices, microphones…

The SDK (Software Development Kit) allows the user to create its own components in C++.

Each component runs in its own thread. The developer is released from the problems of data protection and inherent concurrent accesses of multithread applications. Many data exchange policies between components are integrated (circular buffers, unblocking, re-sampling, etc...), thus offering the behaviour fitting to each application type (recording, real time processing, data conversion, control...).

Distributed RTMaps

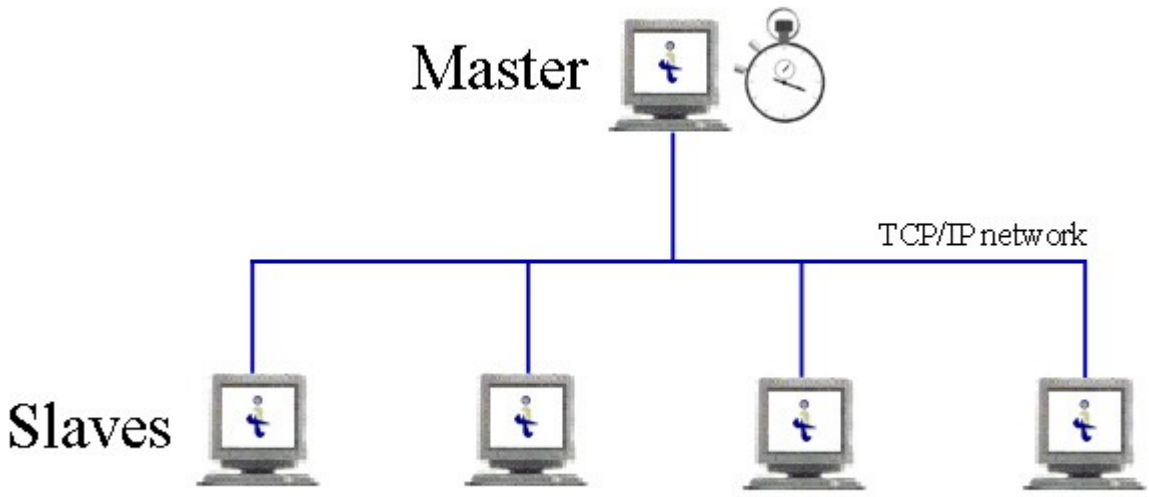


Figure 2 - Distributed RTMaps instances over a TCP/IP network

In order to allow quick and easy development of distributed applications, version 3 of RTMaps introduced functionalities aimed at :

- Synchronizing the clocks of distant RTMaps instances. This functionality enables to perform synchronized recordings on several computers,
- Exchanging data between different RTMaps instances.

These functionalities will be helpful for :

- Increasing the available computing resources by distributing the different parts of an application on several computers : indeed, when developing complex applications, a single computer can be limited in resources. These limitations can come from connectivity (number of USB or PCI slots for example), CPU computation power (several complex algorithms such as image processing, neural networks algorithms etc… may not have enough time to run), bandwidth (when exploiting several high speed data sources such as cameras, a standard PCI bus can be overloaded, standard hard disks may not be able to record more than 50 MB/s also).
- Allowing distant cooperative applications that may exchange data and information between them: different sensors may be placed at distant locations and may not have the ability to be connected to a single PC due to cabling issues for example, or moving agents may need to establish wireless connections between them or with the infrastructure.

Clock sharing

Data timestamping is one of the major focus when developing data fusion applications: knowing when such piece of data has been acquired, when such event has occurred, etc… is particularly critical when exploiting several sources of data within a same data-fusion algorithm.
Latencies can be introduced while data pre-processing or data transmission/reception before the data fusion step, but even if the different data sources are affected, it is particularly important to preserve the timestamp of the origin of each peace of data in order to:

- Allow re-synchronization of the data flows for the data-fusion steps
- Allow extrapolation of a signal state in time
- Be able to discard some pieces of data when they are too old

RTMaps was made for dealing with such constraints when using multiple asynchronous data sources with the following functionalities :

- Sharing of the same timebase among all the components of a single diagram
- Integrated multi-threaded handling of the different components for dealing with asynchronous data sources and events
- Double timestamping functionalities

A distributed application is also subject to such constraints, which are even more difficult to implement. Indeed, the clocks (or timebases) of the different distributed software modules have to be synchronized together so that it can be considered that a single clock is shared among all the computers.

Characteristics of a clock
A clock is a sort of service that can be queried at any time for the current time.
A clock can be characterized by a resolution, an offset to a parent clock and a drift to a parent clock.

- the resolution is the smallest measurable amount of time of the considered clock

- the offset of a clock is the time (in the parent clock timebase) when the clock starts running (this offset can be for example set so that an application clock starts at time 0 when the application starts running).
- the drift is the difference of "time speed" between the parent clock and the child clock. This drift can be due to the time measurement technology used by the child clock (quartz oscillators on which computer clocks are based, are provided with drift characteristics, the drift of quartz oscillators is also subject to change with their temperature for example), or to a controlled behaviour applied to the clock (such as rewind functions which end-up in a negative time speed, fast forward (2X, 4X, etc…), slow forward, pause (time speed = 0), etc…

Note: It can be considered that the "root" clock (the parent of all clocks) is the International Atomic Time (a time scale based on the definition of the second based on the Cesium 133 radiation frequency), on which is based the UTC (Coordinated Universal Time) which differs from the IAT by a round number of seconds. The UTC time is the time base for the GPS system, itself using multiple atomic clocks.

Since the computer clocks (performance counter or system clock under Windows, and Real Time Clock or system clock under Linux for example) are not perfect, and may start with different offsets, there is a need to setup a new clock layer (let's call it the "layer 2 clock") on top of them in order to be able to get a synchronized clock distributed over several computers. This layer 2 clock on each computer will have to:

- first adapt its offset on top of the computer's parent clock so that all the layer 2 clocks can be coordinated at start.
- then permanently correct this offset due to the unknown drifts of the clocks of each computer between them.

Clocks architecture in RTMaps

In an RTMaps distributed environment, one of the instances has to be chosen for being the Master, all the other instances will have to connect to this master as Slaves via TCP/IP.
The Master will impose its own clock among the network to all the connected slaves. It will have the ability to start or shutdown the applications as well as to change the evolution of time (rewinding, pausing, accelerating or slowing down, etc…).
The following pictures shows the 3 layers clocks architecture of an RTMaps based distributed system:

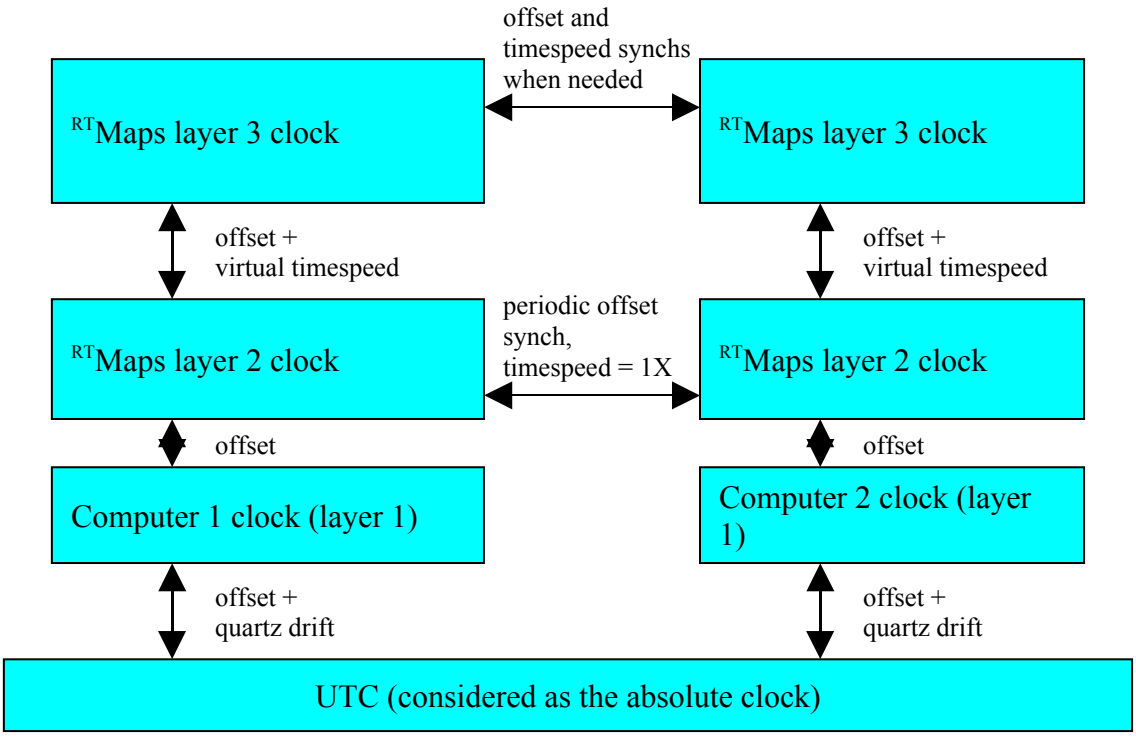


Figure 3 - The 3 layers RTMaps clocks architecture

The computer clocks (layer 1) that RTMaps uses are usually based on the performance counter under Windows (the counter of the CPU cycles) or the RTC clock under Linux. This clock starts at 0 when the computer starts up.
They may have quite an important drift compared to a computer's system clock but they offer the best resolution.

The RTMaps layer 2 clock handles an offset which is periodically updated on the RTMaps Slave computers in order to be synchronized to the RTMaps layer 2 clock of the Master system. By periodically correcting these clocks offset, it ensures a correction of the drifts of the layer 1 clocks which cannot be measured.

The RTMaps layer 3 clocks are then based on clocks that are synchronized together, and that can then be considered as a single clock. So they have to share an offset and a time speed, dictated by the Master system, in order to remain synchronized even in replay mode (start, shutdown, fast-forward, pause, time jumps, etc…)

The synchronization protocol

In order to synchronize the layer 2 clocks between them, the Slave computers periodically exchange synchronization frames as soon as they are connected to the Master. The synchronization protocol is an NTP-like protocol which is detailed below:

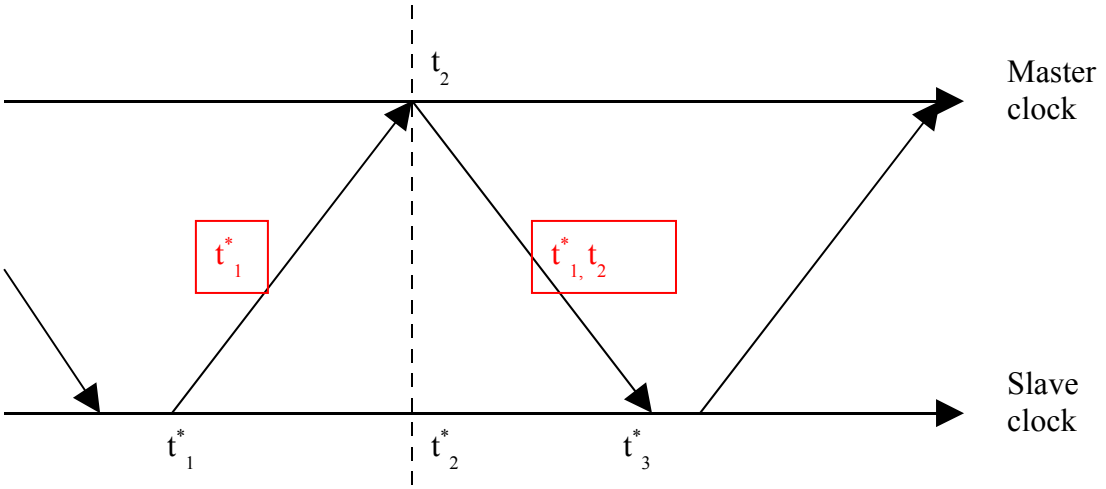


Figure 4 - Synchronization protocol

1. A first synch frame containing the emission time in the slave's clock time base (t^*_1) is emitted to the master.
2. The master receives the frame, queries for its own current time (t_2) and appends it in the frame which is sent back to the slave.
3. The slave receives the answer from the master and queries its own current time again (t_3).

If we consider that the travel time of the frame to reach the master is the same than the one spent to come back to the slave, then t*2 can be easily computed and the offset to apply to the slave clock also:

$$t^*_2 = (t^*_3 - t^*_1)/2$$

$$\Delta_t = t2 - t^*_2$$

Now, the first assumption we made considering that the travel time of the synchronization frames was equal on the way to the Master and on the way back is not correct on a standard TCP/IP network where collisions can occur and prevent a frame from being sent at once on request.

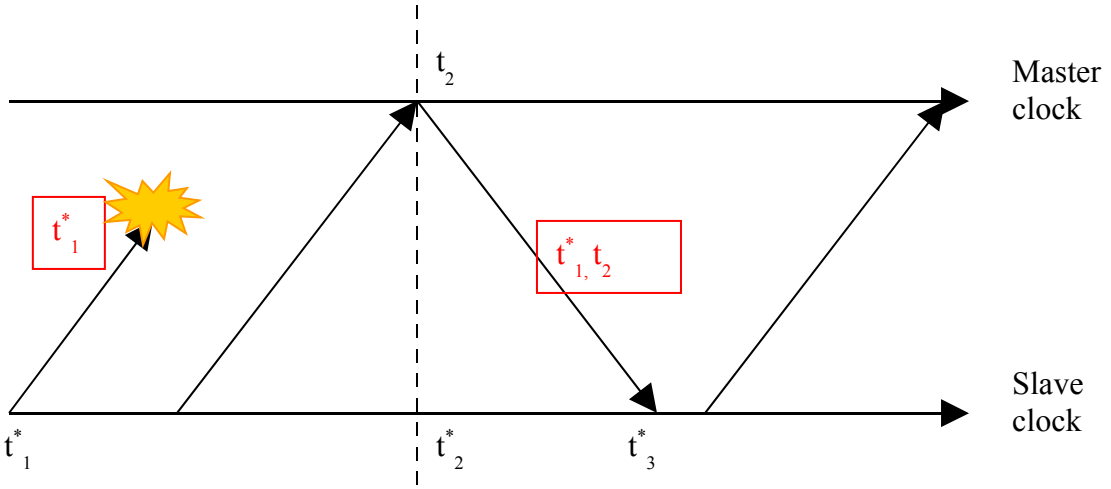


Figure 5 - Synchronization frame subject to TCP collision

In order to achieve correct synchronization anyway, it is possible to send N synchronization frames in a row, and discard every one of them but the one that performed the shortest travel time: then it is possible to assume that this particular frame has not been affected by any collision. The synchronization operation (the offset correction) can then be performed based on this particular synchronization frame, discarding the other ones.

Synchronization accuracy

Some measurements have been performed on two types of standard TCP/IP networks: a 100Mbits/s Ethernet network which is made of several desktop computers connected in an office LAN, and an ad-hoc Wi-Fi network with a WSA protection key.
The test computers were running Microsoft Windows XP SP2.
The synchronization frames were sent 10 times per second and only 1 out of 10 frames was kept for the synchronization operation which ended up in an offset correction once per second.
The following charts present the offset that were applied each second on the slave's layer 2 clock:

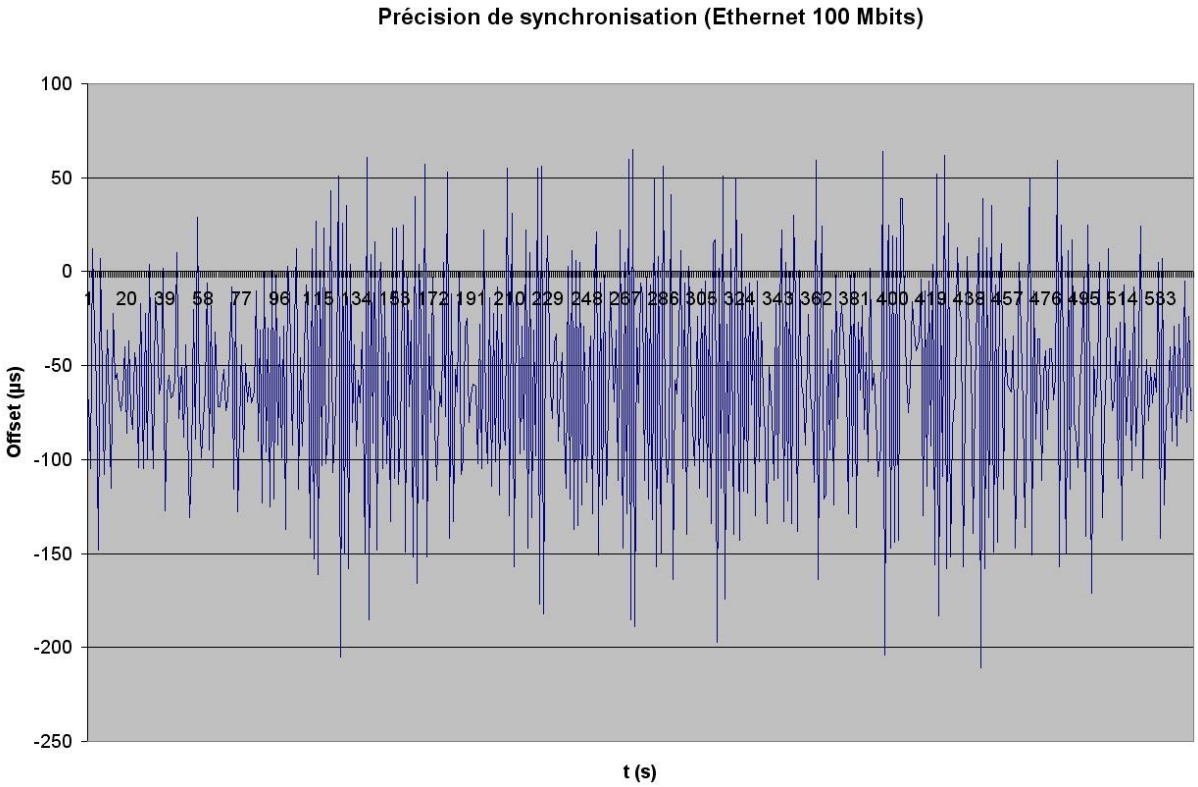


Figure 6 - Synchronization accuracy on a 100 Mbits ethernet LAN network

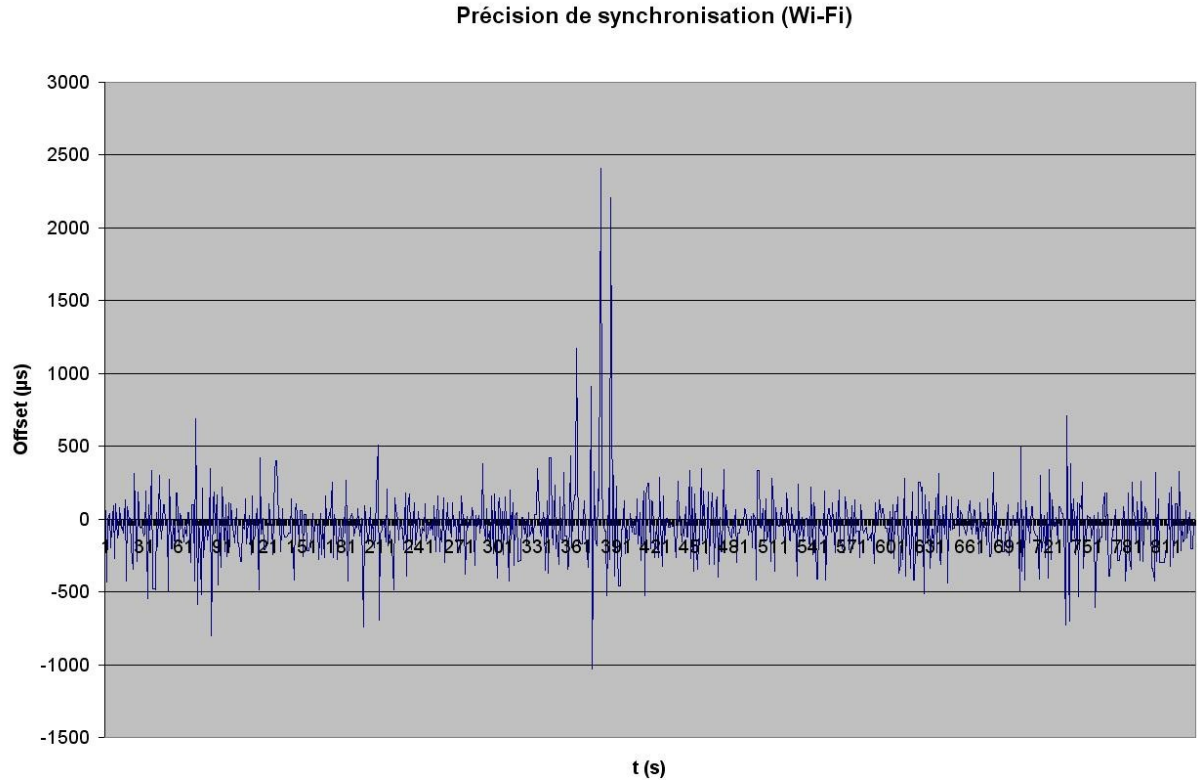


Figure 7 - Synchronization accuracy on a Wifi ad-hoc network with WSA encryption

It shows that on an Ethernet network, the achieved accuracy for the synchronization is around 200 microseconds at worst. The synchronization on a Wifi network is less accurate due to

frames losses and lower quality of service, which may increase very much the variance of the synchronization frames travel time.
The other interesting point is the mean value of the offset that is equal to approximately 50 µs/sec: this constitutes an estimation of the drift of the layer 1 clock of the Slave computer related to the layer 1 clock of the Master computer. Such a drift ($5x10^{-5}$) is quite important for a standard oscillator: the specification of the quartz on which are based the system clocks of standard PCs mention a drift of approximately 10^{-6} related to the absolute UTC time (1 µs/sec). This is due to the fact that in order to achieve the best resolution on data timestamping, the RTMaps layer 1 clock uses the performance counter (counter of the CPU cycles) instead of the system clock.

Synchronizing unconnected computers
Whenever no TCP/IP connection can be established between computers, other synchronization methods can be setup, such as GPS based synchronization method. RTMaps V3 components now have the ability to implement a complete clock.
On each diagram, when a component exposes the clock ability, it can be selected to run the RTMaps clock for the whole process.
A component can then perform GPS data acquisition along with the PPS signal acquisition in order to be accurately synchronized periodically directly with the UTC time, and provide this time to the whole RTMaps application.

Data flows communications

Data exchange between several RTMaps instances can be performed independently of the Master-Slave(s) functionalities via specific Socket components.
Such components have the ability to serialize any RTMaps piece of data (images, CAN frames, stream data, numerical data, etc…) and send it to other instances of RTMaps via TCP, UDP or Multicast in order for it to be used within a distant application.
Socket components also have the ability to transmit the timestamps along with the pieces of data, so that if the RTMaps clocks are synchronized, the data timestamps are still coherent from one computer to another. Data fusion operations can then be performed correctly on the entire RTMaps network.

Application

The PUVAME project

The PUVAME project (http://emotion.inrialpes.fr/puvame/) objective is to detect dangerous situations that may lead to collisions between pedestrians and public transport vehicles such as buses or tramways.
The developed system has to be able to detect dangerous situations thanks to the on-board sensors system and, if available, to match the information with infrastructure sensors systems such as infrastructure cameras that would be installed in dangerous areas such as crossroads or bus stops.

In order to be able to develop such a cooperative system between on-board systems and infrastructure systems, a Cycab from INRIA has been equipped with several sensors:

- a FireWire DCAM camera,
- a Sick LMS lidar,

- ultrasonic sensors (Microsonic Trans'o Prox)

whereas the experimentation site was equipped with 6 analog cameras installed on masts that could cover the entire site.

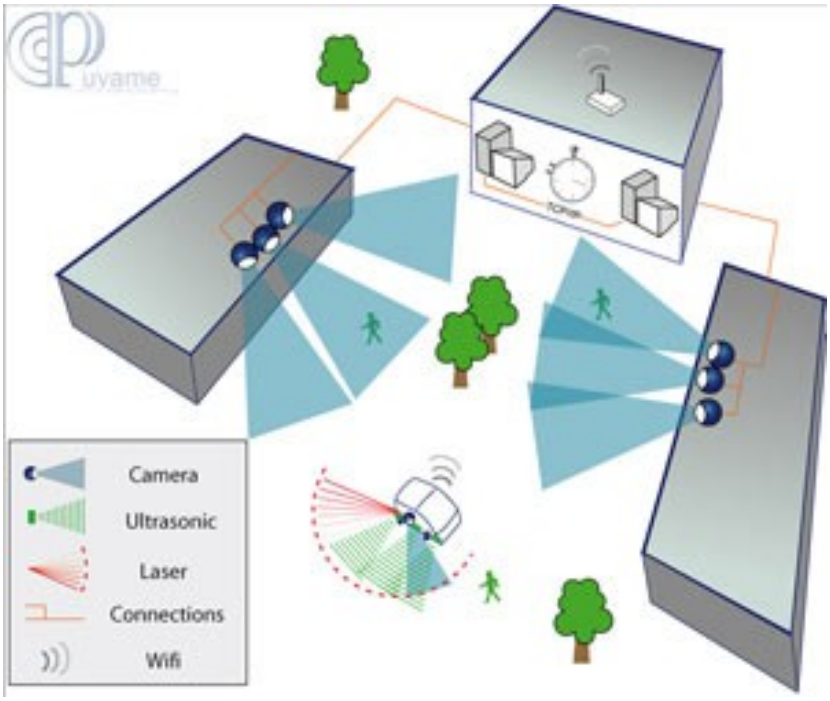


Figure 8 - The PUVAME experimentation system

Infrastructure system had to perform pedestrian detection, vehicle detection, and trajectory prevision of all the actors in order to compute a risk estimation.
The on-board system also performed its own risk estimation based on the on-board sensors informations. Based on the result of data fusion between the on-board system results and the infrastructure system results, reliable alerts could be delivered to the vehicle driver whenever dangerous situations occurred.

In order to be able to develop such a system in cooperation with different partners, there was a need to record the raw data in a synchronized way (both in vehicle and on the infrastructure) and to be able to playback the scenarios in order to develop the detection algorithms and the communication protocol.
In order to quickly setup such a distributed system, an RTMaps software has been installed in the 3 PCs (1 on-board, and 2 on the infrastructure). A Wifi connection has been setup between the vehicle and the infrastructure LAN, and the vehicle software has been set as Master whereas the infrastructure systems were setup as Slaves.
Figure 9 - The PUVAME playback diagram - synchronized playback of 6 infrastructure cameras and on-board sensorspresents the RTMaps diagram which performs the synchronized playback of a PUVAME scenario: 3 Player components are placed on a single diagram, each of which replays a databases that has been recorded on one of the 3 datalogger PCs.

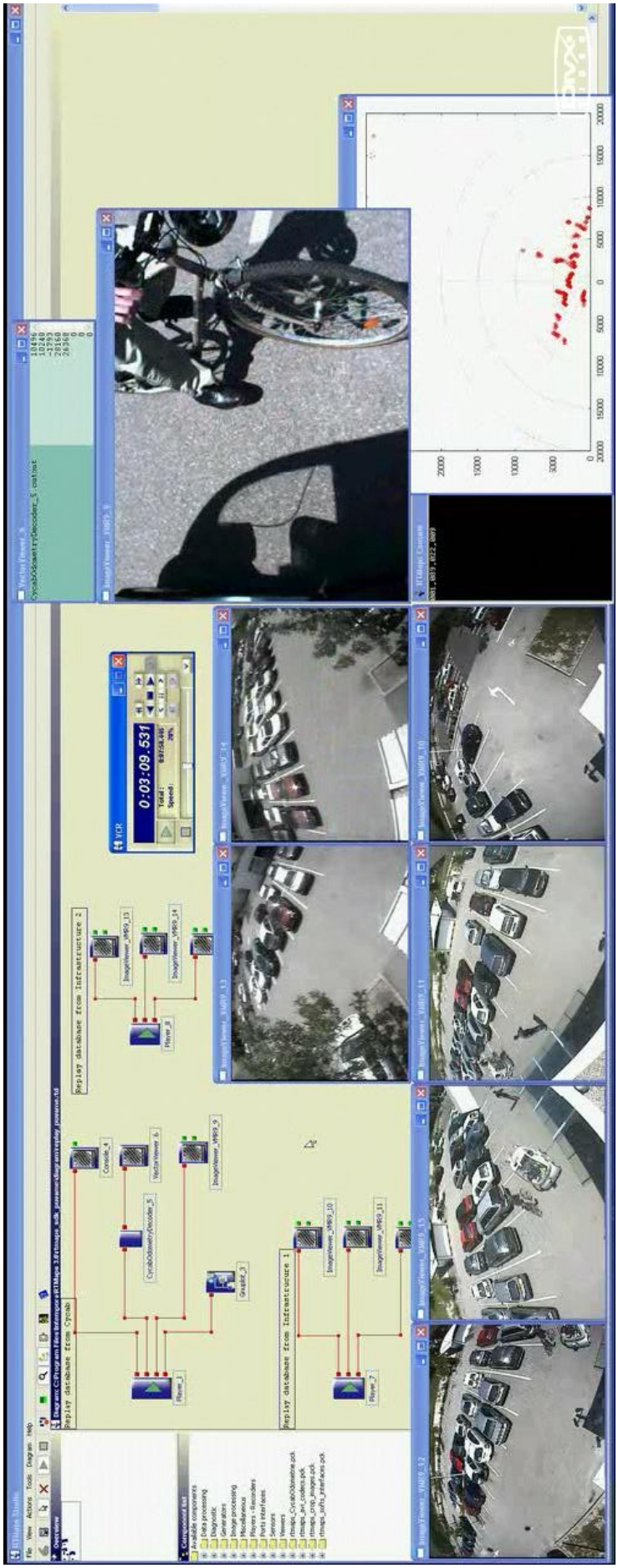


Figure 9 - The PUVAME playback diagram - synchronized playback of 6 infrastructure cameras and on-board sensors

The SACARI interface

The SACARI interface (Supervision of an Autonomous Car in an Augmented Reality Interface), developed by the Venise3D team of the LIMSI laboratory at University Paris XI – Orsay, is an immersive environment aimed at allowing remote driving of a semi-autonomous vehicle.
The whole system will be constituted of the robotized vehicle itself, developed by the IEF laboratory, connected through Wifi to the remote driving room at the LIMSI.
This project involves technologies such as multi-modal interfaces, virtual-reality, augmented reality, multi-sensor perception and vehicle control.

A distributed environment based on [RT]Maps has been setup here for:

- in-vehicle sensors acquisition and transmission to the remote driving immersive room
- setup cooperation in the immersive room between multi-modal systems such as 3D display, head tracking and 3D mousse tracking, 3D audio rendering, voice recognition etc…
- transmission of commands to the robotized vehicle and semi-autonomous control of the vehicle.

More information can be found on
http://www.limsi.fr/Scientifique/venise/ActionVenise/Sacari_presentation.html
http://www.limsi.fr/Scientifique/venise/ActionVenise/posterCGGM.pdf
and on the Intempora website :
http://www.intempora.com/ENG/references/limsi/limsi_cnrs_venise.php?nav1=ref&nav2=proj

BIBLIOGRAPHY

[RTMaps] : a framework for prototyping automotive multi-sensor applications / Bruno Steux, Pierre Coulombeau, Claude Laurgeau / ENSMP CAOR|Centre de CAO et Robotique – ENSMP, 2000

PUVAME - New French Approach for Vulnerable Road Users Safety, Intelligent Vehicles Symposium, 2006 IEEE
Aycard, O. Spalanzani, A. Yguel, M. Burlet, J. Du Lac, N. De La Fortelle, A. Fraichard, T. Ghorayeb, H. Kais, M. Laugier, C. Laurgeau, C. Michel, G. Raulo, D. Steux, B.
GRAVIR-IMAG & INRIA RA - Grenoble - FRANCE. Email: Olivier.Aycard@imag.fr;

SACARI: An Immersive Remote Driving Interface for Autonomous Vehicles. International Conference on Computational Science (2) 2005: 339-342
Antoine Tarault, Patrick Bourdot, Jean-Marc Vézien

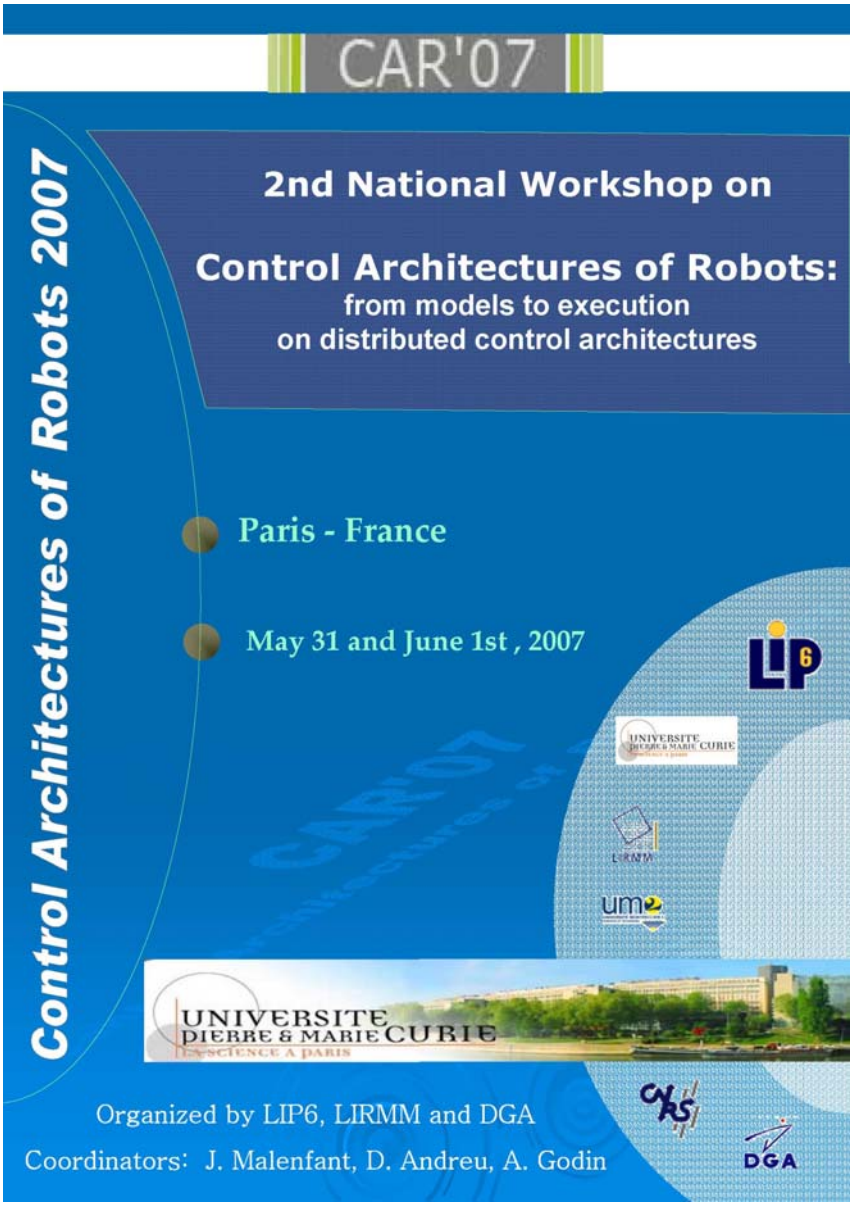
The use of 3D-audio in a multi-modal teleoperation platform for remote driving/supervision
Katz, B. F. G. / Tarault, Antoine / Bourdot, Patrick / Vézien, Jean-Marc
AES 30th international conference on intelligent audio environnements, Finland, march 2007

CONTACTS

Gilles Michel : gilles.michel@intempora.com, +33 1 41 90 03 59
Claire Delaunay : claire.delaunay@intempora.com, +33 1 41 90 08 42
Olivier Meunier : olivier.meunier@intempora.com, + 33 1 41 90 08 42
Nicolas du Lac: nicolas.dulac@intempora.com, +33 1 41 90 08 42

Session

« Validation and verification of control architectures »



Formal assessment techniques for embedded safety critical system

Christel Seguin, Pierre Bieber, Charles Castel, Christophe Kehren
ONERA - Centre de Toulouse,
2 avenue E. Belin
France
{firstname.name@onera.fr}

Keywords: formal models, safety assessment, embedded systems, AltaRica, robotic systems

Abstract

Recently, ONERA was involved in the ISAACS European project. The aim of this project was to investigate new safety assessment techniques based on the use of formal design languages and associated tools. ONERA studied more specifically the applicability of the AltaRica language and the Cecilia OCAS environment to perform the safety assessment of some Airbus aircraft systems. In this paper, we first recall the methodology developed for such traditional embedded safety critical system. Then we discuss its applicability to robotics systems.

Introduction

During the last three years, ONERA was involved in the ISAAC (Improvement of Safety Activities on Aeronautical Complex Systems) European project (ref. 1, http://www.isaac-fp6.org/). This project aimed at developing safety assessment techniques based on the use of formal specification languages and associated tools. So called formal models are traditionally used to specify the expected normal behaviours of software based system. ISAACS partners investigated first how to generalize such models to deal with faulty behaviours of various kinds of systems. Then they proposed new tools or new uses of existing tools to check whether the generalized formal models met qualitative safety requirements. These tools provide not only interactive simulation capabilities but also take advantage of formal language features to support advanced capabilities such as model-checking or fault tree generation. The approach was validated on some existing aircraft systems.

In this paper we first present how AIRBUS and ONERA tackled modelling and safety assessment of aircraft systems using the AltaRica language and a subset of the associated tools. Moreover, the concepts are illustrated by a case study inspired by the hydraulic system of the Airbus A320.
Safety assessment based on formal models raised two main issues. The first issue is to get formal system models meaningful for safety analysis whereas models more often used are fault trees. ISAACS partners are interested in failure propagations in complex dynamic systems. So they consider formal notations for reactive systems, used to support system design such as Statechart (models are automata), Scade (models are equations between synchronous data flows) or dedicated to safety such as AltaRica (models mixing automata and equation concepts). To cope with failure propagations, system models can either be produced by system designers and then extended with failure modes specified by safety engineers, or can directly be produced by safety specialists using libraries. It is worth noting that formal models of failure propagations should have the correct granularity level to ease model exploitation. On one hand, advanced simulation capabilities have good performances when the analyzed model does not go into detailed arithmetic computations. On the other hand, a correct granularity is reached when the scenarios, leading to a failure condition, extracted by the tools are similar to what safety analysts would have envisioned if they had to design a fault tree. In order to get the appropriate granularity at first shot, we chose to define libraries of AltaRica components that focus on failure mode propagation and abstract details of nominal behaviours.
The second issue is related to the choice of the adequate techniques for assessing qualitative safety requirement of complex dynamic systems. Interactive simulation facilities enable to perform a preliminary bottom up analysis since failures can be injected and their effects computed not only locally but at system or even aircraft level. This will be detailed later on. Top down analyses are guided by qualitative requirements such as “no single failure leads to the system loss”. We propose to use model-checkers to assess such kind of requirements. They perform “exhaustive” simulation to check whether a requirement is always met. Moreover, they can distinguish subtle temporal situations such as a transient loss of a function (during a recovery phase for instance) from a permanent one.

The paper has the following structure. First section describes one traditional safety critical aircraft system: a hydraulic system inspired by A320 system. We focus on its safety requirements and architecture. Section 2 introduces the AltaRica language through examples. We explain the modeling philosophy used to build the hydraulic formal model at a satisfying granularity level for safety assessment. Section 3 deals with the benefit of advanced simulation capabilities to assess qualitative safety requirements on dynamic models. We show how the models were analyzed using interactive simulation facilities of Cecilia OCAS and SMV (Symbolic Model Verifier) model-checker. The last section discusses the applicability of such formal assessment techniques for robotic systems.

Case-study Presentation

The role of the hydraulic system is to supply hydraulic power to devices which ensure aircraft control in flight like the flaps, slats, or spoilers as well as devices which are used on ground like the braking system. As the loss of devices powered by this system could lead to the loss of aircraft control, the main safety requirement of this system is:

> A total loss of hydraulic power is considered to be catastrophic. The probability of occurrence of this failure condition should be smaller than 10^{-9} per flight hour and no single failure should lead to this failure condition.

The hydraulic system is mainly composed of three independent sub-systems which generate and transmit the hydraulic power to the consumers. Three kinds of pumps were used in the model of an A320-like hydraulic system. The first one is the Electric Motor Pump (EMP) which is powered by the electric system, the second one is the Engine Driven Pump (EDP) that is powered by one of the two aircraft engines and the last one is the RAT pump that is powered by the Ram Air Turbine. The hydraulic system also contains other types of components such as tanks, valves and gauges.

To meet its main safety requirement, the system is constituted of three channels: Green, Blue and Yellow. The Blue channel is made of one electric pump EMPb, one RAT pump and two distribution lines: prioritary (Pdistb) and non-prioritary (NPdistb). When priority valve PVb is closed consumers connected to Npdistb do not receive hydraulic power. The Green system is made of one pump driven by engine 1 EDPg and two distribution lines Pdistg and NPdistg. The Yellow system is made of one pump driven by engine 2 EDPy, one electric pump EMPy and two distribution lines Pdisty and NPdisty. Moreover a reversible Power Transfer Unit (PTU) transmits pressure between green and yellow channels as soon as the differential pressure between both channels exceeds a given threshold.

These components are controlled by crew actions and reconfiguration logics. The RAT is automatically activated in flight when both engines are lost. The EMPb is automatically activated when the aircraft is in flight or on ground when one engine is running. EMPy is activated by the pilot on ground. We assumed that EDPy, EDPg were activated whenever the corresponding engine was started.

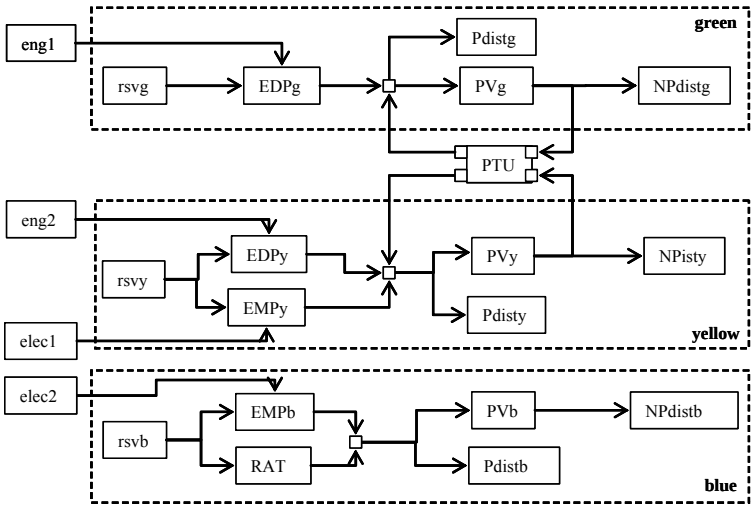


Figure 1 − Hydraulic System Architecture

System Modelling in AltaRica

The AltaRica Language: AltaRica (ref. 2) is a formal language developed at LaBRI (Laboratoire Bordelais de Recherche en Informatique) for modelling both functional and dysfunctional behaviours of systems. Thanks to the language well defined semantics and syntax, safety assessments of AltaRica models can be analysed by numerous reliability or validation tools. Moreover, its capacity to realise compositional and hierarchical models is a great advantage when complex systems must be modelled. The development of AltaRica models is supported by Cecilia OCAS workshop (ref. 3) of Dassault Aviation that provides graphical edition and simulation facilities and integrates fault tree generators. We will now briefly describe this language.
Each system component is modelled by a "node". A node is a mode automaton (ref. 4) defined by three well identified parts. First part is the declaration of the different kinds of node parameters: state, flow and event. States are internal variables which memorize current functioning modes (failure modes or normal ones). Flows are node inputs or outputs. Possible types of states and flows are integer interval, enumeration and boolean. Events are phenomena, which trigger transitions from an internal state to another. They can model pilot actions or the occurrence of failures, or reactions to input conditions (the key word "no_event" is used in this case). This particular event plays a significant role when modelling the impact of cascading failures in the system.

The second part describes the automaton transitions. A transition is a tuple `g |- evt -> e` where `g` is the guard of the transition, `evt` is an event name and `e` is the effect of the transition. The guard is a boolean formula over state or flow variables. It defines the configuration in which the transition is fireable if the event `evt` occurs. The effect `e` is a list of assignations of value to state variables. So the transition part describes how functioning or failure states can evolve.

The third part is a set of assertions. Assertions are atomic equalities or more structured equations using `if-then-else` or `case` construction. They establish relations between the states and the flows of the component and so, describe how component outputs are determined by component inputs and current functioning mode.

These concepts are illustrated by the following example. The component `block` has one `input`, one `output` flow ranging over the domain `{no, low, ok, max}`, one boolean internal state `ok` and one `failure` event. The transition means "if the system is `ok` and if `failure` occurs then the system is no more `ok`". The assertion means "if the system is `ok` then the `output` is equal to the `input` else the `output` value is `no`". We used here the `case` structure but we could use similarly an `if then else` structure.

```
node block
 state
  ok : boolean;
 flow
  input : {no, low, ok, max} : in;
  output :{no, low, ok, max} : out;
 event
  failure;
 trans
  ok |- failure -> ok:= false;
 assert
  output = case {ok : input,
  else : no};
edon
```

In a system model, instances of such nodes are interconnected by assertions which plug input-output flows. Hierarchy of nodes can be used to build complex components and structure the system model.

Case-Studies Modelling: The main step prior to model a system is to collect information on it (e.g. architecture, failure modes). We particularly paid attention to Airbus Functional Hazard Assessment document performed on aircraft functions that describes the failure conditions, effects and severity levels (i.e. catastrophic, hazardous, major or minor) and to the System Safety Assessment which demonstrates that safety objectives are met. In this section we

describe how to model a system using these documents as inputs and use the example of the hydraulic pipe as an illustration.

Failure Modes: Failure modes that could cause the loss of energy supply were modelled. We considered that all components could fail to generate, transmit or deliver energy. We also supposed that leaks could occur in pipes. Finally, blocked positions for valves and PTU were also considered. Table 1, hereunder, sums up the failure modes considered in our component libraries.

Components	**Failure modes**
Pipe	Leakage
Reservoir	Failed, leakage
Pump	Failed, overheat
Valve	Failed, stucked
Consumer	Failed, leakage
PTU	Failed, stucked

Table 1 − Failure Modes of the hydraulic components

Failure Propagation: We have to know what kinds of information are exchanged between the components and how failures will be propagated through pipes. This information is really specific to each system. If we consider a hydraulic circuit pipe we cannot model a leakage only by considering the absence or the presence of fluid in the pipe. Indeed, the real consequence of a leakage is a sudden pressure decrease for all the components located downwards the faulty component and, at last, a lack of fluid in the circuit. As a result, a pipe must transmit the couple fluid/pressure in order to take into account and to correctly propagate the leakage information throughout the model. Moreover, as all the components (i.e. downwards but also upwards) have to be informed of such a failure, the fluid/pressure signal has to be bidirectional.

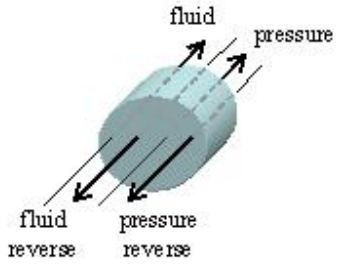


Figure 3 − Pipe section

In the component models, failure propagation will be modelled by assertions that constrain the values of flow variables. The following example shows in details the failure propagation in the pipe model.

Example:

```
node pipe
 flow
  output_pressure : {max,ok,low,no} : out;
  output_fluid : {yes,low,no} : out;
  output_pressure_reverse_info : {max,ok,low,no} : in;
  output_fluid_reverse_info : {yes,low,no} : in;
  input_pressure : {max,ok,low,no} : in;
  input_fluid : {yes,low,no} : in;
  input_pressure_reverse_info : {max,ok,low,no} : out;
  input_fluid_reverse_info : {yes,low,no} : out;
 state
  state_ : {ok,leakage};
 event
  leak;
 trans
  (state_ = ok) |- leak -> state_ := leakage;
```

```
assert
 output_fluid = (case {
 (state_ = ok) : input_fluid,
 (state_ = leakage) and ((input_fluid = yes) or (input_fluid = low)) : low,
 else no}),
 input_fluid_reverse_info = (case {
 (state_ = ok) : output_fluid_reverse_info,
 (state_ = leakage) and ((input_fluid = yes) or (input_fluid = low)) : low,
 else no}),
 output_pressure = (case {
 (state_ = ok) and not((input_fluid = no)) : input_pressure,
 else no}),
 input_pressure_reverse_info = (case {
 (state_ = ok) and not((input_fluid = no)) : output_pressure_reverse_info,
 else max});
init
 state_ := ok;
edon
```

When a pipe is not leaking, output pressures and output fluid levels are equal to input ones. When a leak occurs, the fluid level decreases from `yes` to `low` until the reservoir is empty (the input fluid level is `no`). Moreover, while the pipe is not empty (fluid different from `no`), the leak increases the upwards pressure and decreases the downwards one.

In Cecilia OCAS workshop, each node is associated to an icon and belongs to a library. Once the component library created, the system is easily and quickly modelled. Components are dragged and dropped from the library to the system architecture sheet and then linked graphically. The whole hydraulic system model is made of about 15 component classes.

Safety Assessment Techniques

Formal Safety Requirements: As stated in the case-study presentation section, the main safety requirement for the hydraulic system is: "*Total loss of hydraulic power is classified catastrophic*". We also considered two related requirements: "*Loss of two hydraulic channels is classified major*", "*Loss of one hydraulic channel is classified minor*". We associate with this set of safety requirements three qualitative requirements of the form "*if up to N individual failures occur then the loss of N+1 power channels of hydraulic system shall not occur*" with N = 0,1,2.

To model these qualitative requirements we first have to model the loss of N+1 power channels. Let N = 2, so we consider the total loss of hydraulic power. A first approach consists in using propositional formula 3_Hyd_Loss that would be true whenever the value of flow output_pressure of the distribution lines of the three hydraulic channels is equal to no**.** But this formula fails to adequately describe the failure condition. It could hold in evolutions of the system during a small period of time and then it would no longer hold as the hydraulic power is recovered due to appropriate activation of a backup such as the RAT for instance. The correct description of the failure condition should model the fact that hydraulic power is definitively lost. Hence we use Linear Temporal Logic (ref. 5) operators to model a failure condition. The following temporal formula models the permanent loss of hydraulic power:

```
Permanent_3_Hyd_Loss: F G 3_Hyd_Loss
```

where F is the eventually (or Finally) operator, G is the always (or Globally) operator. Formula Permanent_3_Hyd_Loss can be read "*eventually Hydraulic power is totally lost in all future time steps*". So the general form of qualitative requirements we check is:

```
No_N+1_S_Loss: G upto_N_failures -> ~ F G N+1_S_Loss
```

with N = 0,1,2 and `upto_N_failures` is a property that holds in all states of a system such that up to N individual failures have occurred.

Graphical Interactive Simulation: A Safety Engineer can check the effect of failure occurrences on the system architecture using Cecilia OCAS graphical interactive simulator. The system architecture is depicted by a set of interconnected boxes that represent nodes of the AltaRica model. Icons are associated with a node state. For

instance, a green box is displayed if a distribution line delivers power and a red box is displayed otherwise. These icons help to rapidly assess the component current state.

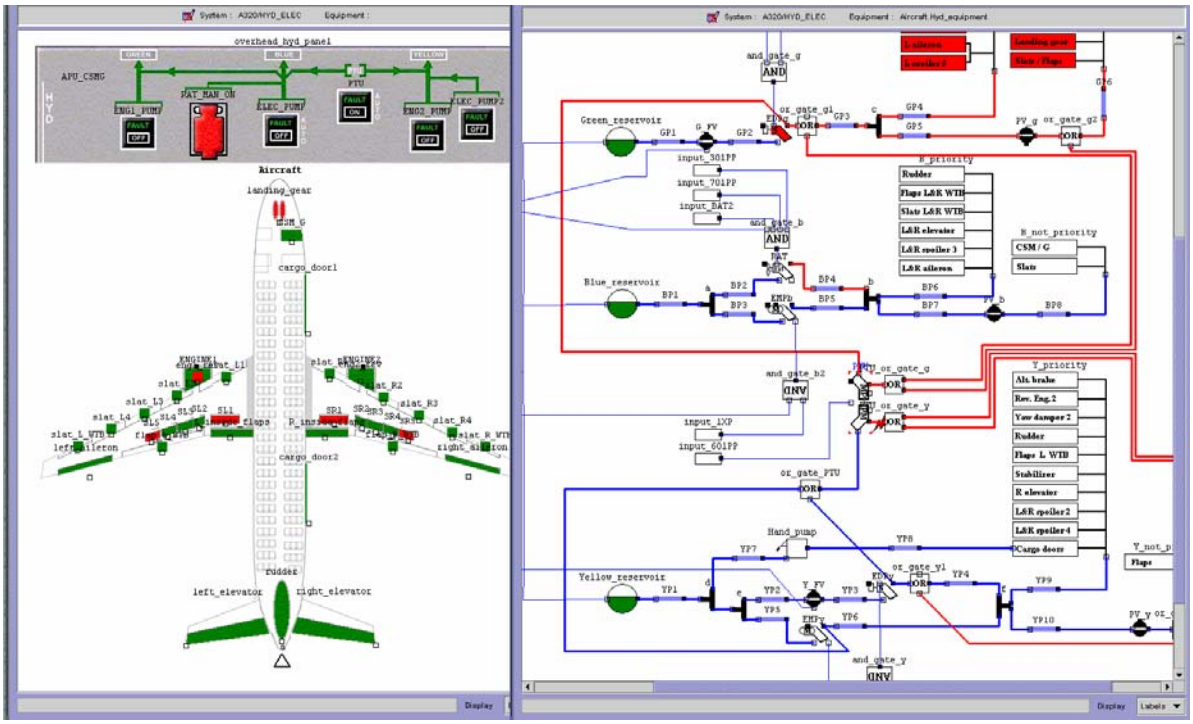


Figure 4 − Cecilia OCAS Graphical Simulator

To observe more complex situation such as the loss of several channels, special nodes called "observers" are added into the model. An observer internal state only depends on the value of other components outputs.
First, the simulator computes the initial state. Then, when the safety engineer selects a node the simulator proposes the set of events that can be performed at this step. This is the set of events with a guard that is true in the current state. The safety engineer chooses an event and the resulting state is computed by the simulator. As failures are events in the AltaRica model, the safety engineer can inject several failure events into the model in order to observe whether a failure condition is reached (such as loss of one or several power channels).

Figure 4 shows the graphical user interface of Cecilia OCAS. The Hydraulic system is displayed in the right window. All basic icons represent a component (tank, pump, distribution line …) of this system. The left window displays a set of observers that show whether aircraft devices powered by the hydraulic system are available or not. At the top of this window, we designed a control panel similar to the aircraft panel with button components that are used to activate or inactivate components in the hydraulic system.

Model-checking: A model-checker as Cadence Labs SMV (ref. 6) performs symbolically an exhaustive simulation of a finite-state model. The model-checker can test whether the qualitative requirements stated as temporal logic formulae are valid in any state of the model. Whenever a formula is not valid, the model-checker produces a counter-example that gives a sequence of states that lead to a violation of the safety requirement.

We developed tools to translate a model written in Altarica into a finite-state SMV model. Thus, we were able to check that both system models enforced their qualitative safety requirements. All requirements were verified in less than ten seconds although the truth value of some formulae depended in each state on as much as 100 boolean variables.

Applicability of the approach to robotic systems

Applicability scope of the approach with respect to traditional safety assessment process: The kinds of models and analysis presented before are devoted to a pivot step of the safety assessment process. Former analysis aim at identifying and classifying the failure conditions according to their criticality. The pivot step is used to demonstrate that a system architecture enable to meet safety requirements. Following steps deal with the verification of the hypothesis used to build the pivot models: are the considered failure modes the good ones? For quantitative part of the analysis, how representative are the considered failure rates? The verification process depends on the kind of system components. In the simplest case, it may consist in checking the compliance of the hypothesis with available data base. When considering software based component, it requires testing the software more or less heavily according to its criticality. Formal assessment techniques like the one presented before were primarily defined to perform such software verification. Nevertheless their application is today more or less successful according to the software complexity. The interested reader may consult for instance ref. 7 and ref 8 for further details.

The case of robotic systems: In the following, we focus on the applicability of the pivot step previously detailed to robotic systems. According to our understanding, such systems consist in a physical devices (sensors, actuators, …) monitored and controlled by embedded software that are structured in a more or less sophisticated control architecture.
We insisted in section 2 on building system models at the right granularity level with respect to the analysis purpose. The selected granularity level identifies the main functions provided by basic component and highlights how the quality of the function outputs depend on the quality of the function inputs, on the current (faulty or nominal) function modes and on protections inserted in the component. Such generic principles can be applied to model and assess simplest robot control architectures against safety requirements.
In most sophisticated architectures where plans are computed on board, the applicability is not straightforward but seems still possible. In such architecture, a part of the policy used to control the robot may be implicitly defined by a plan generation module in order to cope with a numerous number of procedures. In this context, it is useless to enter into the details of the planning algorithms since one is interested only in finding dependencies between components that propagate failure. Nevertheless, these dependencies can be numerous depending on the combination of use of basic components generated by the planner. We already met a similar case when studying a highly reconfigurable aircraft electrical system (ref. 9). In such a case, one can left open the control of the devices piloted by plan. The analysis enables to find out bad plans (with our without combination of failures) that lead to critical situations. The proposal is to derive new safety requirements to avoid the generation of such plans. Then, the analysis can be conducted under these new requirements.

Conclusion and Future Work

Our experiment about traditional embedded safety critical systems shows that safety system modelling and analysis are possible and fruitful using a formal approach provided that models have the right level of detail. We have to observe that our models should not confine to failure propagation related with the functional analysis. They should also include failure propagations that could be related to system-level risks as specification errors, assumptions, synergistic considerations through-out the life-cycle, energy effects, …Previous works such as references 10 and 11 present modelling approaches that, as ours, abstract nominal physical details and focus on failure propagation. However, the author main goal was to generate fault trees, so their models focus on system architecture and do not enable temporal analysis of highly dynamic reconfiguration mechanisms. Following our approach, we could state formally interesting qualitative and temporal safety requirements of aircraft systems and perform assessment analysis with interactive simulation and model-checking tools without performance problems.

Nevertheless, as discussed in section 4, it is worth noting that this fruitful approach assists only one specific step of the safety assessment process. Moreover, if our paper gives a flavour of the use of model-checking techniques for a specific purpose, it does not pretend to give a comprehensive view of the available formal techniques and their uses. For instance, model-checking techniques are also used to generate plan in the robotic field (e.g. ref 12) or to validate the model for some model based planners. Future works at ONERA intends to build such a more comprehensive approach of the safety analysis for autonomous systems and more specifically unmanned vehicle like. Such a step is mandatory if one wants to use more widely drone submitted to stringent regulations.

Acknowledgement

The experiment described in this paper has been developed within the ISAAC Project, a European sponsored project, FP6-2002-Aero-1-501848.
Thank you to CNES, LAAS and ONERA colleagues of AGATA project for fruitful discussion.

References

1. O. Akerlund, P. Bieber, E. Boede, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, J. Gauthier, O. Lisagor, A. Lüdtke, S. Metge, C. Papadopoulos, T. Peikenkamp, L. Sagaspe, C. Seguin, H. Trivedi and L. Valacca, ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects, proceedings Embedded Real Time Software 2006, Toulouse (France), Janvier 2006.

2. A. Arnold, A. Griffault, G. Point, A. Rauzy. The AltaRica formalism for describing concurrent systems. Fundamenta Informaticae n°40, p109-124, 2000.

3. Manuel utilisateur OCAS V3.2, Dassault Aviation, 2005.

4. A. Rauzy. Mode automata and their compilation into fault trees. Reliability Engineering and System Safety, 2002.

5. Z. Manna, A. Pnuelli. The temporal logic of reactive and concurrent systems. Springer − Verlag, New-York, 1992, ISBN 0-387-97664-7.

6. K.L. MacMILLAN. Symbolic Model Checking. Kluwer Academic Publishers, 1993, ISBN 0-7923-9380-5.

7. Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, David Monniaux, Laurent Mauborgne, Xavier Rival.
The ASTRÉE Analyzer. ESOP 2005: The European Symposium on Programming, Edinburgh, Scotland, April 2—10, 2005. Lecture Notes in Computer Science 3444, © Springer, Berlin, pp. 21—30.

8. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Software Model Checker Blast: Applications to Software Engineering. *Int. Journal on Software Tools for Technology Transfer*, Invited to special issue of selected papers from FASE 2005.

9. Ch. Kehren, *et al*, Advanced Simulation Capabilities for Multi-Systems with AltaRica, in Proceedings of the 22nd International System Safety Conference (ISSC), 2004, System Safety Society.

10. Y. Papadopoulos, M. Maruhn. Model-based automated synthesis of fault trees from Matlab-Simulink models. DSN2001, Int. Conf. on Dependable Systems and Networks (former FTCS), Gothenburg, Sweden, pages 77-82, ISBN 0-7695-1101-5, July 2001.

11. Fenelon, P., McDermid, J.A., Nicholson, M. & Pumfrey, D.J. Towards Integrated Safety Analysis and Design, ACM Computing Reviews, Vol. 2, No. 1, p. 21-32, 1994.

12. A. Cimatti, M. Pistore, M. Roveri, P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking , *Artificial Intelligence* , 147 (1-2):35-84, 2003. Elsevier Science publisher

Z and ProCoSA based specification of a distributed FDIR in a satellite formation

Ch. Castel**, J.-Ch. Chaudemar*, J.-F. Gabard**, C. Tessier**
***ONERA-DCSD, *SUPAERO*

Abstract

On-board FDIR (Fault Detection, Isolation and Recovery) is contemplated for autonomous satellite formations. Several FDIR strategies have been specified using the Petri net - based software ProCoSA (for the dynamic aspects) on the one hand, and the set theory - based Z specification language (for the static aspects) on the other hand. ProCoSA enables to specify the different state changes triggered by the different events within the formation; Z enables to describe the relations and constraints (invariants) between the state variables.
The paper focuses on a global specification including both the dynamic and static aspects, through a formal link between ProCoSA and Z. The link is implemented and allows some properties of the strategies to be checked.

1 Introduction

The autonomous formation flying of multiple spacecraft to replace a single large satellite will be an enabling technology for a number of future missions. Potential applications include synthetic apertures for surveillance and high-resolution interferometry missions, or for taking widespread field measurements for atmospheric survey missions [Cra].Very precise autonomous coordination and control differentiate formations from constellations. The challenge is to develop both the software and the hardware to allow separate, unconnected spacecraft to function as if they were a single, solid structure [Nas]. Spacecraft within a formation may be different from one another and the different parts of one instrument may be distributed among several spacecraft.
FDIR (Fault Detection, Isolation and Recovery) is the means to detect off-nominal conditions, isolate the problem to a specific subsystem/component, and recover of vehicle systems and capabilities [NAS05]. Formation flying brings a new concept in FDIR, i.e. the *formation* has to be considered as an entity in itself. Indeed the scientific mission is performed by the formation (and not by the individual spacecraft). Therefore specific FDIR strategies [CGL$^+$06] have to be considered in order to deal with formation specific failures e.g. instrument failure, problems with the formation geometry, inter-spacecraft communication failures.

In this paper, FDIR is considered as an operational function that contributes to the autonomy of the system and whose main purpose is to maintain the availability of each satellite of the formation for the mission. A global specification of FDIR concepts including both the dynamic and static aspects, through a formal link between ProCoSA and Z, is presented. The link is implemented and allows some properties of the strategies to be checked. As an example, a typical anomaly that may affect the formation integrity, namely a violation of the "Keep Out Zone"[1] is considered with a centralised centralised strategy.
The paper is organised as follows : after a short presentation of the Z notation, the next section describes FDIR concepts for a satellite formation and their implementation for a unique anomaly case by using Z specification language. Then the centralised strategy we have designed is executed with a ProCoSA simulation refined by Z data and constraint definitions. We explain the linking between Z schemas and ProCoSA Petri nets for our case study. An analysis section presents the interest to associate Z-modelling and ProCoSA simulation for a safety-critical system like a satellite formation called Simbol-X.

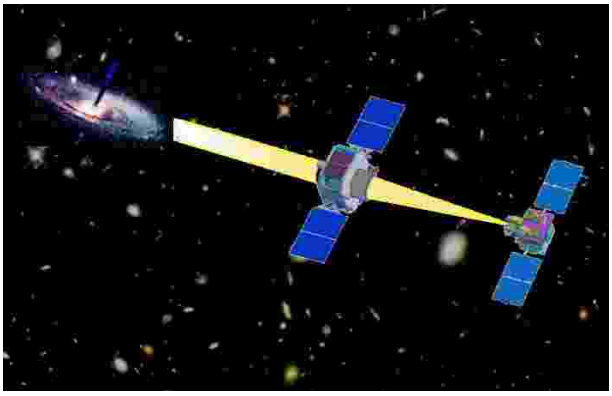


Figure 1: Simbol-X mission (http://apc-p7.org/APC_CS/Experiences/SIMBOLX/index.phtml)

2 Z-modelling of FDIR within a satellite formation

Z [Spi00, But01] is a formal specification language based on the set theory and the predicate logic. The Z specification of a system consists of state variables, an initialisation and a set of operations on state variables. *Invariants*, which represent constraints which must always be satisfied, are associated with state variables. The basic element of Z specification is the *schema*.

[1] Keep Out Zone (KOZ) is defined as a safety sphere around each satellite that another satellite must not enter.

2.1 Z-modelling of FDIR concepts

The first step is to model the main concepts that take part into the formation FDIR. This is a static view of FDIR agents with their main constraints.

FDIR Regardless of the components implementing it, FDIR is considered as a kind of operational function *FONCTION_OP*, including non-empty finite sets of functions for detection (*detections*) and recovery (*reconfigurations*), and dealing with a fixed non-empty finite set of defaults (*traite*).

FDIR

$$
\begin{array}{l}
FONCTION\_OP \\
detections : \mathbb{F}_1\ DETECTION \\
traite : \mathbb{F}_1\ DEFAUT \\
reconfigurations : \mathbb{F}_1\ RECONFIGURATION
\end{array}
$$

SIMBOLX We focus on a particular system called Simbol-X [Sim07], composed of two satellites, *sat1, sat2* within a *formation*, and a non-empty set of ground stations, *stationsol*. This system is characterised by a set of operational functions, *fct_ops* in relation to each other (*participe_a*).
The predicate part states that the formation is composed of one detector satellite and one mirror satellite whose KOZ radius are specified, and the different operational functions are clearly distinct if they relate to either the formation or the detector satellite or the mirror satellite.

SIMBOLX

$$
\begin{array}{l}
sat1, sat2 : SATELLITE \\
formation : FORMATION \\
stationsol : \mathbb{F}_1\ SOL \\
fct\_ops : \mathbb{F}_1\ FONCTION\_OP \\
participe\_a : FONCTION\_OP \leftrightarrow FONCTION\_OP \\
\hline
formation.satellites = \{sat1, sat2\} \\
sat1.type = detecteur \land sat1.koz = 18 \\
sat2.type = miroir \land sat2.koz = 15 \\
\langle formation.dispose\_de\_fop, sat1.dispose\_de\_fop, \\
sat2.dispose\_de\_fop\rangle \ \mathsf{partition}\ fct\_ops \\
\mathrm{dom}\ participe\_a = fct\_ops \\
\mathrm{ran}\ participe\_a = fct\_ops \\
\forall fct : FONCTION\_OP \mid fct \in fct\_ops \bullet \{(fct \mapsto fct)\} \cap participe\_a = \emptyset
\end{array}
$$

SATELLITE A satellite is a kind of FDIR actor (*ACTEUR*) characterised by a *type* (detector or mirror), a *koz* radius and some operational functions (*dispose_de_fop*)

such as a FDIR function (*dispose_de_fdir*), an intersatellite communication function (*dispose_de_fcom_sat*), and a communication function with a ground station (*dispose_de_fcom_sol*).
The single constraint relates to a minimum value for the KOZ radius.

SATELLITE

$$
\begin{array}{l}
ACTEUR \\
type : ROLE \\
koz : \mathbb{N}_1 \\
chainefonctionnelle : \mathbb{F}_1\ CHAINE\_FCT \\
dispose\_de\_fop : \mathbb{F}_1\ FONCTION\_OP \\
dispose\_de\_fdir : FDIR \\
dispose\_de\_fcom\_sat : FCT\_COM\_SAT \\
dispose\_de\_fcom\_sol : FCT\_COM\_SOL \\
\hline
koz \geq 5
\end{array}
$$

SOL A ground station is a kind of actor (*ACTEUR*) characterised by some operational functions (*dispose_de_fop*) and a communication function with a satellite (*dispose_de_fcom_sat*).

SOL

$$
\begin{array}{l}
ACTEUR \\
dispose\_de\_fop : \mathbb{F}_1\ FONCTION\_OP \\
dispose\_de\_fcom\_sat : FCT\_COM\_SAT
\end{array}
$$

FORMATION The formation is a kind of FDIR actor (*ACTEUR*), composed of a non-empty finite set of satellites (*satellites*) characterised by a distance (*distance*), a status related to the KOZ, *status_vkoz* which can take two values that are *normal* for a normal status and *en_panne* for a failure status, and some operational functions (*dispose_de_fop*) such as a FDIR function (*dispose_de_fdir*).
The predicate part states properties concerning distance between both satellites for a normal status.

FORMATION

$$
\begin{array}{l}
ACTEUR \\
satellites : \mathbb{F}_1\, SATELLITE \\
distance : SATELLITE \times SATELLITE \pfun \mathbb{N}_1 \\
dispose\_de\_fop : \mathbb{F}_1\, FONCTION\_OP \\
dispose\_de\_fdir : FDIR \\
status\_vkoz : ETAT\_V\_KOZ \\
\hline
\mathrm{dom}\, distance \subseteq satellites \times satellites \\
status\_vkoz = normal \Leftrightarrow \\
(\forall\, sat1, sat2 : SATELLITE \mid (sat1, sat2) \in \mathrm{dom}\, distance \\
\bullet\ distance(sat1, sat2) > sat1.koz + sat2.koz)
\end{array}
$$

As far as FDIR itself is concerned, several strategies have been defined [CGL+06] and have been implemented with ProCoSA. Only a centralised strategy is dealt with in this paper.

OPERATION A recovery operation is characterised by its type (*nature*), i.e. reset, redundancy switch or movement operations.

OPERATION

$$
\begin{array}{l}
nature : TYPE\_OPERATION
\end{array}
$$

STRATEGIE_FDIR An FDIR strategy has a type, *type_strat*, i.e. centralised, mixed or distributed. It concerns the relationship between a recovery operation and an operation function, connects FDIR function to FDIR cases, implies FDIR functions, intersatellite communications and communications with the ground, and a non-empty set of defaults.
The invariant relationships stipulate that these functions are well associated to the concerned satellites.

STRATEGIE_FDIR

$$
\begin{array}{l}
type\_strat : TYPE\_STRATEGIE\_FDIR \\
reconfig : OPERATION \leftrightarrow RECONFIGURATION \\
satisfait : FDIR \leftrightarrow FDIR\_CAS \\
fdir\_sat : SATELLITE \leftrightarrow FDIR \\
com\_sat : SATELLITE \leftrightarrow FCT\_COM\_SAT \\
com\_satsol : SATELLITE \leftrightarrow FCT\_COM\_SOL \\
defauts : \mathbb{F}_1\, DEFAUT \\
\hline
\forall sat : SATELLITE;\ fd : FDIR \bullet (sat, fd) \in fdir\_sat \\
\Rightarrow fd = sat.dispose\_de\_fdir \\
\forall sat : SATELLITE;\ fc : FCT\_COM\_SAT \bullet (sat, fc) \in com\_sat \\
\Rightarrow fc = sat.dispose\_de\_fcom\_sat \\
\forall sat : SATELLITE;\ fc\_sol : FCT\_COM\_SOL \bullet (sat, fc\_sol) \in com\_satsol \\
\Rightarrow fc\_sol = sat.dispose\_de\_fcom\_sol
\end{array}
$$

S_CENTR_1 This is one of the centralised FDIR strategies that we have designed for satellite formations. This strategy is such as each satellite carries its own detection function and only the detector satellite *sat1* carries the recovery capabilities. Moreover, only *sat1* communicates with the ground for FDIR.

S_CENTR_1

$$
\begin{array}{l}
STRATEGIE\_FDIR \\
SIMBOLX \\
\hline
type\_strat = centralisee \\
defauts \subseteq sat1.dispose\_de\_fdir.traite \cup sat2.dispose\_de\_fdir.traite \\
fdir\_sat = \{(sat1, sat1.dispose\_de\_fdir), (sat2, sat2.dispose\_de\_fdir)\} \\
sat1.dispose\_de\_fdir.detections \neq \emptyset \\
sat2.dispose\_de\_fdir.detections \neq \emptyset \\
sat1.dispose\_de\_fdir.reconfigurations \neq \emptyset \\
sat1.dispose\_de\_fdir.reconfigurations \subseteq \operatorname{ran} reconfig \\
sat2.dispose\_de\_fdir.reconfigurations \cap \operatorname{ran} reconfig = \emptyset \\
com\_sat = \{(sat1, sat1.dispose\_de\_fcom\_sat), (sat2, sat2.dispose\_de\_fcom\_sat)\} \\
com\_satsol = \{(sat1, sat1.dispose\_de\_fcom\_sol)\}
\end{array}
$$

2.2 State evolution for a KOZ violation case

The initial values of the Simbol-X formation are given by the following Z schema:

Init_FORMATION

$FORMATION$
$\Xi SIMBOLX$

$status\_vkoz = normal$
$sat1.type = detecteur \land sat1.koz = 18$
$sat2.type = miroir \land sat2.koz = 15$

Let us consider the operations due to a formation fault such as a KOZ violation. This fault (*ev_fdir?*) is detected when both satellites get closer.

PANNE_V_KOZ

$\Xi SIMBOLX$
$\Delta FORMATION$
$ev\_fdir? : DEFAUT$

$ev\_fdir? \in formation.dispose\_de\_fdir.traite$
$(sat1, sat2) \in \mathrm{dom}\, distance$
$distance(sat1, sat2) > sat1.koz + sat2.koz$
$distance'(sat1, sat2) \leq sat1.koz + sat2.koz$

For the detector satellite and for the mirror satellite, the fault is respectively expressed by *anom_S1?* and *anom_S2?* and detected by their detection functions:

NORMAL2PANNE

$\Xi SIMBOLX$
$\Delta FDIR$
$anom\_S1? : DEFAUT$
$anom\_S2? : DEFAUT$

$anom\_S1? \in sat1.dispose\_de\_fdir.traite$
$anom\_S2? \in sat2.dispose\_de\_fdir.traite$
$sat1.dispose\_de\_fdir.detections \cap detections'$
$\neq detections \cap sat1.dispose\_de\_fdir.detections$
$sat2.dispose\_de\_fdir.detections \cap detections'$
$\neq detections \cap sat2.dispose\_de\_fdir.detections$

To express intersatellite communication, we define a non-exhaustive enumerated type of messages:

$MESSAGE ::= messok \mid messperteISL \mid messperteRF \mid messalarme$
$\mid messreconf \mid messmanoeuvre \mid messsol \mid messcritique$

The mirror satellite sends an alarm message to the detector satellite in order for it to deal with this default and develop an operational strategy.

$INFO2S1 \mathrel{\widehat=} [mess! : MESSAGE \mid mess! = messalarme]$

The KOZ violation requires a quick reaction of the formation. A high priority recovery strategy is planned by the FDIR satellite, i.e. the detector satellite, for the whole formation.

RECONF_F

$\Xi SIMBOLX$
$\Delta STRATEGIE\_FDIR$

$reconfig = \emptyset$
$reconfig' \neq \emptyset$

A message stating the appropriate manoeuvre is sent by the detector to the mirror satellite.

$$RECONF\_S2 \mathrel{\widehat=} [mess! : MESSAGE \mid mess! = messreconf \land mess! = messmanoeuvre]$$

The manoeuvre is finished when a normal state is recovered.

FIN_MANOEUVRE

$\Xi SIMBOLX$
$\Delta FORMATION$

$status\_vkoz = en\_panne$
$status\_vkoz' = normal$

Therefore the KOZ violation processing is: first detection, then a manoeuvre executed by both the satellites.

$$ViolationKOZ \mathrel{\widehat=} NORMAL2PANNE \fatsemi INFO2S1 \fatsemi RECONF\_F \fatsemi RECONF\_S2 \fatsemi FIN\_MANOEUVRE$$

3 ProCoSA simulation of KOZ violation

3.1 Petri nets

The simulation of KOZ violation with ProCoSA (figure 2) distinguishes three behaviours corresponding to the detector satellite state (*etat_S1*), the mirror satellite state (*etat_S2*), and the FDIR satellite (*FDIR_S1*), namely the detector satellite.
A KOZ violation fault affects both satellites and makes the *state-S_i* (*etat_Si*) Petri nets pass from the nominal (*normal*) to the fault state (*en_panne*) whereas *FDIR_S1* net passes from the nominal (*nominal_formation*) to the detection state (*D*). For *etat_S2*, an additional fault state is introduced to take into account the sending of an alarm message to the FDIR satellite if the intersatellite communication link is available (*COM_OK_S2_vers_S1* [2]). Then *FDIR_S1* passes to state *reactif_S1*

[2] COM_OK_S2_vers_S1 and COM_OK_S1_vers_S2 are two global places used in other Petri nets modelling the intersatellite communication state. In this paper, we don't focus on this aspect.

meaning that a security reaction is performed for this type of fault whereas *etat_S1* passes to state *reconf_en_cours*. Indeed, as *FDIR_S1* carries all the FDIR knowledge and algorithms, only *FDIR_S1* can perform a reaction, even if the fault is detected on another spacecraft. *FDIR_S1* then passes to state *att_reconf_S2* if the intersatellite link is available (*COM_OK_S1_vers_S2*) meaning that actions necessary to recovery are expected from the mirror satellite. Thus each *state-S_i* net passes to state *on-going reconfiguration* (*reconf_en_cours*), before going back to the nominal state. At last *FDIR_S1* goes back to the nominal state.

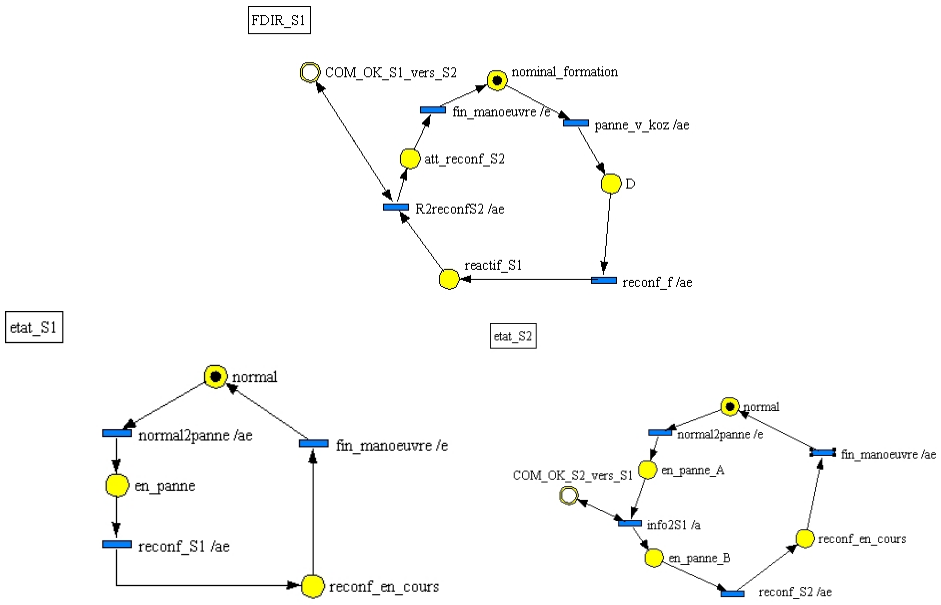


Figure 2: KOZ violation recovery simulated with ProCoSA

3.2 Linking Z schemas and ProCoSA Petri nets

As pointed out in [HH99, Xud01, PJ03], Z schemas are well suited to define data structures, system constraints and functional processing whereas Petri nets are a graph-based formal model for representing the control structures and dynamic behaviours of concurrent and distributed systems that cannot be explicitly described in Z. Accordingly a relationship between Z and Petri nets offers a coherent formalism of specification for designing reliable systems.

In this section, we present how to combine Z schemas and ProCoSA Petri nets for the specification of FDIR in a satellite formation. The first step consists in building a relevant Z model of a satellite formation. By analysing the requirement description of the system, we identify its main components (*FORMATION, SATELLITE, SOL*) and functionalities (*FDIR, OPERATION, STRATEGIE_FDIR*)

which define the overall system structure. Then, the definition of Z-operations (*PANNE_V_KOZ, NORMAL2PANNE, INFO2S1, RECONF_F, RECONF_S2, FIN_MANOEUVRE*) refines the model for the specific case of the KOZ violation. Furthermore, ProCoSA simulation enables to link Z-operations to Petri net transitions: one Z operation defines one Petri net transition. In the predicate part of a Z operation schema, the pre-condition part which is expressed through non-dashed variables is the guard specifying the enabling condition of the corresponding transition, whereas the post-condition part expressed through dashed variables defines its firing result. Moreover, ProCoSA events and messages can be associated with transitions and may respectively correspond to input and output variables in Z operations. According to the rule stated above, the Petri net transitions *panne_v_koz, normal2panne, info2S1, reconf_f, reconf_S2, fin_manoeuvre* correspond to operation schemas described in the Z model. The transition *reconf_S1* is added to mean that formation recovery strategy *reconf_f* is composed of an internal S1 recovery function. This recovery function is partially hinted in the *reconfig* variable of the Z-operation called *RECONF_F*. In fact, this transition is similar to *reconf_f*, so it corresponds to the same Z-operation *RECONF_F*.

In our model, there is only an implicit relationship between local variables in Z operation schemas and input or output places of a transition in ProCoSA Petri nets. Some expressions relate to the system state before and after transition firing, like the expression concerning detection functions in the predicate part of *NORMAL2PANNE* Z operation schema.

Furthermore, the initial marking of the ProCoSA Petri nets is consistent with the initial state schema in Z (*Init_FORMATION*).

For a complete simulation, a new ProCoSA procedure will be developed to simulate the intersatellite distance variation and the Z property concerning the distance and KOZ.

4 Analysis of Z-ProCoSA relationship

For consistency reasons, the two processes, i.e. the Z specification and the ProCoSA simulation, were jointly carried out in order to fully benefit from the advantages of each method. Z provides the formal aspect for the specification of the system, whereas ProCoSA allows to focus on dynamic aspects and the sequences of the state variations. This has enabled a better understanding of the formation behaviour faced with a KOZ violation anomaly by revealing not very precise requirements, e.g. which satellite first operates the collision avoidance manoeuvre.

Thanks to data and constraint definitions, the Z specification has allowed to modify an existing ProCoSA simulation of an FDIR centralised strategy that only took into

account the state evolutions.
Conversely, the ProCoSA simulation also contributes to develop the Z model by describing state changes and control flows between the various Petri nets, i.e. the behavioural aspect. The causal relation or state sequence is represented graphically: normal state, then fault detection, at last recovery with manoeuvre.

Compared to other models [HH99, Xud01, PJ03], the complexity of the approach seems lower and the Z-ProCoSA relationship analysis weakly relates to net places and their corresponding Z schemas and confidently relies on ProCoSA property analysis tool (place safety, detection of dead markings).

5 Conclusion

ProCoSA is suitable to take into account the behaviour of concurrent and distributed systems like FDIR for a satellite formation, whereas Z is well known for data abstraction and functional specification. The idea is that the combination of both formalisms leads to very reliable models.
The next steps in our work are the following:

- implement a hybrid simulation with discrete and continuous state variables, e.g. to simulate the intersatellite distance variation and the Z properties concerning the distance and KOZ ;
- refine the simulation by taking time into account, i.e. state duration, delay between satellite operations and concurrence between both satellites ; thus, we will test other FDIR strategies that need more time for converging or involve ground stations.
- define a formal methodology applying proof checking that is based on combined Z and Petri net specifications.

Appendix: ProCoSA

A Petri net $< P, T, F, B >$ is a bipartite graph with two types of nodes: P is a finite set of places; T is a finite set of transitions [DA05]. Arcs are directed and represent the forward incidence function $F : P \times T \rightarrow \mathbb{N}$ and the backward incidence function $B : P \times T \rightarrow \mathbb{N}$ respectively. The marking of a Petri net is defined as function $\mathcal{M} : \mathcal{P} \rightarrow \mathbb{N}$: tokens are associated with places. The evolution of tokens within the net follows transition firing rules. Petri nets allow sequencing, parallelism and synchronization to be easily represented. An *interpreted Petri net* is such that conditions and events are associated with transitions.

ProCoSA [BGVBT06] is a software environment meant for controlling and monitoring highly autonomous systems. System autonomy is usually obtained by putting

together various functions, among which: data analysis (sensor data, monitoring data, operator's inputs), nominal mission monitoring and control (vehicle and payload control actions), decision (management of disruptive events, replanning). These functions, which are often developed as separate subsystems, have to co-operate in order to fulfil the autonomous system behaviour requirements for the specified missions. More precisely, the needs are the following:

- off-line tasks: specification of the co-operation procedures between subsystem software; subsystem coding for embedded operation;
- on-line tasks: procedure monitoring, event monitoring, and management of the dialog with the operator.

ProCoSA includes the following components:

- EdiPet, a graphical interface for Petri nets which is used both by the developer for procedure design and by the operator for execution monitoring;
- JdP, the Petri net player, that executes the procedures, fires the event-triggered transitions of the Petri nets and synchronises the activation of the associated sub-system functions; a socket-based communication protocol allows data to be exchanged with external subsystem software;
- Tiny, a Lisp interpreter dedicated to distributed embedded applications.

The Petri nets used by ProCoSA are interpreted Petri nets: triggering events such as activation or event generation requests are attached to the transitions. Timers can be programmed: a special activation request enables a timer variable to be instantiated, which allows actions with a limited duration to be modelled.
The ProCoSA procedures are used to model the desired behaviours of the autonomous system; the hierarchical modelling features offered by ProCoSA enable to structure the whole application in a generic way: at the highest description level, generic behaviours can be described, regardless of the characteristics of a given vehicle; at the lowest level, they specify the sequences of elementary actions to be performed by the vehicle or the payloads; this modular approach enables a quick adaptation to system changes (e.g. taking into account a new payload).
An important feature of ProCoSA lies in the fact that there is no code translation step between the Petri net procedures and their execution: they are directly interpreted by the Petri net player, thus avoiding any supplementary error causes.
ProCoSA finally includes a verification tool, which makes use of the Petri net analysis techniques to check that some "good" properties are satisfied by the procedures, both at the single procedure level and at the whole project level (that is to say taking into account inter-net connections); the following properties are checked: place safety (not more than one token per Petri net place), detection of dead markings (deadlocks), detection of cyclic firing sequences (loops).

References

[BGVBT06] M. Barbier, J.-F. Gabard, D. Vizcaino, and O. Bonnet-Torrès. ProCoSA: a software package for autonomous system supervision. In *CAR'06 - First Workshop on Control Architectures of Robots*. Montpellier, FR, 2006.

[But01] M. Butler. *Introductory notes on specification with Z.* University of Southampton, GB, 2001. http://www.ecs.soton.ac.uk/m̃jb/EL208/znotes.pdf.

[CGL+06] Ch. Castel, J.-F. Gabard, B. Laborde, R. Soumagne, and C. Tessier. FDIR strategies for autonomous satellite formations - A preliminary report. In *AAAI 2006 Fall Symposium "Spacecraft Autonomy: Using AI to Expand Human Space Exploration"*. Washington DC, USA, October 2006.

[Cra] Univ. Cranfield. Satellite formation flying for an interferometry mission . http://www.cranfield.ac.uk/soe/space/flying .htm.

[DA05] R. David and H. Alla. *Discrete, continuous, and hybrid Petri nets*, 2005.

[HH99] M. Heiner and M. Heisel. Modeling safety-critical systems with z and petri nets. In *Proceedings of the 18 th International Conference on Computer Computer Safety, Reliability and Security*, volume 1698 of *Lecture Notes In Computer Science*, pages 361 – 374. Springer, 1999.

[Nas] Nasa. Technology - formation flying . http://planetquest.jpl.nasa.gov/technology/formation_flying.cfm.

[NAS05] NASA. Glossary - NASA Crew Exploration Vehicle, SOL NNT05AA01J, Attachment J-6. http://www.spaceref.com/news/viewsr.html?pid=15201, 2005.

[PJ03] Fr. Peschanski and D. Julien. When concurrent control meets functional requirements, or z +. In *Third International Conference of B and Z Users*, volume 2651 of *Lecture Notes In Computer Science*, pages 79–97. Springer, 2003.

[Sim07] Simbol-X - Le vol en formation pour un tlescope X de nouvelle gnration. http://www.cnes.fr/web/5848-simbol-x.php, 2007.

[Spi00] M. Spivey. *The Fuzz manual*. Oxford, GB, 2000. `http://spivey.oriel.ox.ac.uk/mike/fuzz/fuzzman.pdf`.

[Xud01] H. Xudong. PZ nets – a formal method integrating Petri nets with Z. In *Information and Software Technology*, volume 43 Issue 1, pages 1 – 18. Elsevier Science, 2001.

Incremental Component-Based Construction and Verification of a Robotic System

Ananda Basu[†], Matthieu Gallien[*], Charles Lesire[*], Thanh-Hung Nguyen[†], Saddek Bensalem[†], Félix Ingrand[*] and Joseph Sifakis[†]

[*] LAAS/CNRS, Unversity of Toulouse
Toulouse, France
Email: firsname.lastname@laas.fr

[†]VERIMAG CNRS/University Joseph Fourier
Grenoble, France
Email: firstname.lastname@verimag.fr

***Abstract*— Autonomous robots are complex systems that require the interaction/cooperation of numerous heterogeneous software components. Nowadays, robots are critical systems and must meet safety properties including in particular temporal and real-time constraints. We present a methodology for modeling and analyzing a robotic system using the BIP component framework integrated with an existing framework and architecture, the LAAS[1] based on GenoM. The BIP componentization approach has been successfully used in other domains. In this study, we show how it can be seamlessly integrated in the preexisting methodology. We present the componentization of the functional level of a robot, the synthesis of an execution controller as well as validation techniques for checking essential "safety" properties.**

I. Introduction

A central idea in systems engineering is that complex systems are built by assembling components (building blocks). Components are systems characterized by an abstraction that is adequate for composition and re-use. It is possible to obtain large components by composing simpler ones. Component-based design confers many advantages such as reuse of solutions, modular analysis and validation, reconfigurability, controllability, etc.

Autonomous robots are complex systems that require the interaction/cooperation of numerous heterogeneous software components. They are critical systems as they must meet safety properties including in particular, temporal and real-time constraints.

Component-based design relies on the separation between coordination and computation. Systems are built from units processing sequential code insulated from concurrent execution issues. The isolation of coordination mechanisms allows a global treatment and analysis.

One of the main limitations of the current state-of-the-art is the lack of a unified paradigm for describing and analyzing the information flow between components. Such a paradigm would allow system designers and implementers to formulate their solutions in terms of tangible, well-founded and organized concepts instead of using dispersed coordination mechanisms such as semaphores, monitors, message passing, remote call, protocols, etc. It would allow in particular, a comparison of otherwise unrelated architectural solutions and could be a basis for evaluating them and deriving implementations in terms of specific coordination mechanisms.

The designers of complex systems such as autonomous robots need scalable analysis techniques to guaranteeing essential properties such as the one mentioned above. To cope with complexity, these techniques are applied to component-based descriptions of the system. Global properties are enforced by construction or can be inferred from component properties. Furthermore, componentized descriptions provide a basis for reconfiguration and evolutivity.

We present an incremental componentization methodology and technique for an already existing autonomous robot software developed at LAAS. The methodology considers that the global system architecture can be obtained as the hierarchical composition of larger components from a small set of classes of *atomic* components. Atomic components are units processing sequential code that offer interactions through their interface. The technique is based on the use of the *Behavior-Interaction-Priority* (BIP) [2] component framework which encompasses incremental composition of heterogeneous real-time components.

The main contributions of the paper include:

- A methodology for componentizing and architecting autonomous robot systems.
- Composition techniques for organizing and enforcing complex event-based interaction using the BIP framework.
- Validation techniques for checking essential properties, including scalable compositional techniques relying on the analysis of the interactions between components.

The paper is structured as follows. In Section II we illustrate with a real example, the preexisting architecture (based on GenoM [6]) of an autonomous robotic software developed at LAAS. From this architecture, we identify the atomic components used for the componentization of the robot software in BIP. Section III provides a succinct description of the BIP

[1]LAAS Architecture for Autonomous System.

component framework. Section IV presents a methodology for building the BIP model of existing GenoM functional modules and their integration with the rest of the software. Controller synthesis results as well as "safety" properties analysis are also presented. Section V concludes the paper with a state of the art, an analysis of the current results and future work directions.

II. Modular Architecture for Autonomous Systems

Autonomous robots are complex systems that require the interaction/cooperation of numerous and quite different software modules/components. This is usually achieved with the proper architecture, methods and tools.

A. Preexisting Software Architecture

At LAAS, we have developed a framework, a global architecture, that enables the integration of processes with different temporal properties and different representations. This architecture decomposes the robot system into three main levels, having different temporal constraints and manipulating different data representations [1]. This architecture is used on a number of robot, in particular DALA, an iRobot ATRV, and is shown on Fig. 1. The levels in this architecture are :

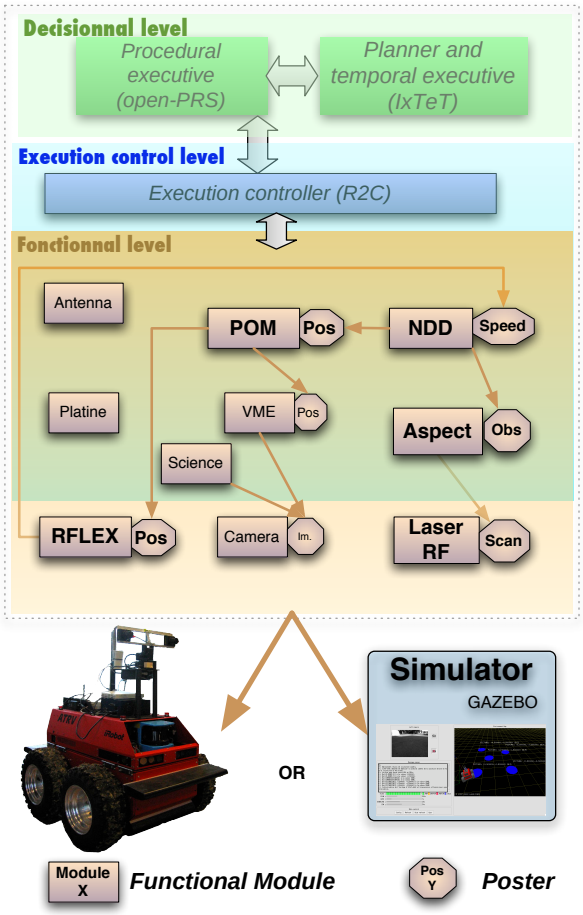


Fig. 1. An instance of the LAAS architecture for the DALA Robot.

- a *functional* level: it includes all the basic built-in robot action and perception capacities. These processing functions and control loops (e.g., image processing, obstacle avoidance, motion control, etc.) are encapsulated into controllable communicating modules developed using GenoM[2]. Each modules provide *services* which can be activated by the decisional level according to the current tasks, and *posters* containing data produced by the module and for other (modules or the decisional level) to use.
- a *decisional level*: this level includes the capacities of producing the task plan and supervising its execution, while being at the same time reactive to events from the functional level. The coexistence of these two features, a time-consuming planning process, and a time-bounded reactive execution process poses the key problem of their interaction and their integration to balance deliberation and reaction at the decisional level.
- At the interface between the decisional and the functional levels, lies an *execution control level* that controls the proper execution of the services according to safety constraints and rules, and prevents functional modules from unforeseen interactions leading to catastrophic outcomes. In recent years, we have used the R2C [15] to play this role, yet it was programmed on the top of existing functional modules, and controlling their services execution and interactions, but not the internal execution of the modules themselves.

The organization of the overall system in layers and the functional level in modules are definitely a plus with respect to the ease of integration and reusability. Yet, an architecture and some tools are not "enough" to warrant a sound and safe behavior of the overall system.

In this paper the componentization method we propose will allow us to synthesize a controller for the overall execution of all the functional modules (which will be componentized) and will enforce by construction the constraints and the rules between the various functional modules. Hence, the ultimate goal of this work is to implement both the current functional and execution control level with BIP.

B. Componentization of GenoM Functional Modules

Each module of the LAAS architecture functional level is responsible for a function of the robot. Complex modalities (such as navigation) can be obtained by having modules "working" together. For example in Fig. 1 (which only shows the data flow of the functional level), there is an implicit processing loop. The module Laser RF acquires the laser range finder and store them in poster `Scan`, from which Aspect builds the obstacle map `Obs`. The module NDD (responsible for the navigation) avoids this obstacle while producing a `Speed` reference to reach a given target from the current position `Pos` produced by POM. Finally, this `Speed` reference is used by RFLEX, which controls the speed of the

[2]The GenoM tool can be downloaded from:
`http://softs.laas.fr/openrobots/wiki/genom`

robots wheels, and also produces the odometry position to be used by POM to generate the current position.[3]

All these modules are built using a unique generic canvas (Fig. 2) which is then instantiated for a particular robot function. We shall now describe this generic module, taking NDD as an example of instance, and identifying the "components" which are typeset in *italic*.

Each *module* can execute several *services* started on client requests. The module can send information relative to the executed requests to the client (such as the final report) or share data with other modules using *posters*. E.g. the NDD *module* provides six *services* corresponding to initializations of the navigation algorithm (definition of parameters), launching and stopping the path computation toward a given goal and a permanent service (*SetParams*, *SetDataSource*, *SetSpeed*, *Stop*, *GoTo* and *Permanent*). NDD also exports a poster (`Speed`) containing the speed reference.

The *services* are managed by a *control task* responsible for launching corresponding *activities* within *execution tasks*.

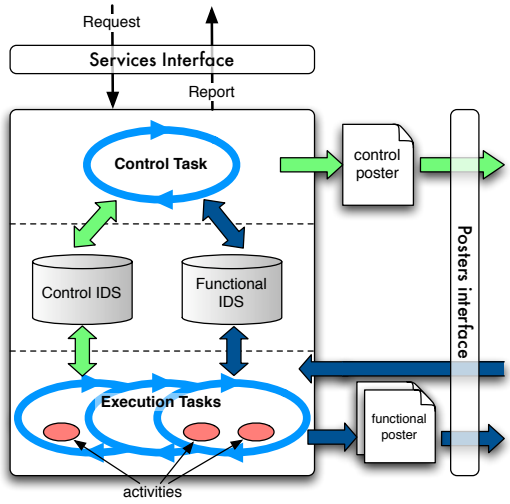


Fig. 2. A GenoM module organization.

Control and execution tasks share data using the internal data structures (IDS). Moreover execution tasks have periods in which the several associated *activities* are scheduled. It is not necessary to have fixed length periods if some services are aperiodic. Fig. 3 presents the behavior of an *activity*, inspired from classical thread life cycle. Activity states correspond to the execution of particular elementary code available through libraries and dedicated either to initialize some parameters (START state), to execute the activity (EXEC state) or to safely end the activity leading to reseting parameters, sending error signals, etc.

The component-based approach considers the functional level as a hierarchical entity which is decomposed successively into simpler components like modules, and further down to atomic components like execution tasks, activities, etc.

[3]This particular setup will serve as an example throughout the rest of the paper.

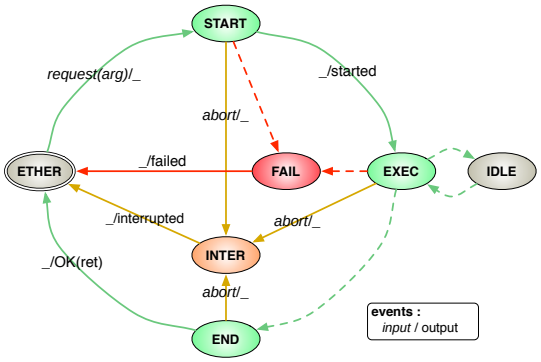


Fig. 3. Execution automaton of an activity.

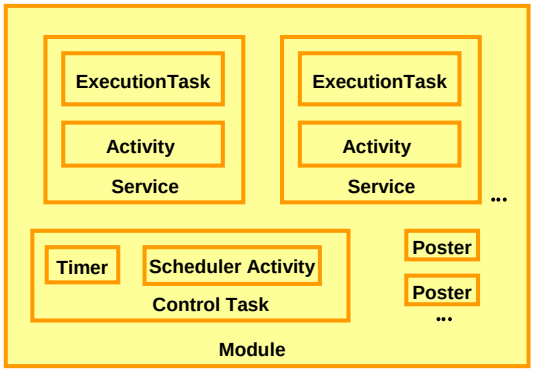


Fig. 4. A componentized GenoM module.

In order to formalize the componentization approach, we propose the following decomposition:

Functional level ::= (Module)+
Module ::= (Service)+ . (Control Task) . (Poster)+
Service ::= (Execution Task) . (Activity)
Control Task ::= (Timer) . (Scheduler Activity)

where "+" means the presence of one or more of the particular component and "." means the composition of different components. The componentized view of a GenoM module is shown in Fig. 4.

The next section introduces the BIP framework which has been used for the componentized implementation of the functional level of the robot.

III. The BIP component framework

BIP[4] [2] is a software framework for modeling heterogeneous real-time components. The BIP component model is the superposition of three layers: the lower layer describes the *behavior* of a component as a set of *transitions* (*i.e* a finite state automaton extended with data); the intermediate layer includes *connectors* describing the *interactions* between transitions of the layer underneath; the upper layer consists of a set of *priority* rules used to describe scheduling policies for

[4]The BIP tool-set can be downloaded from:
`http://www-verimag.imag.fr/~async/BIP/bip.html`.

interactions. Such a layering offers a clear separation between component behavior and structure of a system (interactions and priorities).

BIP allows hierarchical construction of *compound* components from *atomic* ones by using connectors and priorities.

An *atomic* component consists of a set of *ports* used for the synchronization with other components, a set of transitions and a set of local variables. Transitions describe the behavior of the component. They are represented as a labeled relation between *control states*.

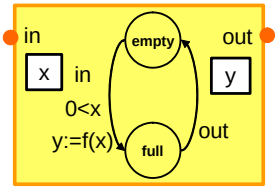


Fig. 5. An example of an atomic component in BIP.

Fig. 5 shows an example of an atomic component with two ports in, out, variables x, y, and control states $empty$, $full$. At control state $empty$, the transition labeled in is possible if $0 < x$. When an interaction through in takes place, the variable x is eventually modified and a new value for y is computed. From control state $full$, the transition labeled out can occur.

Connectors specify the interactions between the atomic components. A connector consists of a set of ports of the atomic components which may interact. An interaction of a connector is any non empty subset of its set of ports. A typing mechanism is used for the ports in order to determine the feasible interactions of a connector and in particular to model the two basic modes of synchronization, *rendezvous* and *broadcast*.

Priorities in BIP are a set of rules used to filter interactions amongst the feasible ones.

The model of a system is represented as a BIP compound component which defines new components from existing components (atoms or compounds) by creating their instances, specifying the connectors between them and the priorities.

The BIP framework consists of a language and a toolset including a front-end for editing and parsing BIP programs and a dedicated platform for the model validation. The platform consists of an Engine and software infrastructure for executing simulation traces of models. It also allows state space exploration and provides access to model-checking tools like *Evaluator* [12]. This permits to validate BIP models and ensure that they meet properties such as deadlock-freedom, state invariants and schedulability.

The back-end, which is the BIP engine, has been entirely implemented in C++ on Linux to allow a smooth integration of components with behavior expressed using plain C/C++ code.

The following section describes the modeling and verification of the functional layer of the robot in the BIP framework and its integration within the LAAS framework.

IV. Modeling, Verifying and Integrating the functional layer of the robot DALA in BIP

In modeling the functional layer of the robot in BIP, we have used the hierarchical decomposition of the functional layer as presented earlier in section II-B. Compound components are created by composing sub-components (atoms or compounds) using the connectors between them and priorities (if required), to build the hierarchy of the complete system.

For example, a compound component modeling a generic *service* is obtained from the atomic components *execution task* and *activity* and the connectors between them, as shown in Fig. 6.

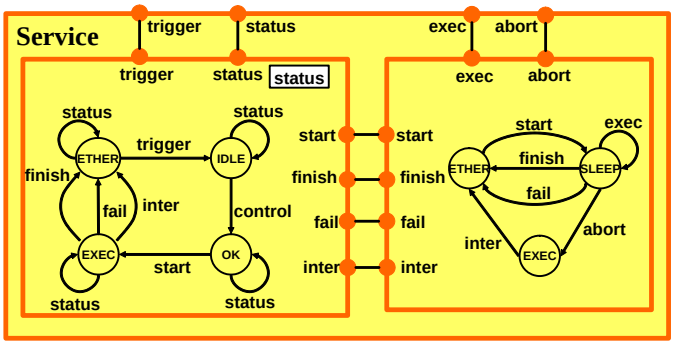


Fig. 6. BIP model of a service.

The left sub-component represents the execution task of a service. It is launched by synchronization through port *trigger*. The execution task then checks the validity of the parameters of the request (if available) and will either *reject* the request or *start* the activity by synchronizing with the activity component (right sub-component). In each state, the status of the execution task is available by synchronizing through port *status*. The activity will then wait for execution (i.e. synchronization on the *exec* port with the control task) and will either safely *finish*, *fail*, or *abort*. Each of the transitions *control*, *start*, *exec*, *fail*, *finish* and *inter* may call an external function.

The *service* components are further composed with *control task* and *poster* components to obtain the *module* components.

The full BIP description of the functional level of the robot, which consists of several modules, is beyond the scope of this paper. We rather focus on the modeling of the NDD module.

A. Modeling the NDD module in BIP

The NDD module contains six *services*, a *poster* and a *control task* as sub-components and the connectors between them, as shown in Fig. 7.

The *control task* wakes up periodically (managed by the bottom-left component with alternating sleep and trigger transitions) and always triggers the *Permanent* service at the beginning of each period. During a period, the services will have authorization to execute through interactions with the control task.

Moreover, the BIP formalism allows complex relations to be defined, such as:

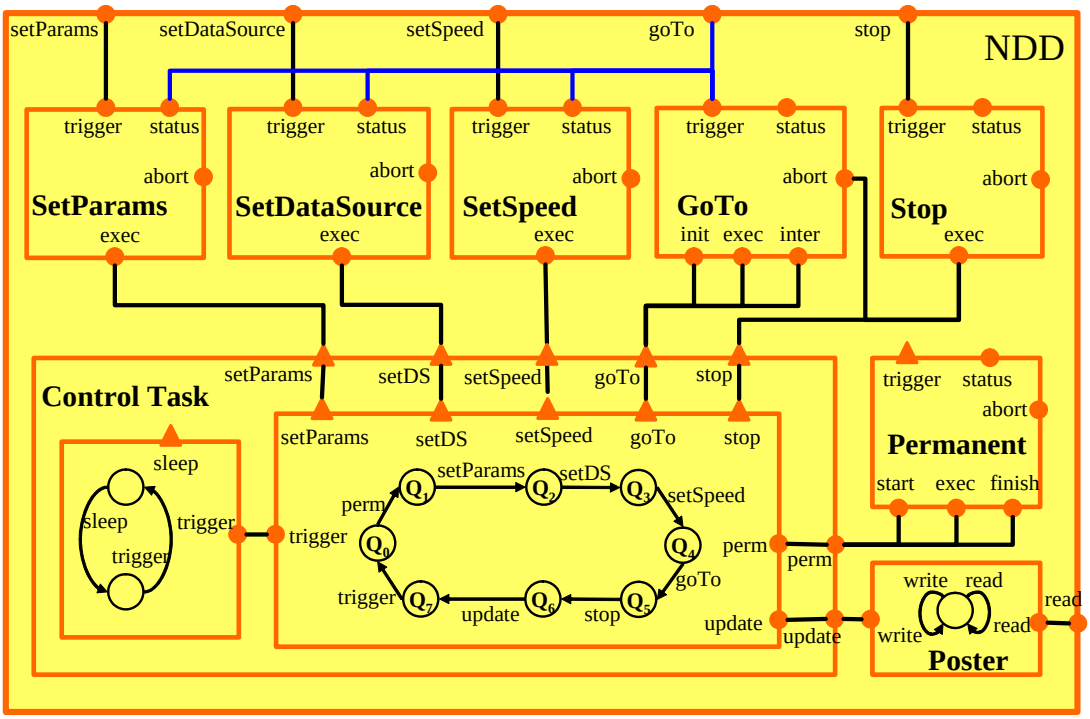


Fig. 7. The NDD module.

- interruptions, as modeled by the connector joining Stop.exec and GoTo.abort; if service Stop is executed, the GoTo algorithm will be aborted;
- constraints, as modeled by the goTo connector (in blue); service GoTo can be launched only if SetParams, SetSpeed and SetDataSource have been already completed (information available through their status port).

B. Executing and Monitoring the BIP model

The BIP tool-chain generates code from the BIP model, which can be executed by the BIP engine. The code contains calls to functions from libraries originally designed for GenoM modules, which executes the real activities of the robotic system. The code generated for the NDD module has been integrated and executed in the robot simulation environment of LAAS [9]. In particular, it was fully integrated with the decisional layer by replacing the functional layer originally modeled with GenoM with the one modeled in BIP.

There is a services interface in the architecture allowing each GenoM module to have its requests called from the decisional level. In the BIP model, each module (e.g. NDD, RFLEX, ...) has an additional component to accept requests and will synchronize with the corresponding ports of the module (for example those of NDD shown in Fig. 7). Hence, the requests sent by the decisional level are received by the component and reports are sent by each service upon completion according to the protocol used by GenoM modules. Indeed, the decisional layer does not need any modification to work with BIP.

The following section demonstrates how the methodology enforces by construction the constraints and the rules between the various functional modules.

C. Functional level Controller synthesis

In the LAAS architecture, a centralized controller (R2C) is used to control the proper execution of the services and to enforce the safety constraints and modules interactions. On the contrary, in the BIP model, we have used separate controllers for each service. The proper execution order and the safety properties are enforced by the BIP connectors between the controllers of different services. A BIP connector has guarded actions associated to each of its possible interactions. Dependency between the controllers of service in different modules are modeled by connectors associated with guards which represents either some valid execution condition or some safety rule. The composite behavior of these local controllers, synchronized by the connectors and restricted by priorities, is equivalent to the behavior of the centralized controller.

As an example, we had to enforce a rule between the NDD and the POM modules which states that the robot can navigate using the GoTo service of the NDD module only if the module POM has already executed successfully its Run service (which updates poster `Pos`). The rule is enforced by constructing a connector between port *trigger* of the Goto service and port *status* of the Run service, and guarded by the *status* value. The *status* value of the Run service is updated when Run has been successfully executed.

The next section presents in detail the methods used for the verification of the robotic system and their results.

D. Verification of Safety Properties

The BIP tool-set can perform an exhaustive state-space exploration of the system. Additionally, it can detect potential deadlocks in the system. These features have been used to verify some properties in the model of the robot and for detection of deadlocks. Two kinds of properties have been verified.

1) Safety properties: A safety property guarantees that something unexpected will never happen. For the verification of such properties, we used methods based on state-space exploration. The basic idea is to generate all reachable states and state changes of the system under consideration, and represent this as a directed graph called the *state-space*. Two different methods have been applied.

a) Model checking [16, 3]: We used the model-checker tool *Evaluator* [12] which performs on-the-fly verification of temporal properties on the state-space generated by the BIP engine on exploration of the system. As an example, we describe the usage of this method in verifying a safety property of the NDD module. It is required that the *GoTo* service is triggered only after a successful termination of *SetSpeed* service. To ensure this, in the BIP model of NDD, we need to guarantee that the interaction *GoTo:trigger* occurs only after the occurrence of the interaction *SetSpeed:finish*. We checked for violations of this property, i.e finding a transition sequence in the state-space where *GoTo:trigger* is not preceded by *SetSpeed:finish*. This safety property can be expressed in branching-time μ-calculus [11] as the following:

μX.(<"SetSpeed:finish">T or <¬"GoTo:trigger">X)

The result obtained by *Evaluator* proves that the initialization property is preserved in the NDD module.

b) Verification using Observers [18, 14]: For a given system S and a safety property P, we construct first an observer for P, i.e. an automaton which monitors the behavior of S and reports an error on violation of P. The verification consists of exploring the state-space of the product system. Such a method has been used to verify a timing property in the NDD module. It is needed to verify that the total time taken by all the services called within a period does not exceeds the period.

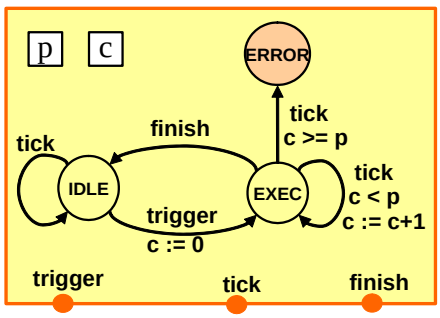


Fig. 8. Observer for the control task period verification.

In BIP, it is possible to model time as symbolic time [2] by using *tick* ports and clock variables in every timed component.

Time progress is by strong synchronization of all the *tick* ports. The clock variables are incremented on a *tick*, to model function execution times. Fig. 8 shows the observer component used to verify the timing property of the NDD module. It has a clock variable c and a parameter p representing the period of the *control task*. It synchronizes with the *control task* and tracks the cumulative time taken by the services triggered by *control task*. If this time exceeds the period p, the observer moves to the *ERROR* state. During exploration, if a global system state, containing the *ERROR* state of the observer is reachable, then the property is violated.

Such a method can also be used to verify timing properties between several modules. The processing loop presented in section II-B manages obstacle avoidance: obstacles detected by the laser are added to the aspect map which is used by NDD to compute a speed reference for RFLEX to control the robot velocity. The following time constraint can be verified: it should take less than a given time (e.g., a second depending on the current robot velocity) between the detection of an obstacle (data written in the Laser poster) and the speed reduction (the execution of the RFLEX permanent service).

2) Deadlock freedom: This is an essential correctness property as it characterizes a system's ability to perform some activity over its life time. The BIP toolset allow detection of potential deadlocks by static analysis of the connectors in the BIP model [7]. It generates a dependency graph and for each cycle in this graph, a boolean formula is generated. The satisfiability of the formula is then checked by the tool *minisat* [4], where a solution corresponds to a potentially deadlocked global state. Presence of an actual deadlock can then be verified by reachability analysis of the deadlocked states, starting from the initial state of the system. The analysis for the NDD module found a potential deadlock for the state where all services are in the *EXEC* state, all activities are in the *ETHER* state, and the control task is in the Q_0 state. However, this state is unreachable, hence the deadlock is not possible.

V. State of the art, Current Results and Prospective

The design and development of autonomous robots and systems is a very active research field. There are other architectures addressing similar problems: to provide an efficient, reusable and formally sound organization of robot software. CLARAty [13], used on various NASA research rovers, provides a nice object oriented hierarchical organization over two layers, but there is no formal model of the component interactions, nor modules canvas. IDEA [5], developed at NASA Ames, has an interesting modular/component organization with a temporal constraint based formalism. However, complexity of constraint propagation is an obstacle for effective deployment on real-time functional modules. RMPL [10, 19] and its associated tools, propose a system based on a model-based approach. The programmers specify state evolution

with invariants expressed in an "Esterel like" language and a controller maintaining them.

In [8], the authors present the CIRCA SSP planner for hard real-time controllers. This planner synthesizes off-line controllers from a domain description and then deduce the corresponding timed automata to control the system on-line. These automata can be formally validated with model checking techniques. However, this work focuses on the decisional part of the overall architecture. In [17] the authors present a system which allows the translation from MPL (Model-based Processing Language) and TDL (Task Description Language) to SMV, a symbolic model checker language. Compared to our approach, this does not address componentization and seems to be more designed for the high level specification of the decisional level.

The paper presents an approach integrating component-based construction and validation of robotic systems. It shows that a complex robotic system can be considered as the composition of a small set of atomic components. The use of BIP is instrumental for achieving this, as it provides powerful constructs for coordinating components. The combination of connectors to describe interactions between components, and priorities to enforce scheduling policies, proves to be essential for incremental modeling. The global model is obtained by progressively composing its atomic components. It is possible to identify in the global model all its atomic components and their interactions. This allows in particular, to study the impact of changes of a component's behavior or structure on the global behavior and its properties. The paper shows that it is possible to combine standard verification techniques, based on global state exploration, with structural analysis techniques for deadlock detection. This is a very interesting work direction that will be further investigated.

Another useful work direction is the online monitoring of the functional level execution using observer components, which would be able to generate feedback actions for the decisional level which can be useful for error-recovery or restarting of services.

Our work is based on the idea that componentization and reasoned construction supported by an appropriate methodological framework and tools, are instrumental for coping with system design complexity. It is possible to enforce by construction some design requirements and avoid as much as possible an a posteriori validation of the global system. We will work towards achieving such a challenging goal in several directions. These include the formalization of the componentization methodology in particular regarding the interplay between atomic components and their coordination mechanisms. Another important work direction is achieving constructivity, that is guaranteeing some properties by construction or by lightweight global analysis. Finally, the methods and tools should be improved in particular to support incremental system construction and analysis.

REFERENCES

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, 'An architecture for autonomy', *International Journal of Robotics Research, Special Issue on Integrated Architectures for Robot Control and Programming*, **17**(4), (1998).

[2] A. Basu, M. Bozga, and J. Sifakis, 'Modeling heterogeneous real-time components in BIP', in *International Conference on Software Engineering and Formal Methods (SEFM)*, Pune, India, (2006).

[3] E. M. Clarke and E. A. Emerson, 'Synthesis of synchronization skeletons for branching time temporal logic', in *Workshop on Logic of Programs*, Yorktown Heights, NY, USA, (1981).

[4] N. Eén and N. Sörensen, 'An extensible SAT−solver', in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, Portofino, Italy, (2003).

[5] A. Finzi, F. Ingrand, and N. Muscettola, 'Robot action planning and execution control', in *International Workshop on Planning and Scheduling for Space (IWPSS)*, Darmstadt, Germany, (2004).

[6] S. Fleury, M. Herrb, and R. Chatila, 'GenoM: A tool for the specification and the implementation of operating modules in a distributed robot architecture', in *International Conference on Intelligent Robots and Systems (IROS)*, Grenoble, France, (1997).

[7] G. Goessler and J. Sifakis, 'Component-based construction of deadlock-free systems', in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Bombay, India, (2003). Invited talk.

[8] R. P. Goldman and D. J. Musliner, 'Using model checking to plan hard real-time controllers', in *AIPS Workshop on Model-Theoretic Approaches to Planning*, Breckenridge, CO, USA, (2000).

[9] S. Joyeux, A. Lampe, R. Alami, and S. Lacroix, 'Simulation in the LAAS architecture', in *ICRA Workshop on Interoperable and Reusable Systems in Robotics*, Barcelona, Spain, (2005).

[10] P. Kim, B. C. Williams, and M. Abramson, 'Executing reactive, model-based programs throgh graph-based temporal planning', in *International Joint Conference on Artificial Intelligence (IJCAI)*, Seattle, WA, USA, (2001).

[11] D. Kozen, 'Results on the propositional μ-calculus', *Theoretical Computer Science*, **27**, (1983).

[12] R. Mateescu and M. Sighireanu, 'Efficient on-the-fly model-checking for regular alternation-free mu-calculus', Technical Report 3899, INRIA Rhône-Alpes, France, (2000).

[13] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin, 'CLARAty and challenges of developing interoperable robotic software', in *International Conference on Intelligent Robots and Systems (IROS)*, Las Vegas, NV, USA, (2003). Invited paper.

[14] M. Phalippou, 'Executable testers', in *International Workshop on Protocol Test Systems (IWPTS)*, Tokyo, Japan, (1994).

[15] F. Py and F. Ingrand, 'Dependable execution control for autonomous robots', in *International Conference on Intelligent Robots and Systems (IROS)*, Sendai, Japan, (2004).

[16] J-P. Queille and J. Sifakis, 'Specification and verification of concurrent systems', in *International Symposium on Programming*, Torino, Italy, (1982).

[17] R. Simmons, C. Pecheur, and G. Srinivasan, 'Towards automatic verification of autonomous systems', in *International conference on Intelligent Robots & Systems (IROS)*, Takamatsu, Japan, (2000).

[18] J. Tretmans, 'A formal approach to conformance testing', in *International Workshop on Protocol Test Systems (IWPTS)*, Tokyo, Japan, (1994).

[19] B. C. Williams, M. D. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. T. Sullivan, 'Model-based programming of fault-aware systems', *Artificial Intelligence*, **24**(4), (2003).

A formal approach to designing autonomous systems: from Intelligent Transport Systems to Autonomous Robots

Fabrice KORDON[1] and Laure PETRUCCI[2]

[1] Université Pierre & Marie Curie, Laboratoire d'Informatique de Paris 6/MoVe
4, place Jussieu, F-75252 Paris CEDEX 05, France
fabrice.kordon@lip6.fr
[2] LIPN, CNRS UMR 7030, Université Paris 13
99, avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
laure.petrucci@lipn.univ-paris13.fr

Abstract. Emerging transport systems involve more and more fully automatic parts that communicate together in order to optimise traffic and security. Such systems are highly distributed, mobile and require physical constraints to be taken into account. The communicating entities may be included in vehicles or the infrastructure ; they must comply with real time and real space constraints ; they should also have some autonomous behaviour in case of e.g. network failure. The systems designed should be proved reliable before being put in operation. We propose to use formal specification and verification techniques for designing these models, prior to any costly hardware implementation.

1 Introduction

Emerging transport systems involve more and more fully automatic parts that communicate together in order to optimise traffic and security. Such systems are highly distributed, mobile and require physical constraints to be taken into account. The communicating entities may be embedded in vehicles or included in the infrastructure.

In such systems, distribution leads to a huge complexity and a strong need to deduce possible (good and bad) behaviours on the global system, from those known of its actors. Moreover, at least part of these systems is embedded with intrinsic real-time and real-space constraints. They are due to the critical, highly reactive environment where both timing and positionning are critical issues.

For such systems, we know that classical development methods are not adequate since the coverage of possible executions is too low [GL97]. This observation leads to investigate the use of formal methods. However, these still lack user-friendly languages and tools that can enable their use by non-specialists. Hence, even though major actors in companies or institutions dealing with critical applications acknowledge the necessity of using formal methods, they also agree on the fact they should be able to scale up: today, only parts of systems are formally analysed.

Up to now, two main types of formal methods are available: *algebraic* approaches and *model checking*. Algebraic approaches such as B [Abr96] allow for describing a

system with axioms and then proving a property on the specification as a theorem to be demonstrated from these axioms. These methods are very interesting since the proof is parameterised. However, theorem provers that are required to elaborate the proof are difficult to use and still require highly skilled and experienced engineers.

In contrast, model checking [CGP00,BBF+01] is the exhaustive investigation of a system state space and can be automated very easily. This technique is theoretically limited by the combinatorial explosion and can mainly address finite systems. However, recent symbolic techniques[3] scale up to more complex systems.

Thus, since formal verification techniques are getting more mature, our capability to build even more complex systems also grows quickly. To catch up with problems complexity and get significant results with formal analysis, we must cope with the complexity at every stage of the process: from the specification phase to the verification itself. The methodology to be applied makes a *pragmatic* use of formal methods, i.e. assumptions simplifying the system under study should be made, which are usually domain specific. Hence, variations of the traditional (unified) process development approaches are necessary.

This paper proposes to tackle the design methodology and techniques that can be applied in order to handle very large systems throughout the modelling and verification processes. Such techniques are here concerned with Intelligent Transport Systems (ITS), i.e. mechanisms which provide driving assistance to a vehicle. This application domain is particularly representative of tomorrow distributed systems including real-time and real-space features, for which traditional programming approaches cannot guarantee the required security, and must thus be adapted.

The paper is structured as follow. Section 2 presents ITS concerns and the related verification problems. Then, section 3 describes the formal notations that we shall use to model such systems. A design methodology is sketched in section 4. Section 5 addresses verification using appropriate model checkers. Finaly, section 6 shows the parallel between Intelligent Transport Systems and autonomous robots.

2 Intelligent Transport Systems

Intelligent Transport Systems (ITS) are highly critical since a failure can lead to dramatic consequences such as fatal accidents. They also involve a significant number of partners that must thus cooperate in an efficient and secure manner. The agents of such a system are road operators, infrastructure, vehicles and their drivers. Some of these might be equipped with active embedded software while the others travel in the usual fashion. Their reaction is then unpredictable and it is essential to obtain relevant and often updated information from captors in order to take them into account.

Development of ITS is a challenge supported by research programs in Europe, USA and Japan [Bis05].

In this section, some ITS issues are first illustrated through a simple example. Then, major problems encountered during the formal specification are discussed.

[3] The word *symbolic* is associated with two different techniques. The first one is based on state space encoding and was introduced in [BCM92]. The second one relies on set-based representations of states having similar structures and was introduced in [CDFH91].

2.1 ITS Example: Safe Insertion in a Motorway

A typical example of an ITS problem is the so-called *black-spot*, which is a dangerous section in the motorway. It is basically a freeway entrance in which *safe insertion* should be guaranteed. Figure 1 represents a motorway with two lanes: L_1 (right lane) and L_2 (left lane). An entrance to the motorway, L_0, is connected to L_1. Vehicles already on the motorway use both lanes L_1 and L_2. Vehicles are supposed to carry an identity which is a number. The notation $V_{i,j}$ indicates that vehicle j is circulating on lane number i. We aim at studying a cooperative insertion of vehicles arriving in the entrance lane L_0.

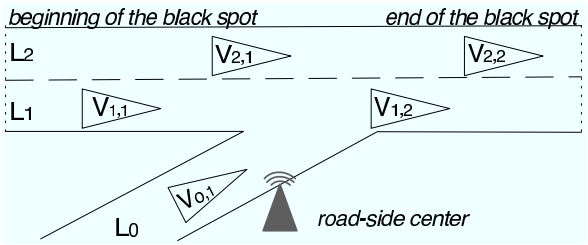


Fig. 1. Safe Insertion in a motorway.

The vehicles on the insertion lane L_0 must enter the motorway without violating the following properties:

1. the distance between two vehicles in the same lane must be greater than a minimum safe distance to let drivers react to sudden and maybe unexpected events;
2. $V_{0,j}$ vehicles must eventually get into the motorway;
3. $V_{i,j}$ vehicles should not have to stop.

We propose the following strategy to ensure a safe motorway entrance:

(a) The motorway has a *road-side center* (RSC) enabling communication with vehicles and which can compute commands related to safety or flow control.
(b) Vehicles receive their positions using satellite localisation technology [Blo05] which may also be combined with ground installations and digitised maps). They periodically their position to the infrastructure. Subsequently, the infrastructure maintains a dynamic map of all vehicles in its communication range.
(c) The infrastructure, vehicles behaviour and interactions operate the following interaction cycle:
  (i) vehicles get their position;
  (ii) they send this information to the infrastructure;
  (iii) when the infrastructure has received all vehicles positions, it issues commands w.r.t. a predefined strategy.

In order to simplify the problem, we assume that all vehicles are equipped with communication devices and drivers follow the instructions issued by the road-side center. Vehicles without the embedded equipment are considered and modelled differently.

2.2 Modelling Issues for ITS

For such critical and reactive systems, both *quantitative* and *qualitative* properties should be ensured. *Quantitative* properties express performance requirements while *qualitative* properties allow for checking whether a faulty behaviour may occur.

Modelling and verifying such systems require:

- managing dynamic actors such as cars that enter and leave the black-spot;
- modelling of physical aspects;
- preserving a fair progression of the system so that actors perform actions at a similar pace.

A qualitative analysis is a first step in the development process: it will ensure a global correct functionning. Then, the specification can be refined so as to include timing or hybrid features, reflecting the actual real-time and real-space behaviour. Quantitative analysis will determine whether the envisionned strategies satisfy physical constraints. In this paper, we will focus on the qualitative aspects.

2.3 Specification Issues

The first issue consists in *selecting an appropriate specification formalism* [HKP06]. Of course, the formalism should be able to capture the relevant aspects of the problem under study, and allow for the necessary verifications. Moreover, a *design methodology* will prove useful, to carry out verifications step by step on more amenable models.

The choice of a specification formalism and the design methodology are of utmost importance for verification to be as successful as possible.

A very popular candidate for specification is UML [COTM05]. Even though, it is useful for structuring a system and having a better view of the interactions between components, UML is not suitable for formal analysis of the system behaviour. Normalisation efforts tends to counter this problem by giving a more precise semantics, but the connection between diagrams is still too loose and leads to various interpretations.

Algebraic techniques such as B can be useful for the verification of behavioural components as Siemens proved in the METEOR project [B]. However, it was also known as a difficult technique to automate compared to model checking based approaches. Thus, this latter type of techniques seems better suited to provide more automated tools.

Many tools allow for model checking. They address different kinds of models. In a first approach, we will focus on the verification of behavioural properties of the system, i.e. qualitative analysis. Thus, we will be able to check that the chosen strategies are relevant, indepently of real-time or real-space constraints. Therefore, a simple model is selected, which is powerful enough for our modelling purposes and provides up-to-date efficient analysis techniques, namely *Symmetric Nets*[4]. The CPN-AMI tool [cpn] constitutes a complete spefication and verification environment for symmetric nets.

[4] *Symmetric Nets* were formerly known as *Well-Formed Nets*, a subclass of *High-level Petri Nets*. The new name was chosen in the context of the ISO standardisation of Petri nets [HKPT06].

In a further step of our modelling and verification process, the models will be enhanced so as to capture the real-time and real-space aspects, i.e. perform quantitative analysis. Model checking tools can handle time (e.g. UPPAAL [upp], based on timed automata; or TINA [tin] based on Timed Petri Nets) or hybrid constructs (e.g. HYTECH [hyt]). It will then be necessary to design a methodology which guarantees a compatibility between the models designed for qualitative analysis and those derived for quantitative analysis purposes.

The remainder of this paper will on behavioural properties. Therefore, section 3 presents the symmetric nets formalism.

3 Symmetric Nets

This section provides an informal presentation of Symmetric Nets as well as associated analysis techniques. Formal definitions can be found in [CDFH91,GV03].

3.1 Formalism Basic Features

Symmetric Nets are Petri nets enhanced with high-level features: tokens can carry data. Therefore, a data type is associated with each place, indicating the data type for tokens sitting in that place. In contrast with Coloured Petri Nets [Jen92], only simple data and manipulation functions are permitted, allowing for powerful analysis techniques. Finite enumerated types, intervals, tuples are allowed and the basic functions are predecessor, successor, selector (in a tuple) and "broadcast". The latter function allows for generating one copy of each possible value in the data type. This is very convenient for e.g. modelling network protocols where a station sends messages to all other stations on the network.

Let us now illustrate Symmetric Nets (SN) by means of a small example. The Petri net in figure 2 represents a class of threads (identified by an identity in type *P*) accessing a critical resource *CR*. Threads can get a value within the type *Val* from **CR**. Constants **PR** and **V** are parameters for the system.

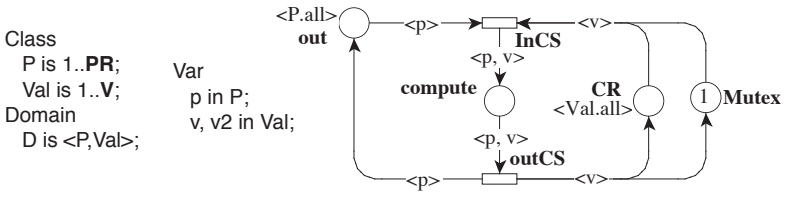


Fig. 2. Example of a Symmetric Net.

The class of threads is represented by places **out** (typed after *P*) and **compute**. Place **compute** (typed after $D = P \times Val$) corresponds to some computation on the basis of the

value provided by **CR**. At this stage, each thread holds a value that is given back when the calculus is finished. Place **Mutex** handles mutual exclusion between threads. Place **out** initially holds one token for each value in P (the marking is then noted $< P.all >$) and place **CR** holds one value for each value in Val (marking $< Val.All >$). Place **Mutex** only contains one token with no value (like a black token in place/transition nets).

Transitions represents evolution of the system. A transition is fired when all precondition places hold a sufficient marking. For example, Transition **inCS** can be fired if there is one token in **out**, one token in **CR** and one token in **Mutex**. When it fires, variables p and v are bound to the values of tokens from place **out** and **CR** respectively (place **Mutex** has no type, hence its tokens do not carry any data). When this transition is fired, a token carrying the value of pair $< p, v >$ is created in the postcondition place **compute**.

3.2 From Symmetric Nets to Place/Transition Nets

A symmetric net can easily be *unfolded* into an equivalent place/transition net.

A SN-place is transformed into a set of PTN-places, one per possible value. The place/transition net in figure 3 is the unfolding of the symmetric net in figure 2 with $P = [1..2]$ and $Val = [1..2]$.

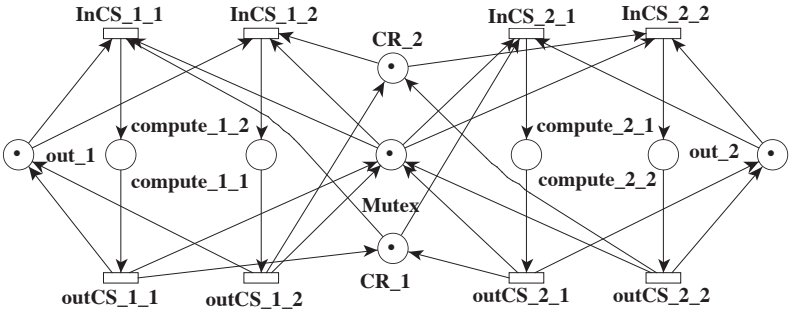


Fig. 3. Unfolded P/T Net from figure 2.

Even large models can be handled, using decision diagram based techniques [KLPA06].

Then, structural properties of the model can be computed from the unfolded net. They are formulas that can be computed without exploring the full state space [GV03], and hold independently of the initial marking (in fact, it often intervenes in a constant value only).

3.3 Symbolic Reachability Graph

One of the main analysis techniques is based on the *state space* (also called *reachability graph*) exploration. It represents all concrete states and possible evolutions of the system. Figure 4 presents the reachability graph for the Petri net of figure 2 with constants

PR and **V** equal to 2. This state space has 5 states (the initial state is represented by a double circle). When increasing the number of possible values, the size of the state space will grow following the cardinality of the cartesian product $P \times Val$.

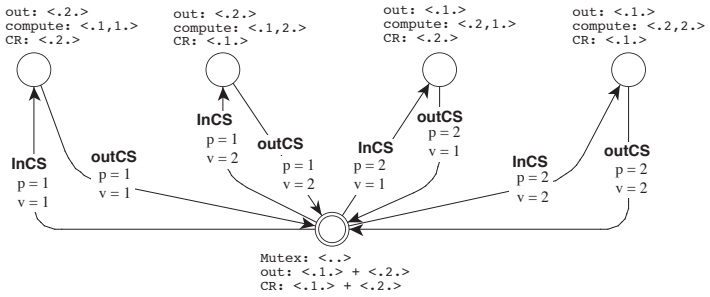


Fig. 4. Reachability Graph of the Net in Figure 2.

Similar to symmetric nets which are a compact representation of a system, the *symbolic reachability graph* is a condensed reprsentation of the states in the system, particularly adapted to the analysis of symmetric nets. A state in the symbolic reachability graph does not represent a concrete state but a set of concrete states that have a similar structure. The symbolic reachability graph of our example is depicted in figure 5. It is composed of only two nodes and does not grow when the types P and Val allow for more values.

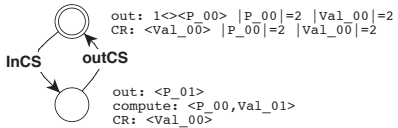


Fig. 5. Symbolic Reachability Graph of the Net in Figure 2.

The definition of states in figure 5 must be read as follow. In the initial state, all possible values in type P are stored in place **out** and all possible values in type Val are stored in place **CR**. In the other state (when transition **InCS** fires), all possible values of type P but one are in place **out** and all possible values of type Val but one are in place **CR**. Place **compute** then contains one token composed with one value of type P (the one that is not in place **out**) and one value of type Val (the one that is not in place **CR**). Thus, this symbolic state represents all possible permutations of the pair of tokens extracted from places **out** and **CR** when firing transition **InCS**.

This symbolic technique is thus based on computed symmetries in the net [TMDM03]. It is successful when representing very large state spaces: there is an exponential gain w.r.t. the construction of concrete states [HTMK+04]. This set-based representation is very efficient, especially when systems are symmetric, which is the case in numerous distributed and embedded systems.

Such a technique is well-suited for analysing Intelligent Transport Systems since they present intrinsic symmetries as similar algorithms are supposed to be executed in each car.

4 Modelling Methodology for ITSs

Considering that the specification formalism for Intelligent Transport Systems is Symmetric nets, we now aim at a suitable dedicated design method. Therefore, we will consider the techniques that are optimal for this kind of systems.

4.1 Model Components and Abstraction Level

The whole ITS system to be modelled involves several components that interact via several mechanisms. The communication can either be asynchronous (and is then modelled by *place fusion*) or synchronous (corresponding to *transition fusion*).

It is thus necessary to enhance the symmetric nets formalism with both of these communication mechanisms. They provide further advantages. In particular, several designs of a same component can be tested without changing the models of the other components, provided that all these designs communicate in the same manner with the rest of the system. But they can operate different strategies or configurations.

This approach leads to a hierarchy of modules, which can have as many levels as necessary. Analysing the system then starts by "flattening" the whole model into a single symmetric net. It is also possible to take advantage of modular analysis techniques such as [LP04].

Since the systems under study are complex, we also consider using refinement of modules [LL01]. This approach allows for first designing an abstract model, analysing it, and then include additional details in a consistent manner (so as to keep the analysis results as much as possible). This process is repeated until the desired abstraction level is obtained. A similar modelling approach will lead to include the real-time and real-space features once the global functionning of the system is proved correct.

Finally, the specific problems identified in section 2.2 must be addressed.

4.2 Managing Dynamic Actors

The vehicles involved in the black-spot are dynamic: they can enter and leave the motorway zone nder study. A natural way of modelling this aspect would be to create new vehicles getting in and discarding vehicles getting out.

However, such an approach is not suitable for several reasons. First, that would mean associating a new number with each new vehicle. The chosen formalism of symmetric nets permits only finite types, thus having a arbitrary high numbering of vehicles

is not feasible. Moreover, let us assume that there is a maximum numbering of vehicles. The states space of the system will be uselessly large when taking into account many different vehicle numbers.

An approach to managing this dynamic aspect is then to consider that there can be a limited number of vehicles on the black-spot. This is a reasonnable assumption since the cars and motorway lengths are a physical limit. Moreover, the identity of vehicles has no consequence on the functionning of the system. They must still be distinguished to consider e.g. different positionning or driving strategies. Hence, when a vehicle gets out of the system, its number can be reused by a new vehicle.

This technique also brings forward an interesting feature. The corresponding system is not expected to deadlock. Thus, a deadlock may correspond either to a property violation or to some mistake in the model itself.

4.3 Modelling Complex Functions

As mentionned in section 3.1, symmetric nets only offer a limited set of mathematical functions to the system designer. This is required for keeping the mathematical structure that enables the computation of symmetries in the specification, necessary for using the symbolic reachability graph [TMDM03].

To cope with the modelling of complex functions (for example, computation of braking distance according to the current speed of a vehicle), they can be discretised and represented in a dedicated place of the Petri net. This approach is similar to sampling and can be applied to arbitrarily complex functions, deterministic or not. However, the discretisation of a function becomes a modeling hypothesis and must be validated separately (to evaluate the accuracy of the sampling).

The main drawback of this technique is a loss in precision compared to continuous systems that require appropriate hybrid techniques [CEF05]. If such a discretisation enables the use of more user-friendly techniques, they must be checked. For example, if we consider distances in our black-spot example, we must ensure that uncertainty remains in a safe range. This means that our metrics must be compliant with the precision to ensure, for example, that if $V_{1,1}$ follows $V_{1,2}$, the minimum distance guarantees that no intersection between the associated volumes is possible.

Using the discretisation technique may be a preliminary to putting in operation more complex formalisms such as timed or hybrid ones which will allow for quantitative analysis of continouous models.

4.4 Fair Execution among System Components

The different actors in an ITS behave in parallel. Their actions should evolve at a similar pace. This aspect is not guaranteed by Petri net behaviour unless appropriate mechanisms are set. To avoid the progress of one component while the other components are stopped, several techniques can be used:

– the addition of a timeline, as in Timed Petri nets [Jen92] changes the firing conditions so that time can advance only if there is no enabled transition at the current time anymore;

– the state space construction could include a branching option that would discard unsuitable sequences.

In the black-spot example, at each time slot, all vehicles should make a move and the infrastructure take decisions, thus following the execution cycle described in section 2.1.

5 Towards Analysis of Intelligent Transport Systems

As mentionned in section 4.2, ITSs include many components, namely the vehicles, which have a similar behaviour. The vehicles identifiers are only used to track them within the system. Therefore, this kind of system is highly symmetric: the diffrent vehicles play the same role. Thus, the use of symmetric nets and the associated analysis technique, i.e. the symbolic reachability graph (see section 3.3), is relevant.

Nevertheless, analysis remains quite difficult since currently implemented model checkers are not sufficient. The ones that implements a concrete state space cannot handle more than a few 10^8 states.

GreatSPN [gsp], a model checker implementing the symbolic reachability graph was successfully used to analyse a middleware core having approximately 10^{18} concrete states [HTMK$^+$04], but it seems inadequate for the complexity of ITS systems when discretisation is realistic and requires types with many values (in [BHKF06], only small configurations could be analysed). Model checkers supporting a symbolic encoding of the state space such as SVM [smv] present the same drawbacks.

A close examination of model checkers behaviour shows that current techniques cannot scale up for these systems yet.

However, model checkers use new techniques that are promising for analysing ITSs:

– symbolic/symbolic techniques;
– distributed model checkers running on clusters of machines;
– handling stable markings;
– hierarchical encoding and modular techniques.

In the following subsections, we will sketch these.

5.1 Symbolic/Symbolic Techniques

The symbolic reachability graph, as described in section 3.3, allows for mastering the complexity of large state spaces, similar to the encoding of states using decision diagrams [BCM92] (also called *symbolic techniques*). The term *symbolic* can thus have sevral meanings. It can relate to:

– grouping of similar states represented by a single abstract one;
– adequate encoding of the states to have a better use of computer memory.

To illustrate the state encoding techniques, let us consider that a state in the system is represented as a boolean vector defining the values of a set of variables. An action in the system usually changes only part of the system state. Hence, we can consider a

differential encoding of states. It is not necessary to encode once more the value of the unchanged variables. Binary Decision Diagrams (BDDs) promote sharinf of common parts in the system. The main drawback of this technique is that its efficiency is strongly related to the ordering of variables.

BDDs are dedicated to systems using booleans, but many decision diagrams based techniques were introduced so as to capture more elaborate models. Among them, Data Decision Diagrams (DDDs) [CEPA+02] encode discrete values instead of binary ones. It is a basis to support symbolic/symbolic techniques [TMIP04] that combine *symbolic encoding* and *symbolic reachability graph*. Experiments show that this technique is promising for the storage of very large state spaces.

5.2 Distributed Model Checking

The main problem of model checking is memory consumption. However, with diagram decision based techniques, another problem arises. The principle of these techniques is to trade memory against CPU. As a typical example, when a new symbolic state is computed, it has to be compared with the existing ones. This requires all states to be canonised in order to have a common and comparable representation.

So, distributing a model checker on a cluster of machines has advantages [KP04]:

- states are generated in parallel using a hash function which distributes states on machines;
- it takes advantage of the CPU and memory available in the whole system.

Initial results are promising and this is a currently active research area. SPIN model checker has already been experimented in a parallel setting [LS99,BFLW05]. A distributed version of GreatSPN has been recently implemented, which provides a supralinear acceleration factor for many examples [HKTM07]. However, even though the distributed generation of the state space has been implemented, analysing properties on those is still to be developped further.

5.3 Management of Stable Marking

The discretisation technique presented in section 4.3 generates places with a large marking which remains constant. Most model checkers do not handle such cases, and the stable marking is represented once per generated state, leading to a huge and useless memory consumption.

Model checkers using a symbolic encoding of states, such places should be detected since their marking is highly shared by all states in the system. An *a priori* analysis of the specification can easily detect such configuration and provide hints for a more appropriate encoding technique.

5.4 Hierarchical Encoding and Modular Techniques

Symbolic encoding of a state space (concrete or symbolic) relies on the sharing of state patterns in the state space of a system. Recent work investigates a hierarchical

representation that could increase the sharing of such patterns on a larger scale. For that purpose, new representations, such as Set Decision Diagrams (SDD) [CTM05] are being investigated. In favourable cases (i.e. when the system exhibits very regular symmetries), the results are impressive. E.g. it is possible, using a recursive folding of the dining philosophers problem [phi], to store the state space for to 2^{10000} philosophers within 512 Mbytes of memory, as reported in [TM04]. The intrinsic high symmetry in ITSs should allow for similar results.

An additional and complementary investigation axis is modular analysis [LP04]. Since the methodology applied for specifying large systems includes design by component, this kind of analysis techniques is suitable. The idea consists in representing the state space using not a single graph, but several: one per component, representing the component local behaviour and local states, plus a synchronisation graph which explicits the synchronisations between the different components and the global states to achieve those.

Combining modular and symbolic techniques should permit to handle very large systems such as ITSs.

6 Towards Autonomous Robots

Autonomous robots are meant to evolve within an environment which may have unexpected behaviour. This can be due to e.g. unknown terrain to explore, other agents (e.g. robots handled by another system, animals), ... 'Moreover, the human control on such robots is rather minor. In order for autonomous robots to take into account the characteristics of the environment, it is necessary for them to get information via sensors. This allows for having an abstract vision of the current situation. Thus, as for ITSs, modelling autonomous robots requires both real-time and real-space concerns to be taken care of.

Thus, the approach developped here for ITSs can be used to tackle autonomous robots verification as well. The analogy could be relating the robots themselves to the vehicles and the interacting humans to the infrastructure. The map of the environment for ITSs is rather simple and it may be more complex for robots. However, there is often an intended route which can initially be modelled using simple data and symmetric nets and can later be further refined when taking into account the space and time aspects.

7 Conclusion

Emerging transport systems involve more and more fully automatic parts that communicate together in order to optimise traffic and security. Such systems are highly distributed, mobile and require physical constraints to be taken into account. The communicating entities may be included in vehicles or the infrastructure ; they must comply with real time and real space constraints ; they should also have some autonomous behaviour in case of e.g. network failure. The systems designed should be proved reliable before being put in operation.

In this paper, we have shown how the design methodologies and current analysis techniques can handle such very large systems. The different approaches have their own

advantages and are complementary. Therefore, our efforts will focus on their combination. An obvious necessity emerging from preliminary analysis is to consider design and analysis issues in parallel so as to capture and handle the relevant problems in a consistent and efficient manner. In particular, domain specific features have to be taken into account at a very early stage.

References

[Abr96] J-R. Abrial. *The B book - Assigning Programs to meanings*. Cambridge University Press, 1996.

[B] *Atelier B*.

[BBF+01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

[BCM92] J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation (Special issue for best papers from LICS90)*, 98(2):153–181, 1992.

[BFLW05] J. Barnat, V. Forejt, M. Leucker, and Michael Weber. DivSPIN - a SPIN compatible distributed model checker. In M. Leucker and J. van de Pol, editors, *4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, Lisbon, Portuga, 2005.

[BHKF06] F. Bonnefoi, L. Hillah, F. Kordon, and G. Frémont. An approach to model variations of a scenario: Application to Intelligent Transport Systems. In *Workshop on Modelling of Objects, Components, and Agents (MOCA'06)*, Turku, Finland, June 2006.

[Bis05] R. Bishop. Intelligent Vehicle R&D: a review and contrast of programs worldwide and emerging trends. *Annals of Telecommunications - Intelligent Transportation Systems*, 60(3-4):228–263, March-April 2005.

[Blo05] J-M. Blosseville. Driving assistance systems and road safety: State-of-the-art and outlook. *Annals of Telecommunications - Intelligent Transportation Systems*, 60(3-4):281–298, March-April 2005.

[CDFH91] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. On well-formed coloured nets and their symbolic reachability graph. In Kurt Jensen and Grzegorz Rozenberg, editors, *Procedings of the 11th International Conference on Application and Theory of Petri Nets (ICATPN'90). Reprinted in High-Level Petri Nets, Theory and Application*. Springer, 1991.

[CEF05] P. Christofides and N. El-Farra. *Control Nonlinear And Hybrid Process Systems: Designs for Uncertainty, Constraints And Time-delays*. Springer, 2005.

[CEPA+02] J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier. Data decision diagrams for Petri net analysis. In *Proc. of ICATPN'2002*, volume 2360 of *Lecture Notes in Computer Science*, pages 101–120. Springer, June 2002.

[CGP00] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[COTM05] B. Charroux, A. Osmani, and Y. Thierry-Mieg. *UML 2*. Pearson Education, 2005.

[cpn] *The CPN-AMI Home page*. `http://www.lip6.fr/cpn-ami`.

[CTM05] J-M. Couvreur and Y. Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In *25th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'05)*, Lecture Notes in Computer Science. Springer, October 2005.

[GL97] J. Gogen and Luqi. Formal methods: Promises and problems. *IEEE Software*, 14(1):75–85, 1997.

[gsp] *GreatSPN V2.0*. `http://www.di.unito.it/~greatspn/index.html`.

[GV03] C. Girault and R. Valk. *Petri Nets for Systems Engineering*. Springer Verlag - ISBN: 3-540-41217-4, 2003.

[HKP06] S. Haddad, F. Kordon, and L. Petrucci. *Méthodes formelles pour les systèmes répartis et coopératifs*. Hermès, November 2006.

[HKPT06] L. Hillah, F. Kordon, L. Petrucci, and N. Trèves. PN standardisation : a survey. In *International Conference on Formal Methods for Networked and Distributed Systems (FORTE'06)*, pages 307–322, Paris, France, September 2006. IFIP.

[HKTM07] A. Hamez, F. Kordon, and Y. Thierry-Mieg. libDMC: a Library to Operate Efficient Distributed Model checking. Technical report, Master's thesis, LIP6, Université P. & M. Curie, 2007.

[HTMK+04] J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir, and T. Vergnaud. On the Formal Verification of Middleware Behavioral Properties. In *9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, pages 139–157. Elsevier, September 2004.

[hyt] HYTECH *homepage*. `http://www-cad.eecs.berkeley.edu/~tah/HyTech/`.

[Jen92] K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and practical use. Volume 1: basic concepts*. Monographs in Theoretical Computer Science. Springer, 1992.

[KLPA06] F. Kordon, A. Linard, and E. Paviot-Adet. Optimized Colored Nets Unfolding. In *International Conference on Formal Methods for Networked and Distributed Systems (FORTE'06)*, pages 339–355, Paris, France, September 2006. IFIP.

[KP04] L. Kristensen and L. Petrucci. An approach to distributed state space exploration for coloured Petri nets. In *Proc. 25th Int. Conf. Application and Theory of Petri Nets (ICATPN'2004), Bologna, Italy, June 2004*, volume 3099 of *Proc. 25th Int. Conf. Application and Theory of Petri Nets (ICATPN'2004), Bologna, Italy, June 2004*, pages 474–483. Springer, June 2004.

[LL01] C. Lakos and G. Lewis. Incremental state space construction of coloured Petri nets. In *Proc. 22nd Int. Conf. Application and Theory of Petri Nets (ICATPN'01), Newcastle, UK, June 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 263–282. Springer, 2001.

[LP04] C. Lakos and L. Petrucci. Modular analysis of systems composed of semiautonomous subsystems. In *Proc. 4th Int. Conf. on Application of Concurrency to System Design (ACSD'04), Hamilton, Canada, June 2004*, pages 185–194. IEEE Comp. Soc. Press, June 2004.

[LS99] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of *Lecture Notes in Computer Science*. Springer, 1999.

[phi] *Dining philosophers problem*. `http://en.wikipedia.org/wiki/Dining_philosophers_problem`.

[smv] *The SMV System*. `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[tin] *TINA, TIme Petri Net Analyzer*. `http://www.laas.fr/tina`.

[TM04] Y. Thierry-Mieg. *Techniques for the model checking of high-level specifications*. PhD thesis, Université P. & M. Curie, 2004.

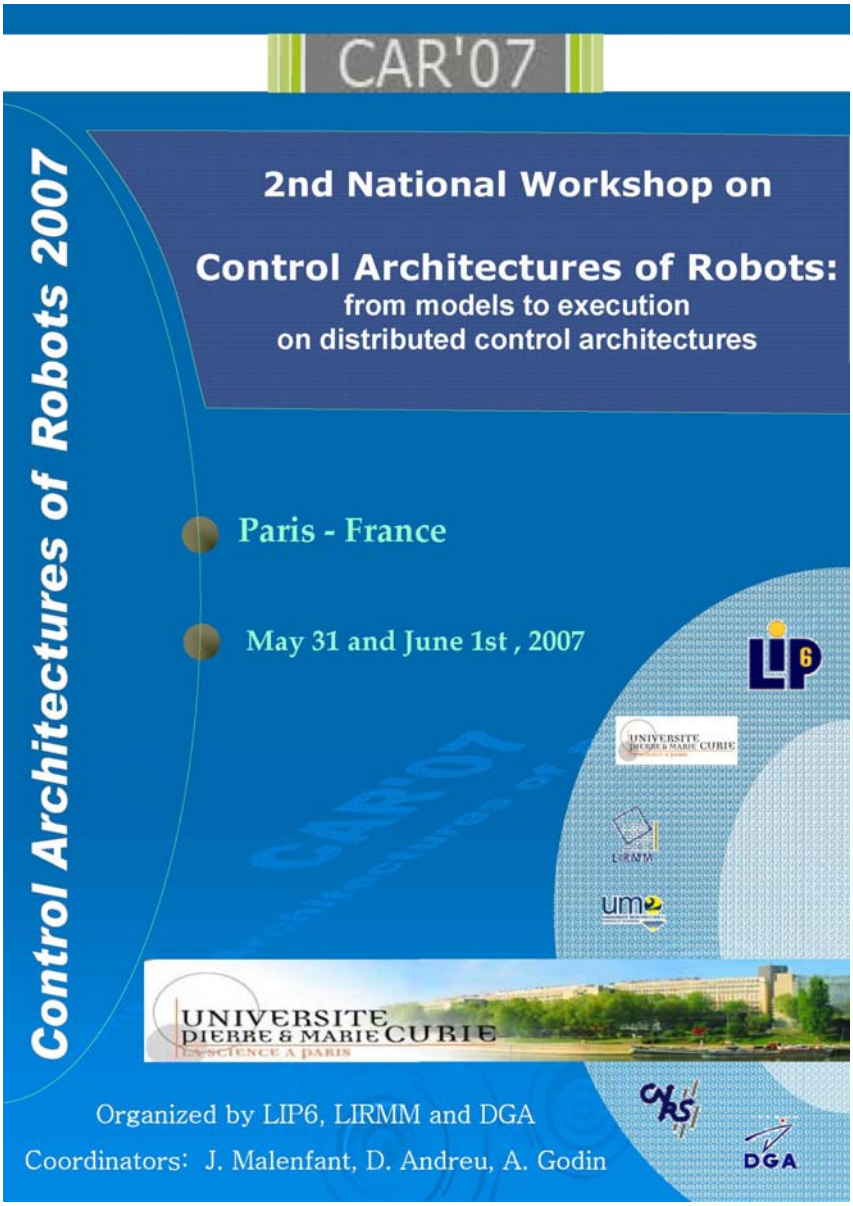
[TMDM03] Y. Thierry-Mieg, C. Dutheillet, and I. Mounier. Automatic symmetry detection in well-formed nets. In *Proc. of ICATPN 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 82–101. Springer, June 2003.

[TMIP04] Y. Thierry-Mieg, J-M. Ilié, and D. Poitrenaud. A symbolic symbolic state space representation. In *24th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'04)*, volume 3235 of *Lecture Notes in Computer Science*, pages 276–291. Springer, July 2004.

[upp] *UPPAAL home page*. `http://www.uppaal.com/`.

Session

« Implementation of control architectures »



An adaptive MP-SoC Architecture for embedded Systems

Gilles Sassatelli, Nicolas Saint-Jean, Pascal Benoit, Lionel Torres, Michel Robert
LIRMM, UMR 5506, University of Montpellier 2-CNRS, 161 rue Ada, 34392 Montpellier Cedex 5, France
Tel.: (33)(0)4-67-41-85-69-69
Email: {first_name.last_name@lirmm.fr}

Abstract. *Scalability of architecture, programming model and task control management will be a major challenge for MP-SOC designs in the coming years. The contribution presented in this paper is HS-Scale, a hardware/software framework to study, define and experiment scalable solutions for next generation MP-SOC. Our architecture, H-Scale, is a homogeneous MP-SOC based on RISC processors, distributed memories and an asynchronous network on chip. S-Scale is a multi-threaded sequential programming model with dedicated communication primitives handled at run-time by a simple Operating System we developed. The hardware validations and experiments on applications such as MJPEG and FIR filters demonstrate the scalability of our approach and draws interesting perspectives for distributed strategies of task control management.*

1. Introduction

"MP-SOC is not just coming: it has arrived"[1]. The OMAP platform [2] with the recent 3430[2] system is one of many existing designs to date. If MP-SOC [3] is a reality, the increasing number of general purpose or dedicated processors on a single chip brings several issues to address: architecture scalability, programming models, task control management and debug are those referenced in [1].

IP core reuse has driven industrial system designers for obvious productivity and performance reasons. One major drawback is that these solutions are poorly scalable in terms of software and hardware. Although being aware of these economic constraints, we strongly believe that an alternative is possible from a basis of a scalable hardware and software framework.

The work presented in this paper aims at exploring and defining principles which grant both hardware and software scalability. The H-Scale architecture is based on a NPU (Network Processing Unit), which is essentially a programmable RISC processor, a small memory and a routing unit. The communication infrastructure is based on an asynchronous packet switching network-on-chip which allows connecting the NPUs in a mesh-based fashion. The S-Scale is a multi-threaded procedural programming model with communication primitives. Implementations carried out show that the HS-Scale framework guarantees any application to be executed regardless the target platform features (number of NPUs) and the chosen mapping. Moreover, several experiments conducted on thread duplications suggest some strategies to automate the task control management and distribute it over the system.

In the following, Section 2 tackles the issues at stakes regarding MP-SOC design and programming. Section 3 presents and details the intrinsic hardware principles. Section 4 is devoted to the programming model and more generally the software part of the framework which includes the online management mechanisms. Section 5 presents the hardware realizations, particularly the FPGA prototype and its debugging interface and provides some performance figures of this realization. Section 6 gives application results on a FIR filter and a MJPEG decoder.

2. Related works

During the last decades, improvements in microprocessor design and compilers were mainly aimed at improving Instruction Level Parallelism. It has nevertheless been stated that trying to further increase ILP is not the best choice, D. Patterson refers it to as the "ILP wall" [6]. Thread (or Task[3]) Level Parallelism (TLP) enables significant speedups and proves more flexible than ILP. TLP is now supported by a growing spectrum of programming environments through programming models, libraries, etc.

From an architecture point of view, a MP-SOC may be either homogeneous, *i.e.* all the processing elements are the same (*e.g.* for server applications), or heterogeneous (CMP, or Chip Multi Processing, *e.g.* for embedded applications). One typical example of a heterogeneous MP-SOC system is a cell phone (for instance, those based on an OMAP platform). One of the toughest aspects in heterogeneous MP-SOC is that software modules have to interrelate with hardware modules. For instance, the authors of [4] advocate the use of high level programming for the abstraction of HW-SW interfaces. Their programming model is made of a set of functions (implicit and/or explicit primitives) that can be used by the SW to interact with HW. In the reconfigurable computing domain, alternative approaches have also been investigated, as the original one in [5] where a scalable programming model (SCORE) is associated to a homogeneous scalable reconfigurable architecture. The model allows indifferently computing a set of tasks in time or in space, following the resources available: the advantage is that software is reusable for any generation of component based on that model.

There are two major programming models deriving from the memory architecture: SMP (Symmetric Multi Processing) where all the processors have a global vision of the memory (shared memory) and AMP (Asymmetric Multi Processing) where the processors are loosely coupled and have generally dedicated local memory resources. Procedural sequential programming (*e.g.* C) is generally the basis of MP-SOC systems as it stands on compilers that are widely available and because "Everybody knows C…". As MP-SOC provides resources to compute several tasks concurrently, multi-threaded programming models have to be examined. With multi-core processor architectures, libraries such as open MP (SMP model) and MPI (Message Passing Interface) (AMP model) provide an interface to the

[1] Grant Martin, Design Automation Conference, 2006 [1]
[2] Superscalar ARM CortexA8 core, IVA2+ accelerator for H264 video, Image Signal Processor, 2D/3D Graphics accelerator

[3] *Thread* and *task* are interchangeable within the scope of the presented work

programmer for execution directives inside the source code. Using this kind of library is currently not realistic for MP-SOC designs for the overhead they imply.

Task control management is also an issue to consider. Threads can be handled at execution time (dynamically on a single processor) by an operating system or at design time (statically) by complex scheduling techniques. A part of the MP-SOC community focuses on static task placement and scheduling in MP-SOC. Indeed, having a complex operating system in memory taking care of run-time mapping is often not feasible for a SOC, because of the restricted memory resources and associated performance overhead. Moreover, these systems are often heterogeneous and dedicated to a few tasks, and a single but efficient scheduling of tasks may be more adapted. For instance in [7], the authors summarize the existing techniques (ILP based or heuristics) and have developed a new framework based on ILP solvers and constraint programming to solve at design time the task allocation/scheduling problem. There are also contributions for the improvement of local performances in [8] and for energy savings in [9].

3. H-Scale architecture model

3.1. System overview

One of the leading principles of our approach relies in the exploration of massive parallelism for embedded MP-SOC; we target scalability of both hardware and software and expect performance to emerge from multitude and not from the intrinsic performance of the processing tile. For that reason, the main concerns regarding the processing element architecture are flexibility and compactness.

Figure 1 shows a system-level overview of the H-Scale architecture and the surrounding components it is supposed to be connected to. As illustrated in this figure, our contribution is a homogeneous MP-SOC as a component of a heterogeneous system. It is based on a scalable architecture with distributed memory (AMP model). It is made of a regular arrangement of processing elements (PE) interconnected by a packet-switching communication network. As we assume that the HS-Scale architecture is a component of a realistic system, some of the PE of this architecture is responsible of establishing the communications with the rest of the system (interface PEs).

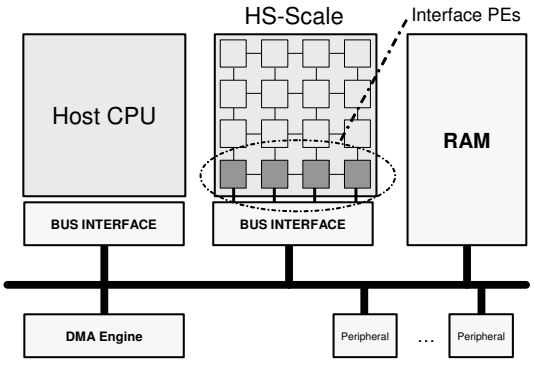


Figure 1: system-level overview

3.2. Network Processing Unit

The architecture we present is made of a homogeneous array of PE communicating through a packet-switching network. For this reason, the PE is called NPU, for Network Processing Unit. Each PE, as detailed later, has multitasking capabilities which enable time-sliced execution of multiple tasks. This is implemented thanks to a tiny preemptive multitasking Operating System[4] which runs on each NPU. The structure of the NPU is depicted in figure 2. It is built around two main layers, the network layer and the processing layer.

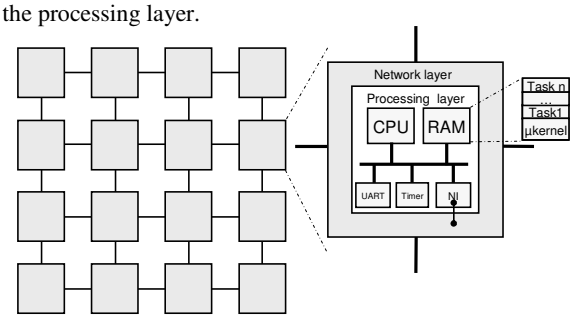


Figure 2. Network Processing Unit

The Network layer is essentially a small routing engine (XY routing). Packets are taken from incoming ports, then either forwarded to outgoing ports or passed to the processing layer. It is compliant with the communication infrastructure presented below. When a packet header (the first flit) specifies the current NPU address, the packet is forwarded to the network interface (NI in figure 2). The network interface buffers incoming data in a small hardware FIFO and simultaneously triggers an interrupt to the processing layer.

The processing layer is based on a simple and compact RISC microprocessor, its static memory (no cache) and a few peripherals (timers, one interrupt controller, UART) as shown in figure 2. A multitasking OS implements the support for time-multiplexed execution of multiple tasks. The microprocessor we use has a compact instruction set comparable to a MIPS-1 [10]. It has 3 pipelines stages, no cache, no Memory Management Unit (MMU) no memory protection support in order to keep it as small as possible.

3. 3. Communication infrastructure

For technology-related concerns, a regular arrangement of processing elements (PEs) with only neighboring connections is favored. This helps in a) preventing using any long lines and their associated undesirable cross-talk effects in deep sub-micron CMOS technologies b) synthesizing the clock distribution network since an asynchronous communication protocol between the PEs might be used. Also, from a communication point of view, the total aggregated bandwidth of the architecture should increase proportionally with the numbers of PEs it possesses, which is granted by the principle of abstracting the communications through routing data in space. The Network-on-Chip paradigm (NoC) enables that easily thanks to packet switching and adaptive routing.

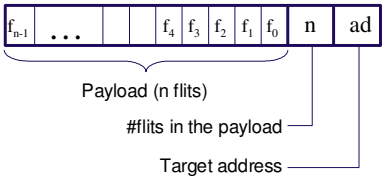


Figure 3: Packet format

The communication framework of H-scale is derived from the Hermes Network-on-chip, refer to [11] for more details.

[4] Operating System are often refered to as microkernels in the area of SOC, mainly because of their very limited memory footprint

The routing is of wormhole type, which means that a packet is made of an arbitrary number of flits which all follow the route taken by the first one which specifies the destination address. Figure 3 depicts the simple packet format used by the network framework constituted by the array of processing elements. Incoming flits are buffered in input buffers (one per port). Arbitration follows a round-robin policy giving alternatively priority to incoming ports. Once access to an output port is granted, the input buffer sends the buffered flits until the entire packet is transmitted.

Inter-NPU communications are fully asynchronous, and are based on the toggle-protocol. As depicted in figure 4, this protocol uses two toggle signals for the synchronization, a given data being considered as valid when a toggle is detected. When the data is latched, another toggle is sent back to the sender to notify the acceptance. This solution allows using completely unrelated clocks on each PE in the architecture.

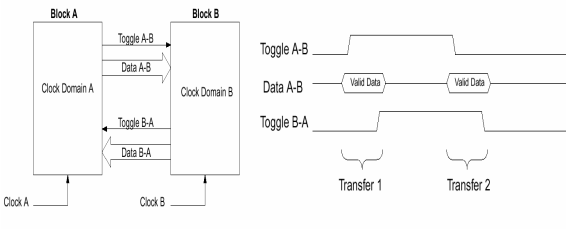


Figure 4: The asynchronous toggle protocol

4. S-Scale programming model

Our goal is to provide a complete scalable solution which assumes first that the architectural model is scalable, but also the programming model. Our model is based on distributed memories (AMP) and allows computing multiple tasks in time (single processor), and in space (multi-processor). For a given application any possible mapping scenario between computing in time and computing in space is supported.

4.1 Multi-Threaded Procedural model, with Communication Primitives

S-scale is a mixed model composed of a Sequential Procedural Programming basis, a Multi-threaded support and Communication Primitives for inter thread communications.

A process is an instance of a program in memory. It consists generally of several functions. When these functions may be scheduled separately, they are called threads. On multiprocessor machines, it is more natural to program applications with multiple threads since they have the possibility to be executed on several processors. The threads in our model are described in C language, the most famous and widely used sequential and procedural language for programming embedded systems.

Since threads may be time-sliced, which means they can run in arbitrary bursts as directed by the operating system, the property of confluence (same result yielded regardless thread execution order) must be guaranteed. The underlying programming style for ensuring the synchronization of the computation in our approach is Kahn Process Networks (KPN) [12]. KPN is a distributed model of computation where processes are connected to each other by unbounded FIFO channels to form a network of processes. KPN can be represented functionally by a Petri net as depicted figure 5.

Reading from a channel is blocking: the single token in the place *P* forbids that the process is executed before the place FIFO IN is filled with data. Writing is non-blocking: when the data has been written to the FIFO OUT, place *P* is filled with its initial marking again allowing new data to be read. A set of communication primitives has been derived from this formalism for ensuring confluence of application execution regardless thread execution order.

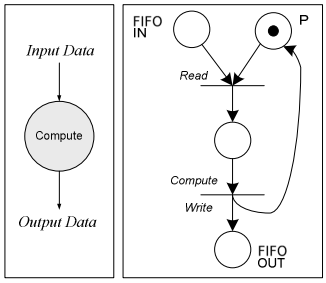


Figure 5. KPN Model of a single task computation

4.2 Communication primitives

They essentially abstract communications so that tasks can communicate with each other without knowing their position on the system (either on the same NPU or a different one). The communication primitives were derived from 5 of the 7 layers of the OSI model as shown on figure 6.

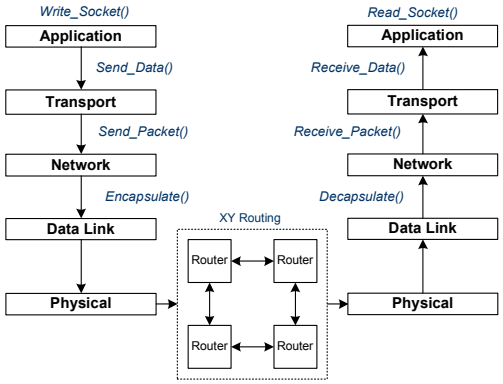


Figure 6. Communication Protocol and Communication Functions on 5 OSI layers

Firstly, communication management between tasks is insured by two dedicated functions. In order to route the packets, these functions use a dynamically updated routing table. *Read_Socket()* and *Write_Socket()* read and write to software FIFO supervised by the operating system. These functions allow transparent data communications between tasks either locally or remotely: the routing is done following this dynamic routing table. When the task is local, the writing of data is done on a local software FIFO. When the task is remote, the operating system must insure that there is enough space for the remote software FIFO to avoid deadlocks on the network. This is done thanks to dedicated functions. As soon as the OS gets a positive answer, he can start encapsulating and sending the data packets to the remote task (*Encapsulate()*, *Send_Data()*) while the remote task can deencapsulate and receive the data packets and write them to its local software FIFO (*Decapsulate()*, *Receive_Data()*).

4.3 Operating System

In order to schedule tasks on a single processor, to handle communications between local and remote tasks with the communication primitives described above, it is necessary to use an Operating System offering these functionalities. After checking the literature and existing embedded OS (uClinux, eCos, etc.), it appeared that our memory restrictions (less than 100kB for data and program on one NPU) were too strong to use these costly solutions.

Therefore, we have developed a lightweight operating system which was designed for our specific needs. Despite being small, this OS does preemptive switching between tasks and also provides them with the communication support for tasks interactions (communication primitives). Figure 7 gives an overview of the operating system infrastructure and the services it provides.

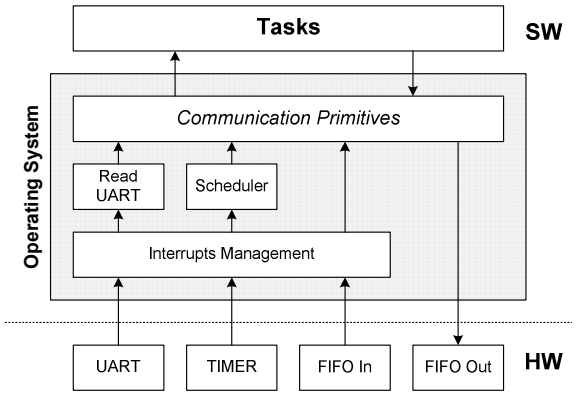


Figure 7: Operating System overview

The interrupts manager may receive interrupts from hardware: UART, Timer and FIFO In. When this happens, it disables the interrupts and save the processor context. Following the type of interruption, it reads from UART, schedules the tasks (timer) or use a communication primitive (interrupt from the FIFO). Afterwards, it restores the processor context and enables again the interrupts. The scheduler is the core of the OS but is quite simple. Each time a timer interrupt occurs, it checks if there is a new task to run. In the positive case, it executes this new task. Else, it has two possibilities: either there is no task to schedule then just runs an *idle* task, or there is at least one task to schedule. This way, each task is scheduled periodically in a round robin fashion (there is no priority management between tasks).

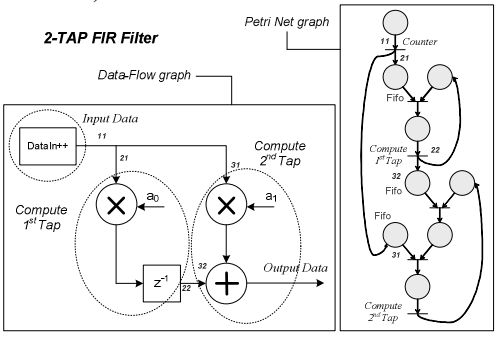


Figure 8: FIR example

4.4 Programming Example

In the following, we give a very simple example on how to program our architecture with the proposed programming model. It shows how from a classical C code of a 2-TAP FIR filter we introduce thread directives based on our communication primitives. This is just an illustration of what a programmer could do with this model and would not make sense "in the real world" to improve the performances of such an algorithm. However, it demonstrates the scalability of the programming model.

The figure 8 illustrates the filter with two representations: the first one is a simple data-flow graph and the other one is a KPN process networks with Petri net.

On a classical processor, the C-code could look like that:

```
int main()
{
  int data_in,r1,r2,r1p=0,a0=1,a1=2,data_out;
  data_in=0;
  while(1)
    {
        r1=data_in*a0;  // Compute 1st tap
        r2=data_in*a1;  // Compute 2nd tap
        data_out = r2 + r1p;
        r1p = r1; //Delay
        data_in++; // data increment
    }
  return 0;
}
```

With our programming model, it is possible to fork the process in 3 different threads as shown below:

```
Type_task thread1(void)
{
  int data_in=0;
  while(1)
    {
        write_socket(21, &data_in, 1, 1);
        write_socket(31, &data_in, 1, 1);

        data_in++; // data increment
  }
  return 0;
}

Type_task thread2(void)
{
  int data_in, r1 ,a0 = 1, zero=0;
  /*Data synchronization*/
  write_socket(31, &zero, 1, 1);
  while(1)
    {
        read_socket(21,&data_in,1,1);
        r1 = data_in * a0; // Compute 1st tap
         write_socket(32, &r1, 1, 1);
    }
  return 0;
}

Type_task thread3(void)
{
  int data_in, r1 ,r2 ,a1 = 2, data_out;
  while(1)
    {
        read_socket(31, &data_in, 1, 1);
        read_socket(32, &r1, 1, 1);
        r2 = data_in * a1; // Compute 2nd tap
        data_out = r2 + r1;
    }
  return 0;
}
```

One can notice that the functions *Write_socket()* and *Read_socket()* are used to establish communication channels between the three tasks. The parameters represent the socket identifiers (21, 31, …) which are used in the routing tables of the NPU, the address of the data block and the number of data in the block.

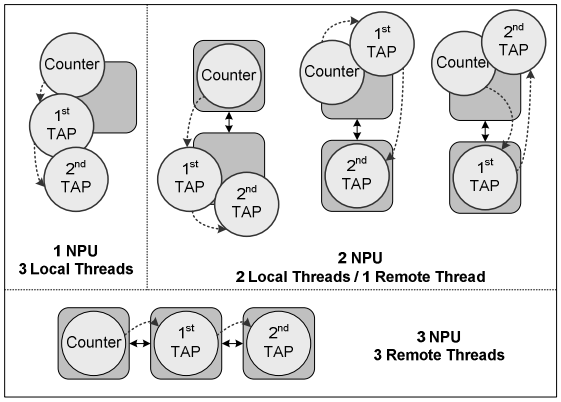


Figure 9: 3 functionally equivalent mappings on 1, 2 or 3 NPUs, with the same C program

By the way, this program can be mapped (statically) to the architecture on a single or multiple NPU indifferently as depicted figure 9, *i.e.* with the same functionality since the communication primitives of the OS ensure task interactions.

5. Validations

5.1. Hardware Prototype

A complete synthesizable RTL level description of the NPU has been developed. It has allowed us to validate the hardware prototype, estimate areas and power consumptions (post place and route, with AMS 0.35µ design kit), and improve the design. Any instance of the H-Scale MP-SOC system may be easily generated with our generic parameters, and then evaluated with CAD Tools (Encounter Cadence flow).

# NPU	*1*	*2*	*4 (2*2)*	*9 (3*3)*
Area (mm²)	18.22	36.63	73.61	165.30
Power Cons. (mW/MHz)	2.56	5.14	10.34	23.26

Table 1: Area and Power consumption[5] scalability

Table 1 summarizes these evaluations. The hardware prototype has been placed and routed with a 64KB local memory, which actually represents in a single NPU 87% of the total area. In the 13% remaining, the Processor represents 54% (1.2 mm²), the router 38% (0.85 mm²) and the rest (UART, interrupt controller, Network Interface, etc.) about 7%. The power consumption has been evaluated thanks to simulation database dump (vcd files) and Cadence tools. It has been then optimised with Gated Clock insertion. The power consumption repartition figure is slightly the same compared to the area. The table above clearly shows the scalability of area and power consumption of our H-Scale System (the very low overhead is due to the wires needed to interconnect the NPU).

5.2. FPGA Prototype

RTL Simulations are too slow for significant applications such as MJPEG performed on streams of data. We decided then to use a Xilinx Development Kit to synthesize and validate our design. 6 NPUs could be fitted on a XC2VP30 FPGA from Xilinx. A NPU occupied 2151 slices on the FPGA which is rather small.

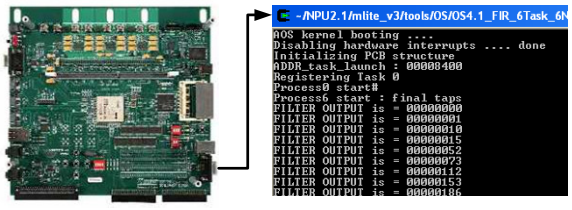


Figure 10: FPGA prototyping board and its debugging interface

Debugging on the prototype takes place thanks to a UART interface between one interfacing NPU and the workstation; some additional services were added to the NPU kernel for feeding back debugging information directly to the PC (figure 10).

5.3. Software Tool Chain

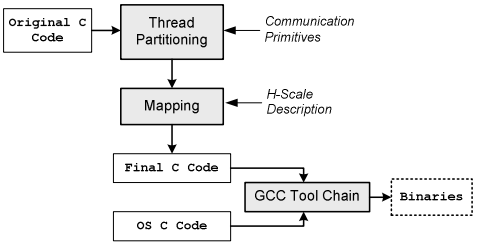


Figure 11: Software flow

Figure 11 depicts the software flow used in our framework to port an application from its original C-code to the HS-Scale MP-SOC. A thread partitioning is first done by the programmer. This can be helped by the original procedure partitions (function calls) and profiling tools. The communication primitives are then used to elaborate the communications between the threads. Then, a hand-made mapping of each thread is performed on the H-Scale instance and a routing table is derived. The final C-code is composed of each thread C-code, and then is compiled with the OS C file, allowing thus generating the binaries to load into the memories of the NPU.

OS Min. Time (cycles)	*OS Max. Time (cycles)*	*Communication Primitives (KB)*	*Total OS Size (KB)*
325	373	2.73	5.75

Table 2: Operating System Time and Memory costs

Table 2 is provided to give an overview of the overhead issued by our Operating System. In terms of time penalty, each time the OS is invoked (each time an interrupt happens), it requires between 325 and 373 cycles to perform its job. The effective time penalty regarding applications performances will be analysed in the next section. In terms of memory overhead, it requires 5.75 KB, which represents less than 10% of the 64KB memories we used in our experiments. The communication primitives represent almost half of the total memory required by our OS.

6. Application Results

6.1. FIR

In order to evaluate the overhead introduced by the OS we carried out some experiments for the FIR application. This application has a very high communication over

[5] Average power consumption performed on NPUs running the OS and several tasks

computation ratio which reveals a much too fine task granularity. Hence, both the kernel scheduler and its communication primitives are highly solicited and therefore tend to slowdown the computation. Table 3 shows the FIR performance results for different task mappings, with and without operating system. The results are given for the processing of 10.000 input samples.

	w/o OS	*With OS*		
# NPU	1	1	1	3
# Threads	1	1 Local	3 Local	3 Remote
Cycles	550634	553992	5982187	2036976
Tp (MB/s)	**7.09**	**7.05**	**0.669**	**1.92**

Table 3: Throughput (Tp) Performance of a 2-TAP FIR Filter

Comparing the results of the two first columns of Table 3 shows that both in terms of processing time and Throughput (Tp) the overhead remain below 1%. As expected, when the FIR algorithm is split into several tasks running sequentially on the same NPU, the communication overhead highly degrades the performance (column 3). Distributing the processing among several NPUs (column 4) shows the benefit of using task-level parallelism; without however matching the performance of the single task implementation.

6.2. MJPEG

In order to evaluate the performance of HS-Scale for realistic applications, we have implemented a MJPEG decoder. We naturally chose to use a traditional task partitioning as depicted in figure 12 with both a task-level dataflow description and a functional Petri net equivalent. The first step of the processing is the inverse variable length coding (IVLC) which relies on a Huffman decoder. This processing time for that task is data dependent. The two last tasks of the processing pipelines are respectively the inverse quantization (IQ) and the inverse discrete cosine transforms (IDCT). The atomic data transmitted from task to task is a 8x8 pixel block which has a size of 256 bit.

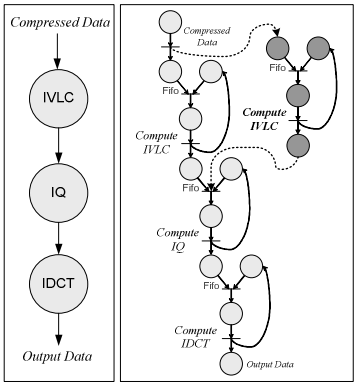


Figure 12: MJPEG Data-Flow and Petri Net Representation

a. From Simple pipeline implementation to multi-threads

Table 4 summarizes the performance figures obtained for several implementations of the MJPEG decoder. Similarly to the FIR implementation, the operating system communication primitives induce a performance overhead when the decoder is splitted into 3 tasks (Table 4, column 2). Distributing the processing on 2 NPUs (Table 4, column 4) immediately pays nevertheless with a significant increase in the throughput. The fully distributed implementation exhibits no performance improvement, which is due to the fact that the critical task in the processing pipeline already fully employs the processing resources of a given NPU.

	w/o OS	*With OS*			
# NPU	1	1	1	2	3
# Threads	1	1 Local	3 Locals	2 Locals, 1 Remote	3 Remotes
Tp (KB/s)	**229**	**228**	**161**	**244**	**246**

Table 4: MJPEG Throughputs (Tp) comparisons

b. From multi-threads, to thread Replication

Many applications such as dataflow applications present tasks that exhibit different and potentially time-changing computational loads over time. Data compression algorithms for instance always feature a variable-length coding task that can be very demanding in performance depending on processed data. In such scenarios, allocating hardware resources at run-time may help better meeting performance requirements without the traditional over-dimensioning problem of static allocation. The principle developed in this section relies in a multi-graph description of the same application; the processors are then responsible to switch from one graph to another depending on run-time requirements.

Figure 13.a depicts a synthetic task graph. A profiling may show that task2 is (i) the most demanding and (ii) exhibit data-dependent computational load. Replicating it helps in increasing the performance which would lead to the scenarios depicted on Figure 13.b and Figure 13.c. In such cases, of course all three instances of task 2 would be hosted on a dedicated processor. The experiments conducted implement the automated replication strategy based on a multi-graph description of the application. Strategies enabling run-time replication may either be simple (fork() and join() in this case) or more difficult, therefore requiring programmer attention.

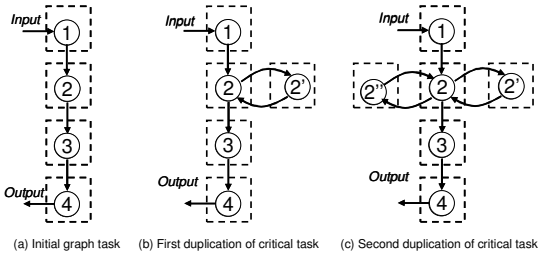


Figure 13: Initial application task graph(a) and replication of task 2 (b and c).

In the case of the MJPEG application the replication is permitted by the absence of inter-block dependencies in the MJPEG application Table 5 shows the performances achieved for replications of some tasks. As clearly suggested by the throughput values, the IVLC task is the critical one in our case since duplicating and triplicating it (columns 5 and 6) significantly increase the performance. Allocating a fourth NPU to this task does not further improve the performance meaning that another task then became the critical step in the processing pipeline.

	w/o Duplic.	*IQ Duplic.*	*IDCT Duplic.*	*IVLC Duplic.*	*IVLC Triplic.*
# NPU	3	4	4	4	5
# Threads	3 Remote	4 Remote	4 Remote	4 Remote	5 Remote
Tp (KB/s)	**246**	**220**	**241**	**332**	**432**

Table 5: MJPEG comparisons with or without thread duplications

c. Load balancing

As mentioned previously, many applications feature highly asymmetric computational load for their constituting tasks. Similarly to the process explained above, where demanding tasks are replicated, we have statically observed the potential benefits of merging several sub-critical tasks onto the same processor. This results in time-sliced execution of those tasks. Figure 14 schematically explains that principle, where task2 fully exploits the processing resource of a given processor (since it has been identified as critical) and task3 and task4 are executed onto a single processor.

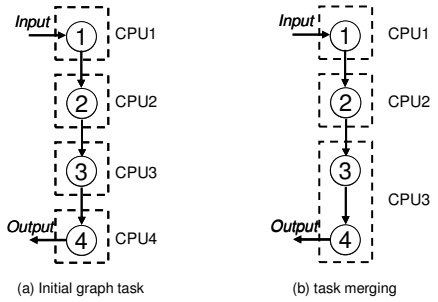


Figure 14 – Principle of load balancing.

Implementing such a mechanism at run-time implies to migrate tasks; which translates in a performance over-head directly proportional to the task code and state to migrate.

The testbench used in this case is similar in every respect to the previous one but a single additional FIR task was moved from NPU to NPU. Table 6 shows the corresponding results, where one can see how much the performance of each application is affected depending on the task mapping. Assigning the FIR filter to the NPU hosting the critical IVLC task (column 3) yields to a significant slowdown of both applications (40% for the FIR, 46% for the MJPEG). Contrarily, assigning the FIR task to one of the other NPUs hosting the sub-critical tasks marginally affects the performance of both applications (columns 3 and 4). The FIR was also assigned for IVLC-duplicated versions of the MJPEG decoder; likewise a similar slowdown is observed only when the FIR is assigned to one of the NPU hosting a critical task.

# NPU	*1*	*3*	*3*	*3*
# Threads FIR	1 Local	1 with IVLC	1 with Iquant	1 with IDCT
Tp (MB/s)	**3.8**	**4.0**	**5.7**	**6.0**
# Threads MJPEG	1 Local	3 Remote	3 Remote	3 Remote
Tp (KB/s)	**122**	**134**	**245**	**246**

Table 6: MJPEG and FIR simultaneous execution performance results

7. Conclusion and Perspectives

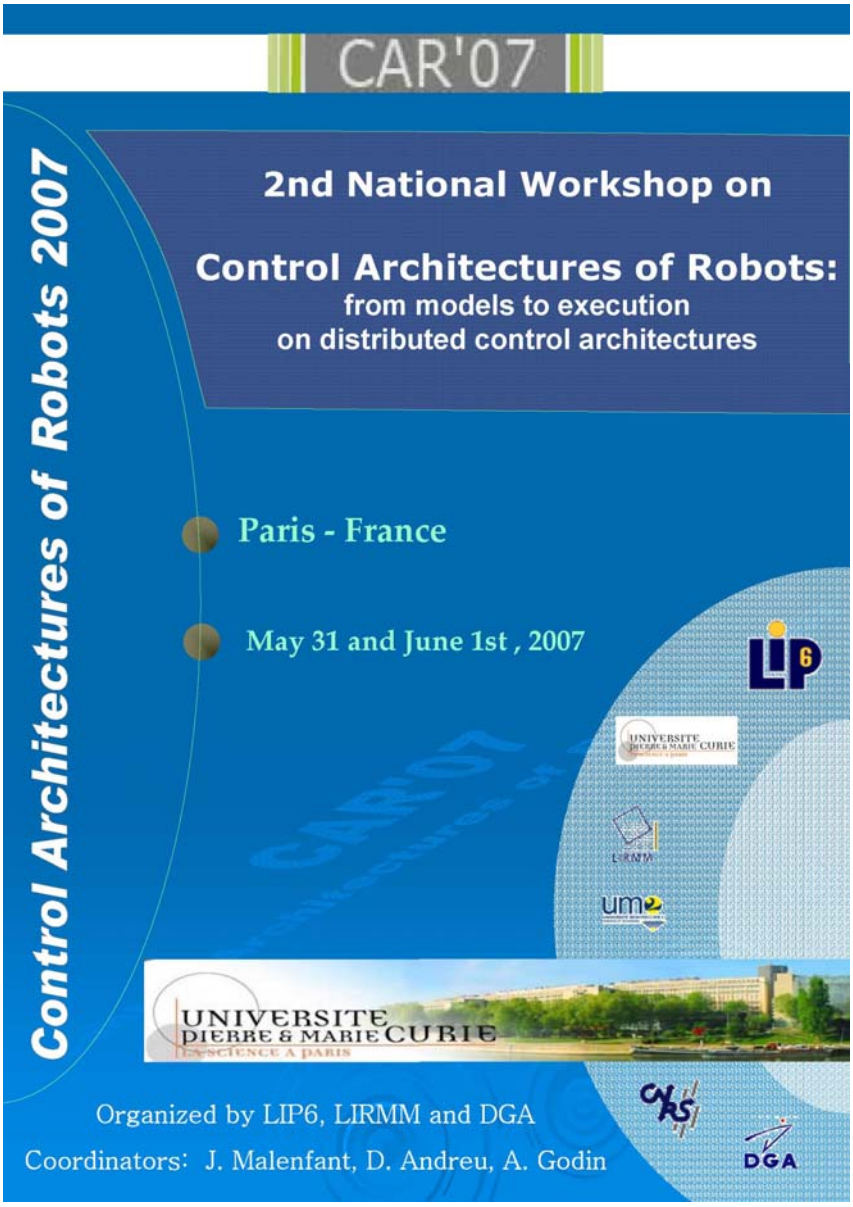
A scalable hardware and software framework has been proposed and detailed throughout this paper. Based on a regular arrangement of homogeneous processing units endowed with multitasking capabilities our architecture is capable to support an almost unlimited number of task mapping combinations. The tiny and efficient OS combined to the packet-switching communication architecture we use gives the programmers a huge flexibility at reasonable cost. We have highlighted through some examples that in the case of multiple applications, some mappings may allow to map additional applications at almost no cost.
Regarding the performances obtained in the results section, our current work aims now at extending the OS functionalities to automate the task mapping, migration and duplication (dynamic and continuous task mapping) for achieving run-time adaptability.

8. References

[1] Grant Martin "Overview of the MPSoC Design Challenge**",** Proceedings of the 43rd annual conference on Design automation, San Francisco, USA, 2006
[2] OMAP, Texas Instrument Technology, http://www.omap.com
[3] Ahmed A. Jerraya and Wayne Wolf (editors), M*ultiprocessor Systems-on-Chip*, Elsevier Morgan Kaufmann, San Francisco, California, 2005
[4] Ahmed A. Jerraya, Aimen Bouchhima, Frédéric Pétrot, "Programming Models and HW-SW Interfaces Abstraction for Multi-Processor SoC", Proceedings of the 43rd annual conference on Design automation, San Francisco, USA, 2006
[5] Caspi E., Chu M., Huang R., Weaver N., Yeh J., Wawrzynek J., and A. DeHon, "Stream Computations Organized for Reconfigurable Execution (SCORE)", *FPL'2000*, LNCS 1896, pp. 605-614, 2000
[6] David A. Patterson, "Future of Computer Architecture", Berkeley EECS Annual Research Symposium (BEARS), College of Engineering, UC Berkeley, US, February 23, 2006
[7] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti and M.Milano, "Communication-Aware Allocation and Scheduling Framework for Stream-Oriented Multi-Processor Systems-on-Chip", IEEE Design Automation and Test inEurope, Munich, Germany, March 2006
[8] M.T. Kandemir, G. Chen, "'Locality-Aware Process Scheduling for Embedded MPSoCs,'" Proceedings of DATE, pp. 870–875, 2005
[9] Hu J., Marculescu R., "Energy-Aware Communication and Task Scheduling for Network-on-Chip Architectures under Real-Time Constraints." DATE 04, 234-239
[10] MIPS corp., http://www.mips.com
[11] Moraes, F. G.; Mello, A. V. de; Möller, L. H.; Ost, L.; Calazans, N. L. V.. "HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip.", Elsevier Integration, The VLSI journal / Special issue: Networks on chip and reconfigurable fabrics, Vol. 38, Number 1, pp 69-93; Oct. 2004.
[12] G. Kahn, "The semantics of a simple language for parallel programming", *Information Processing*, pages 471-475, 1974

Session

« Languages and MDA »



A Modeling Language for Communicating Architectural Solutions in the Domain of Robot Control

Robin Passama

OBASCO Group, École des Mines de Nantes – INRIA, LINA
4 rue Alfred Kastler, 44307 Nantes cedex 3, France
Robin.Passama@emn.fr

Abstract. An open issue in robotics is to explain and compare very different solutions for control decomposition proposed by robot architects. This paper presents a domain specific modelling language dedicated to overcome this problem. The underlying goal of this work is to promote the communication of robot architects' expertise. The paper starts from a reflexion on the state of the art in research on control architectures. It results from this the report that to compare control architectures, it is necessary to be focused on the decision process organization by abstracting from technological features. A conceptual model for robotic architectures, defining domain terminology and concepts used in the description of decision process organization, is presented. Then, the language used to model organization of the decision process is defined via a meta-model. Finally, the use of the language is illustrated with the description of architectural solution of Aura.
keywords: robot control architecture, domain specific modelling language, decision process decomposition

1. Introduction

According to [10], a robot is "a machine that physically interacts with its environment to reach an objective assigned to it. It is a polyvalent entity able to adapt itself to variations of its working conditions. It has perception, decision and action functionalities [...] It also has, at various level, the ability to cooperate with man." A robot can be decomposed into two part (1) an operative - or mechanical - part including physical elements, sensors, actuators, etc, (2) a controller part in which decision are taken and reactions computed. The controller part is a complex composition of software and hardware pieces putting in place robot decisional process.

The design of controller is a very complex task. One important factor of this complexity is that design process involves a great variety of expertises in automatics, informatics, telecommunications, electronics, mechanics, etc. As intelligent robotics becomes an industrial challenge, the need of powerful methodologies, languages and tools now arises as a very important issue to reduce the complexity of the design of robot controllers. In this frame, the notion of *architecture*, as an artifact that describes a modular decomposition and main properties of the controller of a given robot, is certainly becoming as important as it is today in software engineering. An *architectural solution for robot control*, or *control architecture solution*, is a solution for decomposing design of controllers in a more or less generic way: the same solution can be used and specialized for the design of more than one robot controller, in a more or less restrictive way according to a given set of operational requirements associated to this solution.

To date, expressing architectural solutions is always made "by hand" in most of research or industrial papers, without formalization or standardisation, unlike it is done in software engineering with the used of UML [14]. Without a common language for designing robot control architectures , it so very difficult to understand, to communicate and to compare the different solutions. It is also a barrier for the adoption of robot control design patterns as solutions of specific robot control problems, like design pattern [12] are used in software engineering to express reusable software design solutions. UML could be used to design control architectures, but it does not support any domain abstractions, which leads each robot control designer to redefine new abstractions and terminology. The consequences would be that (1) concepts and terminology would not be compatible, making human understanding and solutions comparison very difficult, (2) users would reinvent the wheel for each of their design, wasting a considerable amount of time, (3) graphical representations of models would not emphasis the characteristics of domain expert "way of doing", making the communication of models more difficult. Furthermore, UML class and component diagrams, focus on software aspects. But a controller is an hybrid software/hardware (electronic) system and a same control architecture solution can be implemented in very different ways depending on the responsibilities granted to software and hardware parts.

That is why a domain specific modeling language for designing control architecture solutions is required. This paper presents a first attempt to define such a language. The difficulties to overcome are (1) the complexity and diversity of robot software controllers from both a structural and a behavioral point of view that makes it complex to find the right abstractions, (2) the difficulty to find the right separation of concerns in order to improve human understanding

and expertise reuse, (3) the need to bridge the gap imposed by cultural and historical differences of practices in the community.

Section 2 presents a global reflexion on state of the art in control architecture design and make a synthesis of important concepts and practices. Section 3 starts from this synthesis and defines the proposed modeling language with a meta-model. Section 4 then illustrates the use of the language on Aura Architectural solution. Finally, section 5 concludes this paper and provides some perspectives.

2. Reflexion on the State of the Art in Robot Control Architectures

2.1. Challenge, Problem and Direction

Currently, the global challenge in robotics development is to express very different human expertise into identified software (and even hardware pieces) and to integrate all theses pieces in a cohesive manner into control architecture. This integration is not limited to expertise in robotic fields as control, navigation, vision, world modeling or artificial intelligence, but also concerns various ”non-functional” domains as security, transactions, persistence and so on. Moreover, the emergence of service robotics will involve the necessity to express a large set of ”business” domains, like medicine, human security, defense, etc. Managing each of independently already requires a high degree of expertise, but the emerging robotic industry will require managing them simultaneously for a given system. This challenge can be related to two complementary problems : *domain engineering* for the management of domain expertise and *separation of concerns* for expertise integration.

Historically, software engineering aims at providing solutions for software analysis, design, development, deployment and testing. But generally these solutions are centered on programmer's, software architect's or software project manager's points of view, like for example in object, aspect, or component paradigms. Even if software frameworks encapsulate specific domain expertises, they only provide a solution usable by programmers, but not by most of domain experts. Consequently, we see two main problems in the use of well-known software engineering approach: they don't provide to domain experts an adequate frame to apply their expertise to a given problem nor to reuse their solutions; most of them don't provide techniques at an adequate domain abstraction level to merge solutions from different domains of expertise into a global software system. In the frame of robotic systems design and development, this issue is a really important one since robot control architecture designers cannot all be software engineers.

The management of domain expertise at an adequate level of abstraction is the intended goals of the Domain Specific Languages (DSLs) approach [20]. The main idea of this pragmatic approach is to provide high-level languages to domain experts to describe solutions of domain problems. One advantage is that, thanks to the degree of abstraction of a DSL, empirical or formal rules can be checked on models in order to validate or to verify various domain properties. Another property of DSLs is sometime to allow for automatic code generation from solution models. The present paper investigates in that promising direction by defining the conceptual basis of a DSL for robot architects, but without considering analysis or code generation, only description of solutions being the subject of this work. In a DSL design, the first step, which is mandatory before being able to define language syntax and semantics that matches domain terminology, is the domain analysis. It consists to define common concepts to understand and communicate domain expertise. It corresponds, in our context, to the definition of common concepts usable by robot architects to explain control architecture solutions. To define domain concepts, next subsection gives a global overview of current methodological practices and their related issues.

2.2. Current Practices and Issues in Control Architectures

When studying current practices two main issues emerge: the diversity of control design methodologies [18] and the different levels of abstraction in the description of control architecture solutions.

This latter issue is discussed first. When looking at research papers on control architectures, one can notice that descriptions made are really different, uneasily comparable (if possible) and sometime quite ”fuzzy”. This report is certainly the first motivation of the present work. There are two factor for explaining this report. The first factor is the language or graphical conventions used to describe control architectures. Some authors use standard description languages like for instance UML class, component and deployment diagrams (for example [11]), which, as said earlier, limits the vision of the architecture to a specific implementation paradigm and does not really well captures the domain expertise. Many others use ”ad-hoc” description features, which capture domain expertise at a greater abstraction level but which are often not well defined nor comparable. The second factor is certainly the merging of implementation detail with control decomposition details in the description, which impacts greatly in the diversity of terminologies and concepts. Many proposals are close to specific frameworks. For example CLARATy [29][28] is closed to a locomotion and navigation framework, LAAS [1] and ORCCAD [25] are close to specific execution frameworks and Chimera [27], OROCOS [24] and MirpaX [16] are close to communication frameworks. With so deep differences among proposals in their technological foundations, it is obviously difficult to denote recurrent (shared) concepts in all of them.

From this first report, the intuitive solution is to provide a domain language to allow the description of key design concepts without binding these concepts with underlying technologies, with implementation languages or paradigms or even with specific algorithms. The chosen direction should then be ”abstraction”, as in software engineering few years

ago with the use of models written in a standard language [14] and of design patterns [12]. The direct consequence of this report is to differentiate the notion of robot control architecture from the notion of robot software architecture. As a first attempt to define these two notions, this paper proposes general definitions:

- The *robot control architecture* is a model of a robot controller that captures the decomposition of robot decisional process, its perception and action capabilities into interacting pieces of different levels of decisional complexity and of different responsibilities within each level.
- The *robot software architecture* is a model of the software system embedded in the robot, that defines the robot control architecture realization with software artifacts and interfaces it with robot hardware architecture.

So, regarding the previous conclusions, if the use of standard software modeling languages seems to be really useful to describe *robot software architectures* they are not adapted to describe *robot control architectures*. In the same time a language for designing *robot control architectures* is needed to avoid "ad-hoc" models and to allow for a better understanding, a better communication and a better comparison of robot architects' design solutions.

The other issue, regarding robot architects's practices, is the management of the diversity of control design methodologies. A classification of these methodologies, proposed in [18], has emerged along robotic history and defines four approaches: *reactive*, *deliberative*, *hybrid* and *behavior-based.*

Historically, the two main approaches for control architecture decomposition, defined since the 1980's, are deliberative and reactive approaches. The *deliberative* approach, also called *hierarchical* proposes a decomposition of a controller into a set of hierarchical layers, each one being a control and "decision-making system". A layer directly control the direct lower layer and is under control of its direct upper layer. The higher the layer is the more important are the decisions for the course of robot mission. The lower the layer is the more time constraints are strong in order to preserve robot reactivity (its ability to compute and apply adequate reactions in a time compatible with physical controlled system). Lower layers are responsible for simple control and reflex decisions like for example control law application loops or environment observation loops. Higher layers are involved in high-level and complex decisions such as planning. Classically, *deliberative* architectures are three layered [13], but some ones define a greater number of layers like *NASREM* [3] and *4D/RCS* [2]. Quickly speaking, the *deliberative* approach proposes a functional decomposition of robot decisional process from complex and long-term decision to simple and short-term ones. The main advantage of this approach is an interesting way to separate decisional concerns and the drawback can be a bad reactivity: the upper is the layer that makes the decision in response of a given stimuli, the higher the reaction time because information has to cross all lower layers to be handled. The *reactive* approach proposed by example in Brooks' subsumption architectures [7], proposes a decomposition of a controller into set of reactive autonomous entities, often called *reactive behaviors* because they implement behaviors of the robot specialized for a given finality (e.g. "reaching the nearest heat source"). Each reactive behavior define a *Perception - Decision - Reaction* cyclic process, where *Perception* is the mechanism that recovers sensor data, *Decision* is the mechanism that computes the adequate reaction and *Reaction* is the mechanism that apply to actuators the computed reaction. Interactions are not restricted to occur within a hierarchical schema: sensors data are simultaneously available to several reactive behaviors that decide of reactions and propagate them to actuators. The complexity is that many reactive behaviors can generate, at the same time, contradictory reactions. Reactive architecture so integrate an arbitration mechanism that allow the robot to adopt a coherent global behavior according to its mission objectives. Arbitration consists in recovering behaviors' reactions and synthesizing them in a global reaction that is actually applied to actuators. This arbitration is realized in different way, achieved by different types of interactions, for example with a complex vote protocol in *DAMN* [23] or with subsumption links in subsumption architectures [7]. The global behavior, issued from such a partially uncontrolled arbitration of behaviors is viewed as emergent. So, quickly speaking, the *reactive* approach proposes an multi-agent decomposition of robot decisional process. The main advantage is reactivity and adaptability according to physical world variations and the main drawback is an architectural and implementation complexity induced by the management of reactions arbitration.

To mix advantages of both approaches (understandability, manageability of architectures and reactivity of the resulting controller) the *hybrid* (for example *CLARAty*[29] or *LAAS* [1] architectures) and *behavioral-based* [19] approaches have been proposed. They are supposed to combine hierarchical decisional process decomposition and the ability to react quickly to environment stimuli. This is achieved in so many different ways that it is obviously complicated to list them all.

Nevertheless, a precise study of the domain shows that (1) each control architecture is based on a personal interpretation (by the authors) of the chosen approach and (2) that the distinction between all these approaches can be really fuzzy. First of all, the distinction between *deliberative* and *reactive* approaches, that seems to be clear can be attenuated by the fact that lower layer of deliberative architectures contain reactive behaviors (i.e. control law application loops) and by the fact that reactive architectures can be arranged according to a set of hierarchical layers representing different levels of decision complexity [7]. What primitively differentiate these architectures is the underlying decomposition "philosophy" (functional and multi-agent).

Another example is the difference between reactive and behavioral-based approaches that is really thin since it mainly relies on a criterion ("the behavior-based can store representations while reactive cannot" [18]) that can be considered as subjective (in fact, only the life time of the representation differs).

One more example is the difference between hybrid architectures like *Aura* [4], *3T* [5] or *ARM-GALS* [17] on the one hand and *LAAS*[1], *CLARAty* [29] on the other hand. In fact, the first ones are hybrid in the sense that their

lower layer is organized around reactive behaviors coordinated with an arbitration mechanism (like reactive or behavior-based approaches) and this layer is under the control of a decisional layer in charge of complex decisions (resulting in reconfigurations of reactive layer). The second ones are more or less organized as deliberative architectures but according to the authors with a greater uncoupling between layers which is a quite subjective criterion. Fortunately this criterion can be specified thanks to architectures, for example *ORCCAD* [6] or *LIRMM* [22] architectures, that clarify in a more or less explicit way this point: architectures are viewed as hybrids because they have a layered style that allows for direct interactions (under given conditions) between non adjacent (not directly in relation) layers, allowing so sensing or reaction information to cross frontiers between layers in order to improve reactivity. To conclude on that point, the difference between hybrid architectures is important when considering the control design within the lower layers: the former approach being based on a multi-agent decomposition, the latter being rather based on a functional one.

The final example is the difference between hybrid architectures like *IDEA* [21] or *Chimera* [26] and other hybrid architectures, that lies in the decomposition of the control architecture into control sub-systems that incorporate the control of specific parts of the controlled robotic system. This organization is viewed as hybrid because (1) each subsystem encapsulates both reactive and deliberative capabilities and so can be viewed as a hierarchically layered architecture (even if it is not explicit in papers) and (2) subsystems are independent from each other and coordinate in a complex way, being so considered as autonomous interacting agents. This organization is, in a limited way, generalized in *LIRMM* [22] but also in *CLARATy* [29] where each subsystems is in turn incorporated into a more global layered system that controls the whole robot: subsystems are then under the control of a higher decisional layer. This approach has the benefit to render more intuitive the decisional process decomposition because it couples it with the decomposition of morphological and infrastructural (hardware) attributes of the controlled sub-system.

As a conclusion for this subsection, the intuitive direction to follow for a language for communicating architectural solutions is to provide common concepts that on one hand abstract from these subtle differences between approaches and on the other hand allow all existing control design methodologies to be used.

2.3. Commonalities

Defining the right abstractions to design control architecture solutions requires in the first time to focus on commonalities between proposed architectures. Commonalities are identified in different ways. The first (and rather classical in domain analysis) way is to consider as ”common” the concepts that are the most recurrent in (since there is no concept shared by all) control architectures. The second is to consider as ”common” the concepts that are explicitly used in really few architectures but that are in fact really useful and always applicable for control architecture decomposition. The last way consists in identifying recurrent variations in control architectures and generalizing them into a concept that can capture all possible variants.

The most easy to find recurrent concept is the one of *layer*. The *layer* concept exists in a huge number of architecture, initially in *deliberative* architectures, but also in most of *hybrid*, *reactive* and *behavior-based* architectures. Unfortunately, one have to admit that nearly each author has its own vision of what is exactly a *layer*. For example, in *subsumption architectures* [7] layers are abstractions of reactive behavior roles (e.g. robot movements, robot integrity, etc.), and in *LAAS architectures* [1] they separate AI decision-making mechanisms from control law modules. What is important in the concept of layers, is that a layered organization separates decisional concerns in a clear way, whatever the precise meanings of layers are. The layer is so the vector of a hierarchical decomposition of decisional process.

Another recurrent concept is the one of *activity* or *task*. An *activity* or *task* denotes a part of the decisional process with specific decisional, control and perception responsibilities. The terms *activity* or *task* are not standards since many other terms are often use, like for example real-time task and real-time procedure [6], reactive behavior [7], genom module [1], module [9], agent [21], port-based object [26], Motor or perception Schema [4], depending on authors point of view that is influenced by implementation or methodological concerns. Sometimes *activities* are not explicit in the architecture, even if they exists, for example *CLARATy* architecture hides control and perception threads inside its object-oriented design. In fact this latter point is also true for nearly all layered architectures, since upper ”decisional” or ”deliberative” layers are made of entities (often explicitly drawn) that are responsible of AI-based planning [1] [29] or environment representation and navigation [4] mechanisms. These entities can also be viewed as *activities* of higher level of decision. One important property is that *activities* are arranged into *layers* according to their decisional complexity.

As reported in the previous subsection, some architectures propose a decomposition of control architectures into *systems*. The concept of *system*, as a part of the decisional process that incorporates the control of specific parts of the controlled robot, explicitly exists in a really limited number of architectures and under different names and forms like IDEA agents [21] or Robotic Resource [22]. This concept also exists in other architectures but without being explicitly defined. The advantages of this concept are that (1) it is useful for decomposing decisional process (cf. previous subsection) and (2) it can be generalized in such a way that it is applicable for all architectures. The generalization consists in a *systemic* organization of control architectures: a system can potentially be decomposed into subsystems that are systems responsible of the control of sub-parts of the controlled physical part of robot. Sub-systems being systems they can in turn be decomposed in such a way. Since there can be a single system for decomposing the control architecture of a given robot (i.e. no systemic decomposition) and that an entire robot team can be viewed as a *system* itself decomposable into subsystems (one for each robot), the concept of *system* can be easily applied to all control architectures. One can notice that the *systemic* organization is orthogonal to the *hierarchical* one: each subsystem can incorporate both reactive and long-term decision-making *activities* and so can itself be *layered*.

Another useful but rather exceptional concept is the *knowledge* concept, that helps to identify the data and know-how used in a control architecture. If never mentioned as with the term *knowledge* this concept explicitly exists in architectures using object-oriented models in their description, more precisely *CLARATy* [29] and *LIRMM* [9] architectures. Object-class hierarchies represent the knowledge used in the architecture, for example robot physical properties knowledge or environment knowledge. The functional layer of *CLARATy* partially merges knowledge with activities while these two aspects are clearly separated respectively in object class and Petri-net modules in *LIRMM* proposal. The advantages of this concept are that (1) it is useful for qualifying more precisely the responsibilities of *activities* by explicitly defining what type of information they use and (2) it implicitly exists in all architectures. Indeed, whatever the *activity* taken into account in any architecture, it uses a specific knowledge of the robot and the environment. For example, any control *activity* is based on a representation of the robot morphology and kynodynamics. Of course, this representation is in most of time completely hidden in the architecture design. Another related aspect is *knowledge specialization*, effective in *CLARATy* thanks to class specialization mechanism. *Knowledge specialization* helps to define different levels of knowledge refinement and to qualify more precisely at which level of refinement an *activity* works.

One thing that emerges when studying the different architectural solutions proposed along history is the huge diversity of interactions used. Nearly each architecture use a specific set of interaction protocols, of different degrees of abstraction according to their implementation. Interactions are the most recurrent variations that can be found between control architectures. Since its impossible to list all possible interactions (more especially as new ones could be defined in the future), a concept generalizing them is necessary in a domain language for control architecture description. Interaction partly influence the control design methodology. For example, specific protocols, like subsumption links [7] or vote protocols [23], are used in *reactive* and *behavior-based* architectures to put in place arbitration mechanisms; layered architectures use different event notification protocols to make sensing activities communicate with upper layer activities [9]. If interactions can be explicitly represented, it would certainly be useful for a better understanding of the design methodology used for a given control architecture solution.

3. A Control Architecture Modeling Language

The proposed modeling language focus on the control *control architecture* -i.e. the decomposition of robot decisional process, and does not take into account implementation aspects -i.e. robot software and hardware architectures.

3.1. Concepts, Terminology and Graphical Conventions

One important thing to take into account when designing a domain specific language is that it has to be complete, minimalist and easily readable. Complete, because each domain solution has to be expressible with the language. Minimalist, because the minimal set of concepts has to be provided to express solutions. Easily readable because a quick understanding of a solution is necessary. This is a big challenge in itself because the right balance has to be chosen between a great variety of precise domain concepts on one hand and a restricted set of generic domain concepts that can be more easily learn on the other hand.

3.1.1. Main Concepts

First, we define four main concepts: *Knowledge*, *Activity*, *Coordination*, *System*. To this end, the term task is used, and it has to be understood in the general meaning, -i.e. the fact of doing something.

Knowledge: a *Knowledge* entity identifies a structured piece of information about the world within which the robot controller evolves. It can directly refer to the physical world (environment, robot body) or to a concept bound up to this physical world like a ”phenomenon” or an ”event”. It can also refer to a know-how of the robot relative to this world, i.e. a way of detecting/solving problems relevant to this world (e.g. criteria defining singular configurations of the robot). Finally, the control architecture being itself part of the robot world, a *Knowledge* entity can also explicitly refer to (parts of) it, to get for instance a form of introspection.

Activity: an *Activity* entity is responsible for the achievement of a task that plays a role in the robot decisional process, using a set of *Knowledge* entities. For example, an *Activity* entity (activity for short) can refer to the observation of the environment state, of the robot (body) state, or even of the controller state. It can also refer to: low-level control of the robot, like the application of a control law, medium-level control as control context commutation, and high-level control like planning. Finally, it can also refer to a learning activity (creating or refining knowledge) whatever the level of control is concerned. An activity uses (internally) *Knowledge* entities to elaborate its decisions. For example, an activity defining a control law application uses knowledge on environment and robot morphology to compute actuator commands.

Coordination: a *Coordination* entity is responsible of the way a set of activities interact by exchanging or sharing knowledge. For example, it can express collaboration, i.e. an interaction type defining how different tasks are distributed among activities to achieve a more complex task. This is the most common case of *Coordination* entities, for instance command execution requesting or event notification. It can also express competition, i.e. an interaction type defining how several activities achieve the same task. This the the type used in many *behavior-based* or *reactive* architectures, for instance vote protocols or subsumption links. A same activity can be involved in more that one coordination. *Knowledge* entities that are exchanged or shared during coordination depends on the nature of the interaction. For example, in an

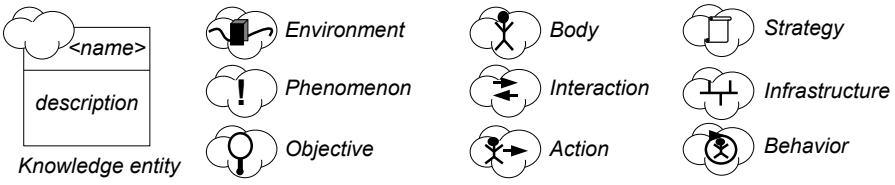


Figure 1. Some graphical conventions for representing *Knowledge* entities

event notification, *Knowledge* entities can represent specific states of the observed physical system (environment or robot body).

Systems: a *System* entity is an abstraction of the control of a physical (mechanical) entity (morphologically distributed or not). For example, a *System* entity (system for short) can refer to the control of a robot's part (e.g. the arm or the vehicle of a mobile robot), to the control of the (entire) robot (e.g. the mobile robot) or even to the control of a robot team. It is responsible of the way a set of *Activity* and *Coordination* entities, concerning this physical entity, are organized in order to achieve a set of tasks. This organization is done with a hierarchy of *layers*. A *layer* is an abstraction that symbolizes a "level of decision complexity" whatever the complexity of the decision is (from "simple" reactive decisions to "complex" deliberative decisions). A system organizes its internal activities by associating each of them to a *layer* according to its relative decision complexity in the system. A system also defines relations between its internal control architecture and *infrastructure* elements, like sensors and actuators. An *infrastructure* element is an abstraction that represents primitive part of the robot that provides to systems inputs (sensors), outputs (actuators) or both (physical communication links) or abstract composition of that parts (e.g. all actuators of the arm) by which the system retrieves information from the world within which the robot evolves.

3.1.2. Knowledge

A *Knowledge* entity is quite complex to detail without more specialization. At the highest level of abstraction, the only thing that can be said is that it contains a model synthesizing the type of knowledge and data representing the parameterization of this model for a given context. "Model" means any form of representation of the knowledge, being it mathematical formulas, empirical models, declarative models like CSP, biologically inspired models like neural networks, geometrical models, etc.

Since *Knowledge* entities are used to qualify other entities, its specialization to more precise domain abstractions is a way to improve easy understanding of models. In this way, the work done by Brugali and Salvaneschi in stable aspects of robot development [8] provides a good terminological basis that is extended here. So, knowledge entities can be arranged following three categories: *Embodiment*, *Situatedness* and *Intelligence*.

The *Embodiment* refers to the consciousness of having a body that allows the robot to experience the world directly. In this category, we find Knowledge entities like that representing the *Body* or *Infrastructure* of a robot, or some of their parts. The *Body* Knowledge entities allow for representing the electro-mechanical devices of robot operative part. *Body* entities can be themselves associated with other Knowledge entities of the *Embodiment* category. For instance a *Body* is associated to *Morphologies* and *Kinodynamics* [8] that are physical properties of robot body. A *Morphology* knowledge entity represents shape of a *Body*, its physical components and their structural relationship. For example, a robot can have a humanoid morphology, an animal-inspired or a human-vehicle inspired morphology. A *KinoDynamics* knowledge entity represents the kinematic (position and velocity) and dynamic (acceleration, force) constraints that limit the relative movement of the robot's body in the environment (morphological constraint like links and joints, physics law like gravity, etc.). The *Infrastructure* entities allow for representing the electronic-communications devices of robot controller part. *Infrastructure* entities represent elements like *Sensors* and *Actuators* or even more complex composite elements. *Body* and *Infrastructure* entities can themselves be decomposed into smaller part. For example, a rover robot entire *Body* can be decomposed into an *Arm*, a *Vehicle* and a *Camera*. Knowledge on embodiment can be so modularized.

The *Situatedness* refers to evolving in a complex, dynamic and unstructured environment that strongly affects the robot behavior. In this category, there are Knowledge entities like that representing the *Environment*. The environment is viewed as a continuum of physical configurations, but from a decisional point of view it is a discrete spatial-temporal milieu that is made up of any kind of dynamic or static elements such as people, robots, equipments, buildings, animals, mountains, etc., and hosts all their (potential) mutual *Interactions*. It can be decomposed into sets of *Environment* entities that represent any spatial part of it. *Interactions* are knowledge entities that represent interactions that can occur between the robot and the environment. Contrary to [8] the concept of *Interaction* is here limited to interactions governed by physic's laws (movement in environment, objects grasping, physical quantities measurement, etc.). In this category, there are also Knowledge entities like that representing the *Phenomena*. A *Phenomenon* refers to something that can occur in the environment, which is directly perceptible (thanks to specific *Interactions*) or estimable by the robot. To describe these entities we need others ones like those representing *Places* in the environment (the "where"), *Objects* (the "what") and *Time* (the "when").

The *Intelligence* refers to the ability of the robot to adopt adequate and useful behaviors while interacting with the dynamic environment. In this category, we can find *Behaviors* and *Actions*. Briefly, an *Action* denotes a capability of

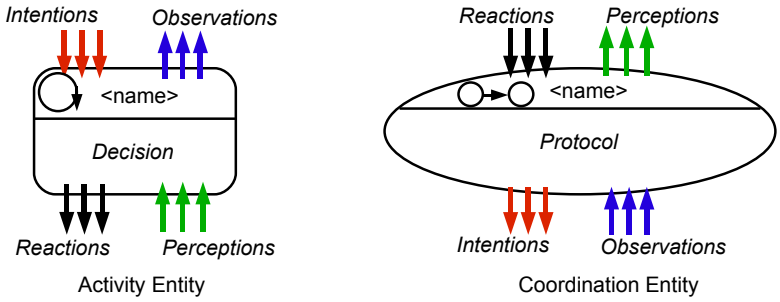


Figure 2. Graphical conventions for representing *Activity* and *Coordination* entities

the robot (or part of it) to act in order to obtain a given result (e.g. a given effect on the environment). For example, an *Action* can represent the fact of "reaching a given place at a given time". The expected result of *Actions* can be represented by *Objectives* (e.g. the place and time). *Behavior* denotes an abstract representation of the way the robot behaves when realizing *Actions* according to the possible robot-environment *Interactions*. For instance, a *Behavior* can be "being attracted by the nearest heat sources", it is defined thanks to *Actions* like "reaching a given place", "detecting heat sources" and "defining nearest reachable source" and according to *Interactions* like "heat measurement", "movement in room" and "environment perceiving". *Knowledge* entities of this category can also represent *Strategies* associated to each *System*. A knowledge entity representing a *Strategy* contains an action planner and for each action defines the behavior(s) to be selected (and if necessary merges) to reach its objective. *Behavior* is the result of the activation of a set of coordinated *Activities* (or a single one). So, to allow the precise description of *Strategies*, Knowledge entities can also represent the *Activities* and *Coordination* used to give concrete expression to the *Strategy*, to allow the robot to reason on. This conceptual decomposition of *Intelligence* is partially detached from [8] proposal, to put in adequacy the organization of knowledge entities with other main domain concepts.

Figure 1 shows some graphical conventions used to describe *Knowledge* entities. The different types of *Knowledge* entities are represented using different symbols to clarify their intrinsic differences. To this point, concepts allows only to model the "passive" characteristics of robot controller architecture, not the "active" ones.

3.1.3. Activity and Coordination

"Active" characteristics are expressed thanks the *Activity* and the *Coordination* entities. An *Activity* is an entity of any level of decision that puts in place *Perception-Decision-Reaction* cycles. Graphical conventions to represent them and their properties are represented in figure 2.

It receives *Perceptions* (required pieces of information, or significant phenomena notification) from other activities or from the infrastructure (sensors). Each *Perception* of an activity is associated with one or more knowledge entities of any level of abstraction, from simple sensor data to complex computed robot or environment states. It contains a *Decision* mechanism that computes *Reactions*. This mechanism is of any level of abstraction, from simple control law computation to a high-level planning or supervisory control mechanism. The *Decision* mechanism handles activity internal knowledge (*Body* and *Environment* for instance) and knowledge coming from *Perceptions* to determine the adequate *Reactions* to adopt. *Reactions* represent the way an activity wants its *decision* to be realized by (eventually) others activities or by the infrastructure (actuators). Each *Reaction* is associated with one or more knowledge entities of any level of abstraction, from simple actuator data to a high-level order (e.g. an *Objective*). The *Decision* mechanism can be influenced by *Intentions* it receives. An *Intention* represents a goal that an activity intends to accomplish -i.e. a goal influencing its *Decision*. An *Intention* is associated to knowledge entities representing, for instance, the desired state of the robot Body, related or not to the *Environment*, or a desired *Behavior*. The *Reaction* emitted by an activity can be viewed as an *Intention* by the activity that receives it. The *Decision* mechanism can also emit *Observations*. An *Observation* represents an interesting state of: decisional process, body or environment. For instance, an activity that detects an obstacle in the environment can transmit the corresponding *Observation* associated with corresponding knowledge entities representing the obstacle. An emitted *Observation* can be viewed as a *Perception* by activities that receive it. Like for knowledge entities, activities could be categorized into more specific entities like for instance, *Environment Observers* or *Body Motion Planners*, and so on, but this specialization should take place at the moment when a consensus on decisional process decomposition will be accepted.

Intentions, *Observations*, *Reactions* and *Perceptions* are exchanged by activities thanks to *Coordination* entities. A *Coordination* entity is an entity of any level of abstraction, that imposes a protocol for knowledge exchange or share between a set of activities. It can represent various interactions like simple event notifications and resource request on time interval as the one used in CLARATy, specific subsumption or inhibition links, as well as a complex vote protocol like in DAMN. In fact, it depends on the protocol and on the nature of *Intentions*, *Observations*, *Perceptions* and *Reactions* taken into account by the *Coordination*. Graphical conventions to represent *Coordination* entities are presented in figure 2. *Intentions*, *Observations*, *Perceptions* and *Reactions* are optional features for both activity and coordination entities (even if using none of them makes no sense).

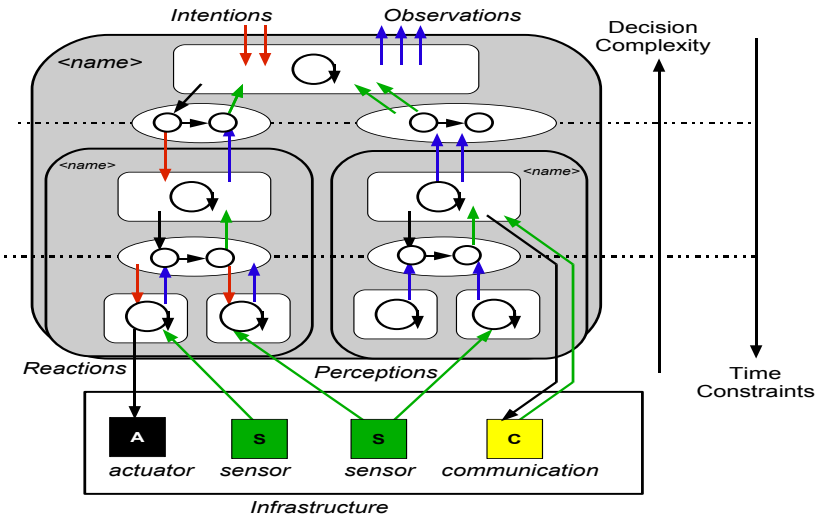


Figure 3. Graphical conventions for representing *Systems* entities

3.1.4. System

Systems are entities used to describe control architectures. Each *System* contains a set of coordinated activities and share with some of them some of their inputs and outputs (i.e. *Intentions*, *Observations*, *Perceptions* and *Reactions*). So a *System* can be considered itself as a decision-making system just as activities and can coordinate with other activities and/or *Systems*. Inside *Systems*, activities are organized according to a layered approach. The number and the precise semantics of each layer (decisional, reactive, executive, functional, etc.) is let undetermined to allow a maximal flexibility, main hierarchical organization criterion being the ”decisional complexity” (increasing from down to up) and real-time constraints (increasing from up to down).

Systems being used to describe the control architecture of an identified piece of the robot, they are in relation with *Body* Knowledge entities they are responsible of. These latter are useful to ”reason” about the robot body while systems are useful to exploit it. For example a *Manipulator System* control a robot *Arm*. System being able to contain other systems, the control of the robot or group of robots can also be described recursively. For example, the system controlling a rover robot can be composed of one system controlling its arm (*Manipulator System*) and another one controlling its vehicle (*Locomotor System*), like in CLARATy. All activities contained in a *System* participate to the control of the same part of the robot *Body*. A *System* can also explicitly refers to the *Infrastructure* of the *Robot Body* to associate its *Intentions*, *Observations*, *Perceptions* and *Reactions* with related *Infrastructure* elements, and more particularly *Sensors* and *Actuators*. For example, the *Manipulator System* refers to joints sensors and actuators of the arm and put them in relation with its *Perceptions* and *Reactions*. Graphical conventions for representing systems and robot infrastructure are presented in figure 3. This figure represents a *System* containing two *Subsystems*, where *Subsystems* contains two layers (layers are differentiated by dotted lines) and the *System* contains three layers, including the two preceding layers and an upper layer. Activities contained in *subsystems* interact with sensors and actuators of a more global infrastructural element (e.g. arm sensing and actuating infrastructure).

3.2. Language Meta-model

Now that concepts have been defined, this section presents the modeling language meta-model which reifies these concepts at the modelling level.

3.2.1. Reifying Concepts

The diagram of figure 4 shows the way main concepts are reified into the meta-model. Interesting things to denote in this diagram are:

- All control architecture entities are generalized into an *Entity* abstract class, in turn specialized into two abstract class *Knowledge entity* and *Decisional Entity*. This latter class is a generalization of all entities used in decisional process decomposition.
- A composite pattern is used to describe relations between *Activity*, *Coordination* and *System* entities allowing so ”recursive” decomposition of architecture into coarse-grained *Systems*.
- A *Decisional Entity* uses internally a set of *Knowledge Entities* (association with the *used* role).
- All types of inputs and outputs of *Decisional Entities* are generalized into a *Interaction Point* abstract class. According to the diagram, a same *Interaction Point* can be shared by more than one *Decisional entity*, for example

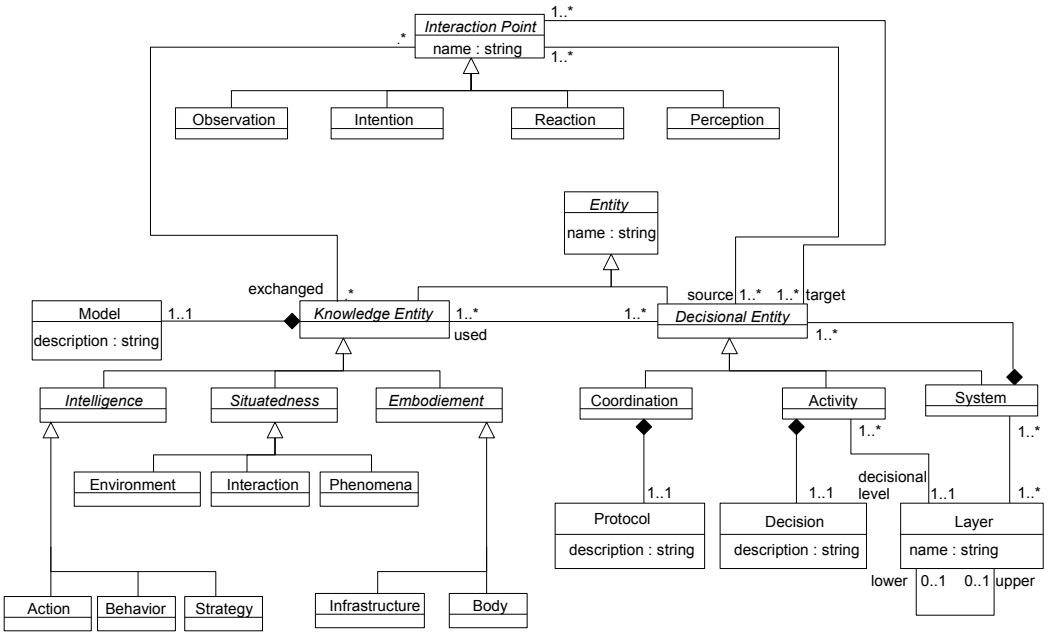


Figure 4. Meta-model : diagram of main concepts

a same *Reaction* can be shared by an *Activity* and its containing *System* (than in fact exports this *Reaction* outside its limits). In consequence it can have more than one source and more than one target.

- An *Interaction Point* exchanges a set of *Knowledge Entities* and a same *Knowledge Entity* can be exchanged by any number of *Interaction Points*.
- A *System* contains a set of hierarchically ordered *layers* (according to the *lower-upper* relation) and *layers* can be defined across many *Systems*.
- An *Activity* is associated to a unique *layer* that corresponds to its "decisional level" and a *layer* can contain many *activities*.

The meta-model does not precise the way *Models*, *Protocols* and *Decision* are described. Their string *description* attribute is added to allow for a description in a natural language, which of course is not formal but, in a first time, the most simple solution is preferred.

This diagram shows that *Knowledge Entities* are shared by the other entities. This is explained by the fact that a same knowledge can be use in many parts of the robot control architecture. For example knowledge on robot body is used in all activities that put in place control loops. This can be compared to a kind of separation of concerns since *Knowledge Entities* can be viewed as aspects that crosscut the decisional process decomposition. This is taken into account in the language by differentiating knowledge representation concern from decisional decomposition one: *Knowledge Entities* are modeled in a first dimension and *Decisional Entities* in a second one. This two dimensions are related to each other with links (represented in decisional dimension) between *Knowledge* and *Decisional Entities* (relations where *Knowledge entities* play *used* and *exchanged* role), which represents aspects weaving.

3.2.2. Details on Knowledge Entities

Relations between the different types of *Knowledge Entities* are described in the diagram of figure 5. With the purpose of conciseness, the diagram does not precise all meta-classes, like *Time*, *Places*, *Objects*, *Morphologies* and *Kinodynamics*, but they can be deduced from previous discussions.

The diagram focuses on structural relationships between *Knowledge Entities* that are according to previous definitions.

- A *Body* evolves in a set of *Environments*, it can support a set of *Infrastructure* elements and it participates to a set of *Interactions*. A *Body* can be decomposed into smaller *Bodies* as well as *Infrastructure*.
- An *Environment* can be decomposed into a set of more spatially restricted *Environments*.
- A *Phenomenom* occurs within an *Environment* and is detected thanks to a set of *Interactions*.
- An *Interaction* occurs within *Environments*.
- A *Behavior* is activable according to a set of possible *Interactions* and can be observed when the robot realizes a given set of *Actions*.
- An *Action* commands a *Body* and has, as goals, a set of *Objectives*.
- An *Objective* references the target *Environement* to express its spatial localization.

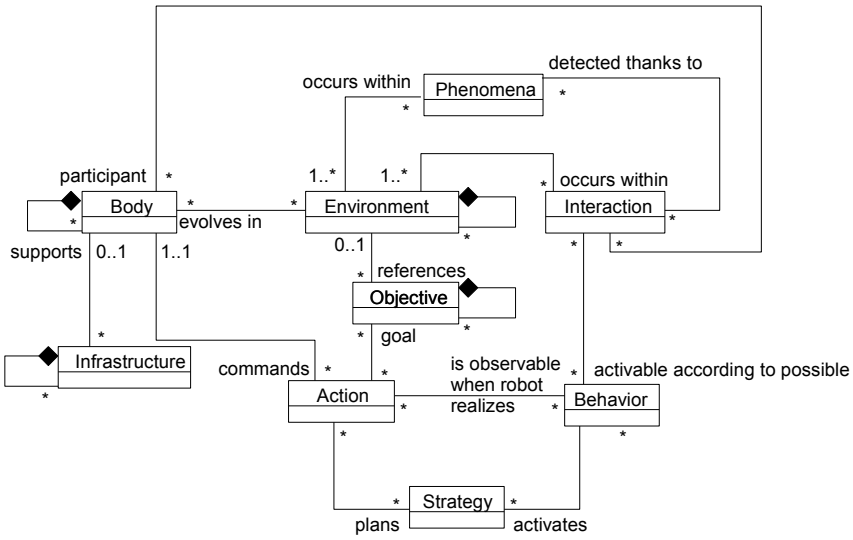


Figure 5. Meta-model : diagram detailing important *Knowlegde Entities*

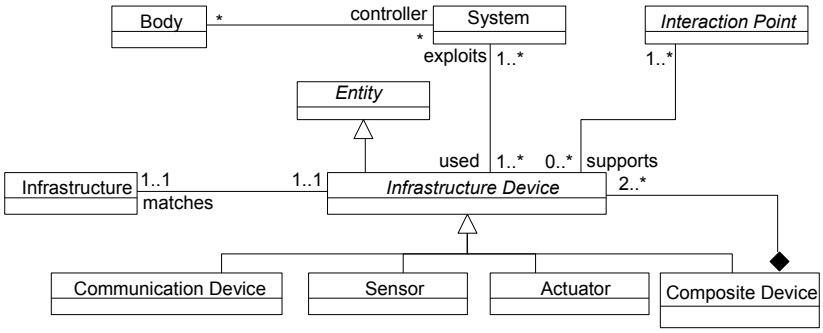


Figure 6. Meta-model : diagram defining entities used in detailed architecture description

- A *Strategy* plans a set of *Actions* to realize, and according to this set it activates a set of *Behaviors*.

This relations in the meta-model will be traduced by links between *Knowledge Entities* instances in a control architecture model. These links will be stereotyped with << ... >> label. For instance for links between *Body* and *Environment*, the stereotype <<evolvesin>> will be used. Composition links between *Knowledge Entities* in the meta-model are traduced by composition links in a model.

3.2.3. Complete Control Architecture Description

Since a precise description of control architectures, like it is presented in figure 3, has to be possible, the relation between *Decisional Entities* and *Physical Infrastructure Elements* have to be described. This is the purpose of the diagram of figure 6. This part of the meta-model more specifically details *Systems* according to their relation with operative part they control and physical elements they use to control it.

The relation between a *System* and the robot operative part it controls it expressed according to the *controller* relation between *Body* and *System* meta-classes. A constraint, that is not expressed in the diagram, is that all *Decisional Entities* it contains participate to the control of the same *Body* or sub-parts of it. To describe the way a *System* controls a *Body*, the diagram introduces the abstract class *Infrastructure Device* that is itself specialized into four concrete classes :

- *Communication Device* (square with *C* label cf. fig. 3) that represents a device that the robot uses to communicate with human operators or other robot, for instance a Wifi device.
- *Sensor* (square with *S* label cf. fig. 3) that represents a physical sensor, for instance a joint position sensor.
- *Actuator* (square with *A* label cf. fig. 3) that represents a physical actuator, for instance a joint position command actuator.

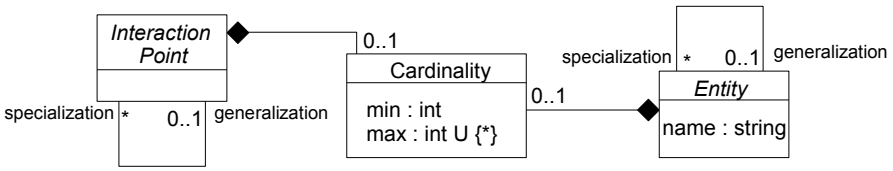


Figure 7. Meta-model : diagram defining Genericity, Refinement and Optionality management

- *Composite Device* (represented with a white rectangle around other devices in figure 3) that represents a set of simpler devices that can be grouped according to a given infrastructure decomposition. For instance, it can represent all joint sensors and all joint actuators of a same robot arm or all devices of a robot.

All *Infrastructure Devices* support a set of *Interaction Points* (restricted to *Perceptions* for *Sensors* and *Reactions* for *Actuators* by which they can interact with *Decisional Entities*. A *Composite Device* just supports the set of *Interaction Points* of its contained *Infrastructure Devices*. A *System* so exploits a given set of *Infrastructure Devices* by putting in relation its *Interaction Points* with theirs. For example, a *Manipulator System* exploits (at least) the *Composite Device* that represent the sensors and actuators of arm. It has to be noticed that an *Infrastructure Device* can be used by many *Systems* (of course it has to be done with care). Finally, diagram shows that an *Infrastructure* knowledge entity refers to a corresponding *Infrastructure Device*.

3.2.4. Genericity, Refinement and Variability

Now that all primitive structures of the language used to describe control architectures have been defined, this subsection states the management of control architecture solutions description. To describe architectural solutions, mechanisms allowing to deal with different degrees of genericity-specialization are necessary Mechanisms introduced in the language are defined in the diagram of figure 7. These mechanisms are all defined thanks the *Cardinality* class, inspired from UML [14] cardinalities.

Individually, a *Cardinality* just represents minimal and maximal amounts, where maximal amount can be infinite number (using * value). When a *Cardinality* is associated to an *Entity*, this means that this *Entity* can be refined, in an architecture that conforms to the solution, as many times as allowed by the *Cardinality* maximal and minimal value. So, *Cardinality* is used to deal with the genericity of control architecture solutions. For example, when considering a team of robots where each robot has the same control architecture, the architectural solution is expressed with a single *System* and its associated *Cardinality* that expresses the minimal and maximal number of team members. If no *Cardinality* is explicitly associated to an *Entity*, this means that this latter has a `[1..1]` cardinality. When a cardinality is associated to an *Entity* having a composite relation with other *Entities* (e.g. *System*, *Body*, etc.), this *Cardinality* is applied to all its internal description, allowing so to express a possible duplication of the *Entity* internal structure. For example, when a cardinality is associated to a *System*, this means that each of its contained *Activity* and *Coordination* can be duplicated as many times as specified by the *System*'s *Cardinality*. *Cardinalities* can also be associated with *Interaction Points*, meaning these points can be duplicated and refined when their source and or target *Entity* are refined.

Variability in a model allows the user to express different possible choices in its design. In feature models [15] used to express product lines, variability is expressed in two ways : optional features and group features. An optional feature corresponds to a feature that can or not be present in the resulting product. A group feature corresponds to a possible choice between a set of feature. In the present work, optionality means that an *Entity* is present or not in a refined control architecture. It is expressed with a `[0..1]` cardinality applied on this *Entity* or *Interaction Point* and can be extended to `[0..n]` cardinalities, whatever `n` value is. In a first time, the language only incorporate description feature for optionality and does not deal with group cardinalities. When an optional entity disappears in a refinement all its interaction points (for decisional entities) or relations (for knowledge entities) disappear. Furthermore, when a *Coordination* has only one possible participating entity in a refinement, it disappears.

Refinement is expressed in the meta-model with the *generalization-specialization* relationship on *Entity* and *Interaction Point* classes (cf. fig. 7), that is similar to the UML class *specialization* feature. This relation allow to refine a set of *Entities* to adapt an architural solution to a control architecture of a given robot. So the refinement is restricted according to cardinality. When a *Knowledge Entity* is refined, its relations with other *Knowledge entities* are themselves specialized with relations between related specialized *Knowledge Entities*. By default, in an architectural solution, all relations supports a `[0..*]` cardinality, meaning they are optional but can be duplicated many times.

4. Example

This section presents the use of the modeling language for the description of Aura Architectural solution. This example is defined according to a personal understanding and interpretation of Aura solution, based on its related bibliography. This is an important precision: since only the authors deeply know Aura, their initial viewpoints could be unintentionally

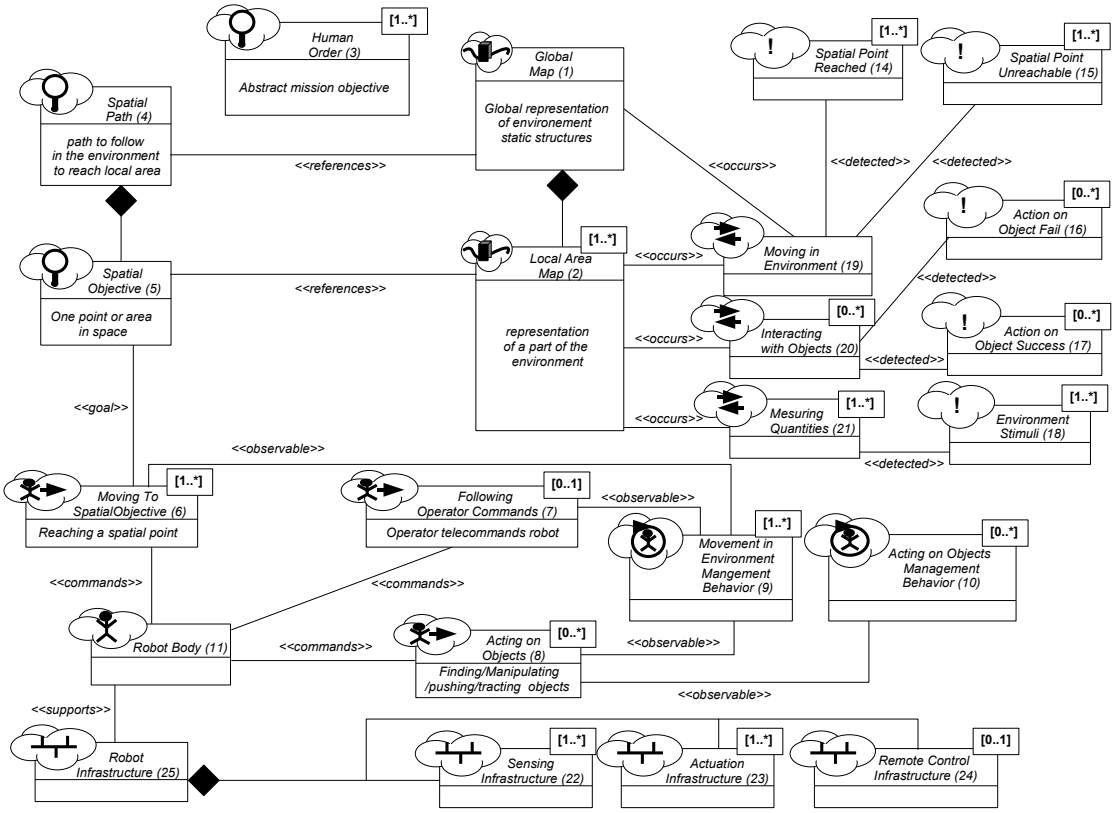


Figure 8. Aura Architectural Solution : Knowledge dimension (a)

not respected. So these examples should be seen as an illustration of the use of the modeling language rather than a ”definitive” opinion on the way Aura architectures are designed.

Aura is a generic and abstract control architecture solution which merges a reactive approach for low-level control design with a hierarchical ”deliberative” approach for high-level control design. It has been chosen because Aura is a very complex and general architectural solution which integrates preoccupations like teleoperation and learning.

The example mainly refers to [4]. The architectural solution is decomposed in two dimensions : the knowledge dimension presented in figures 8 and 9 and the decisional dimension presented in figure 10. The relations between these two dimensions are expressed thanks to numbers in parenthesis associated to knowledge entities (cf. fig. 8 and 9) and used in decisional entities (cf. 10).

4.1. Knowledge Dimension

The knowledge dimension decomposes (cf. fig. 8) the generic knowledge entities used in the Aura architectural solution. *Human orders* are abstract mission objectives given by human operators. The *Human Order* knowledge can be refined many times to describes different types of mission objectives. The *Global Map* is a global representation of static elements of the environment. It is decomposed into *Local Areas*, representing for instance things like rooms or corridors. The *Local Area* knowledge can be refined many times to describes these different types of *Local Area* if it is of importance. The *Spatial Path* is an objective that defines the path the robot has to follow in the environment. It is decomposed into a set of *Spatial Objectives* representing important places where the robot has to accomplish specific objectives. The *Spatial Objective* entity can itself be refined into specific objectives for specific *Local Areas*. Three generic knowledge entities are used to represent interactions: *Moving In Environment*, *Interacting with Objects* and *Measuring Quantities*. All these *interactions* occurs in *Local Area*, except *Moving In Environment* that also occurs within *Global Map* (i.e. going from one local area to another). A set of *phenomena* are detected thanks to these interactions: *Spatial Point Reached* and *Spatial Point Unreachable* detected thanks to *Moving In Environment*, *Action on Object Fail* and *Action on Object Success* thanks to *Interacting with Objects*, *Environment Stimuli* thanks to *Measuring Quantities*. Figure 8 shows that *Interacting with Objects* interaction and related phenomena are optional, in the sense that Aura Robot does not necessarily have means to manipulate objects nor have for mission to transport object in the environment. Other Interactions and phenomena are not considered to be optional (they have to be refined at least one time in a control architecture).

Figure 8 defines three types of actions: *Moving to Spatial Objective* which consists for the robot to go from current point to a spatial objective, *Following Operator Commands* which consists for the robot to be teleoperated, *Acting on Objects* which consists to make an action on an object to obtain a given result (e.g. catching an object, throwing it in

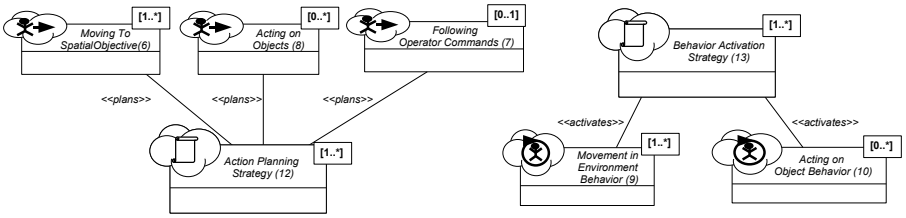


Figure 9. Aura Architectural Solution : Knowledge dimension (b)

a bin, etc.). All actions are considered to be optional except *Moving to Spatial Objective*, because Aura is initially a solution for robot mobility. Two knowledge entities generalize all behaviors of robot: *Movement in Environment Behavior* that generalizes behaviors like "approaching spatial point" or "obstacle avoidance" and *Acting on Objects Behavior* like "catching nearest waste". For reason of conciseness of the model, relations between interactions and behaviors are not represented. For example, *Acting on Objects Behavior* should be linked with a `<<possible>>` relation with all three interactions. All actions commands a unique *Robot Body* because Aura does not propose a decomposition of the knowledge of robot operative part into smaller controlled parts. The *Robot Body* supports a *Robot Infrastructure* itself composed of sensing and acting infrastructural elements and optionally a remote control infrastructural element. Control strategies are defined in figure 9. As the decomposition of knowledge of Aura architectural solution has been understood there are two categories of strategies represented by two generic knowledge entities: *Action Planning Strategy* defines an action planner and *Behavior Activation Strategy* defines a way a set of behaviors are activated and merged.

Next subsection shows the way all these entities are used in the decisional process decomposition of Aura architectures.

4.2. Decisional Dimension

The decomposition of robot architectures into subsystems does not exist in Aura: each robot is associated to a single system. System decomposition arises as soon as a group of collaborative robots is considered, each system (i.e. robot) being attached on its own infrastructure (cf. fig. 10). Robot system is decomposed into five layers, two reactive layers and three deliberative layers.

At the top layer there is the *Mission Planner* activity, in charge of collecting user intentions (i.e. mission long term goals and constraints) represented with *Human Order* (3) knowledge entity. At the layer below, the *Spatial Reasoner* activity receives requests from the *Mission Planner* to define the *Spatial Path* (4) (sequence of *Spatial Objectives*(5)) the robot must or can follow to achieve *Human Order* (3). Once a path is defined the *Plan Sequencer* activity is invoked to define the sequence of actions (6-8) required to follow the path and to achieve *Human Order* (3). For example, if the *Human Order* is to clean a building, the first action sequence would be: "go to room 1" (6), "collect waste"(8) and "put waste in the bin" (8) if the *Spatial Path* path is "room1, room2, room3, etc.". The action sequence corresponds to a state diagram where states are actions to perform and transitions are action changes. Transitions are associated to specific phenomena (14-17) that enable the state change. The planning itself is defined according to a given *Action Planning Strategy* (12) and according to *Global Map* and *Local Area* knowledge.

The activity entities of the deliberative layers interact around a *Long Term Environment Memory Sharing* coordination entity to consult and update *Global Map* and *Local Areas* knowledge (1,2).

Once a sequence of actions has been defined, the *Plan Sequencer* invokes the *Schema Controller* activity of the *Reactive Action Execution Layer* to realize each action (6-8). To this end the *Schema Controller* defines a *Behavior Activation Strategy* for each action to realize. *Schema Controller* interacts with *Perception Schemas* and *Motor Schemas* activities of the *Reactive Control Layer*. *Perception Schemas* activities are responsible of the production of *Stimuli* (18) or other action execution related phenomena (14-17) from sensors data. *Stimuli* are phenomena that contain partial instantaneous representations of the *environment* (1,2) or *robot body* (11). Other phenomena (14-17) are representing interesting state of actions execution. *Perception Schemas* can also produce long term Environment (1,2) representations (e.g. map of a room, update of the *Global Map*). *Motor Schema* activities put in place specific behaviors (9,10). Each *Motor Schema* activity is viewed as a control low, computed using *stimuli* (18) and *Robot Body* (11), to obtain a given behavior (9,10), as for example "obstacle avoidance". *Schema Controller* activity coordinates *Perception Schemas* and *Motor Schemas* in different ways. First, it translates the action execution request into a composition of Schemas (cf. *Composition Selection*): *Motor Schemas* are activated according to the behavior they represent ; *Stimuli* generated by *Perception Schemas* are redirected to *Motors Schemas* following a predefined *Behavior Activation Strategy* (13) and *Perception Schemas* can be configured with a *Spatial objective* (5). Second, some *Perception Schemas* are activated to generate action execution status (cf. *Action State Notification*). *Schema Controller* uses these phenomena to know if the current action has been (or cannot be) realized. It can then reply to the *Plan Sequencer* to indicate if the action succeeded or failed; if the action failed it tries to re-plan a sequence of actions or it indicates the *Spatial Reasoner* that the path cannot be followed. *Perception Schemas* can also interact with the higher-level activities by updating the *Long Term Environment Memory*. Finally, the *Schema Controller* sums and balances the commands to motors generated by activated

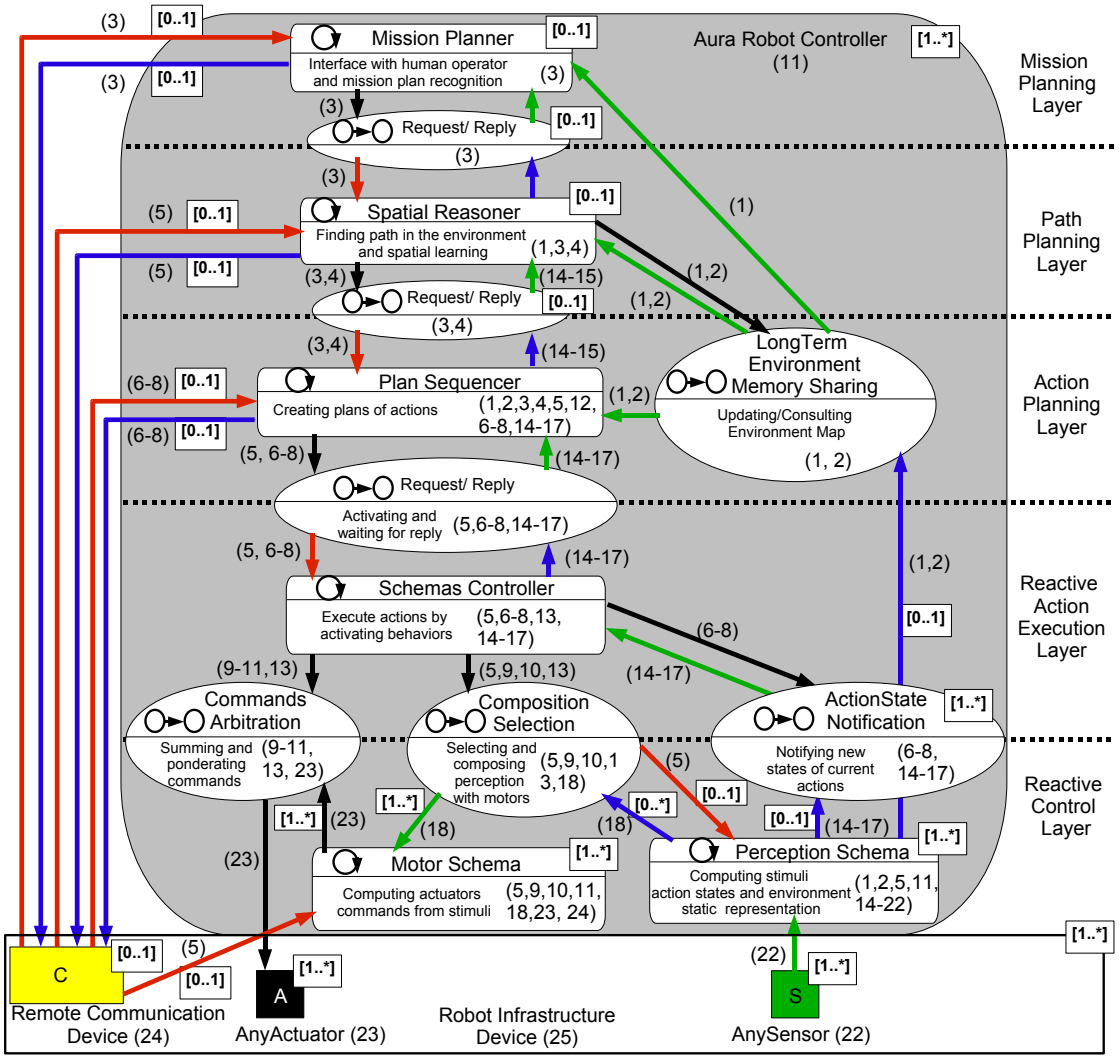


Figure 10. Aura Architectural Solution : Decision dimension

Motor Schemas (cf. *Command Arbitration*) according to the respective importance of behaviors in the chosen *Behavior Activation Strategy*. Once done, the command vector is applied to robot's motors. Human machine interaction can take place in many ways in an Aura architecture. Human can send *Human Order* to the *Mission Planner* at the top most level, it can directly send *Spatial Objectives* to the *Spatial Reasoner*, it can send actions to execute to the *Plan Sequencer* and finally it can directly interact with a specific *Motor Schema* to control robot movements.

5. Conclusion

This paper has argued on the necessity to define a dedicated language for communicating and understanding architectural choices in robotics. It proposed such a modeling language that embeds conceptual and terminological properties of the domain. It allows expressing architectural specificities by providing adequate abstractions and by promoting the importance of control organization thanks to different abstract types of entities: System, Activity, Coordination and Knowledge. An example of use of this model have been developed on a well-known architecture.

Future work aims for the design of decision mechanisms of Activity entities and that of protocols of Coordination entities. Simple concepts and terminologies have to be defined to easily express their respective properties. In the same way, more detail should be integrated to the different Knowledge entities types, to describe more precisely their respective models.

Another important point is a better understanding of genericity and refinement mechanisms. The refinement mechanism and cardinalities' impact on refinement should be formalized to avoid ambiguities. It would also be usefull to provide a way to describe alternative possibilities. In that sense, feature models [15] is a good source of inspiration because it provides structures, like group features and include/excluse constraints, to deal with it.

The short term goal of this work is to propose a precise frame to compare existing control architecture solutions and to highlight their advantages and limitations. In a longer term, the goal is to find commonalities and variations between all the main designs proposed in the domain.

References

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research, vol.17, n°4 p.315-337*, 1998.

[2] J.S. Albus and al. 4d/rdc : A reference model architecture for unmanned vehicle systems. Technical report, NISTIR 6910, 2002.

[3] J.S. Albus, R. Lumia, J. Fiala, and A. Wavening. Nasrem : The nasa/nbs standard reference model for telerobot control system architecture. Technical report, Robot System Division, National Institute of Standards and Technologies, 1997.

[4] R.C. Arkin and T. Balch. Aura : principles and practice in review. Technical report, College of Computing, Georgia Institute of Technology, 1997.

[5] R.P. Bonnasso, D. Kortenkamp, D.P. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. Technical report, NASA Johnson Space center, 1995.

[6] J.J. Borrely, E. CosteManier, B. Espiau, K. Kapellos., R. Pissard-Gibollet, D. Simon, and N. Turro. The orccad architecture. *International Journal of Robotics Research, Special issues on Integrated Architectures for Robot Control and Programming, vol 17, no 4, p. 338-359*, April 1998.

[7] R.A. Brooks. A robust layered control system for a mobile robot. *IEEE journal of Robotics and Automation, vol. 2, no. 1, pp.14-23*, 1986.

[8] D. Brugali and P. Salvaneschi. Stable aspects in robot software development. *Advanced Robotic Systems, vol. 3, n° 1*, pages 17–22, 2006.

[9] J.D. Carbou, D. Andreu, and P. Fraisse. Events as a key of an autonomous robot controller. In *15th IFAC World Congress (IFAC b'02)*, 2002.

[10] B. Espiau. La peur des robots. *Science Revue , Hors série n°10, p.49*, mars 2003.

[11] L. Fluckiger and H. Utz. Lessons from applying modern software methods and technologies to robotics. In *Software Integration and Development in Robotics, SDIR07 at ICRA07 (International Conference of Robotic and Automation)*, 2007.

[12] E. Gama, R. Helm, R. Jonhson, J. Vlissides, and G. Booch. *Design Patterns : Element of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing, 1995.

[13] E. Gat. On three-layer architectures. *A.I. and mobile robots, D. Korten Kamp & al. Eds. AAAI Press*, 1998.

[14] Object Management Group. Uml ressource page, http://www.uml.org/.

[15] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (foda) feasability study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Pittsburgh, PA, November 1990.

[16] J. Maaß, N. Kohn, and J. Hesselbach. Open modular robot control architecture for assembly unsing the task frame formalism. *Advanced Robotic Systems, vol. 3, n° 1*, pages 1–10, 2006.

[17] J. Malenfant and S. Denier. Architecture réflexive pour le contrôle de robot autonomes. In *revue L'objet. Langage et Modèle à Objets LMO'04, pp.17-30*, Octobre 2004.

[18] M. Mataric. Situated robotics, in encyclopedia of cognitive science, 2002.

[19] M. Mataric. Behavior-based control: example from navigation, learning and group behavior. *Journal of Experimental and Theoritical Artificial Intelligence 9*, pages 323–336, 1998.

[20] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain specific languages. *ACM Computing Surveys, Vol.37, n°4*, pages 316–344, 2005.

[21] N. Muscettola, G.A. Dorais, C. Fry, R. Levinson, and C. Plaunt. Idea : Planning at the core of autonomous reactive agents. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002.

[22] R. Passama, D. Andreu, C. Dony, and T. Libourel. Overview of a new robot controller development methodology. In *1st conference on Control Architecture of robots (CAR'06), pp. 145-163*, 2006.

[23] J.K. Rosenblatt. Damn : a distributed architecture for mobile navigation. Technical report, The Robotic Institute, Canergie Mellon University, 1997.

[24] C. Schlegel. A component approach for robotics software: Communication patterns in the orocos context. *18 Fachtagung Autonome Mobile Systeme (AMS)*, pages 253–263, 2003.

[25] D. Simon, B. Espiau, K. Kapellos, and R. Pissard-Gibollet. Orccad: Software engineering for real-time robotics, a technical insight. *Robotica, Special issues on Languages and Software in Robotics, vol 15, no 1, pp. 111-116*, March 1997.

[26] D.B. Stewart. The chimera methodology : Designing dynamically reconfigurable and reusable real-time software using port-based objects. *International Journal of Software Engineering and Knowledge Engineering, vol.6, n°2, pp.249-277*, June 1996.

[27] D.B. Stewart. Designing software components for real-time applications. In *2001 Embeded System Conference*, San Francisco, 2001.

[28] C. Urmson, R. Simmons, and I. Nesnas. A generic framework for robotic navigation. In *IEEE Aerospace Conference, vol. 5, pp. 2463-2470*, 2003.

[29] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The claraty architecture for robotic autonomy. In *IEEE Aerospace Conference (IAC-2001)*, March 2001.

Benefits of the MDE approach for the development of embedded and robotic systems

Application to Aibo

Xavier Blanc[1], Jérôme Delatour[2], Tewfik Ziadi[1]

[1]*Laboratoire d'Informatique de Paris 6 (LIP6)*
104 avenue du président Kennedy
75016 Paris
Xavier.Blanc@lip6.fr, tewfik.ziadi@lip6.fr

[2]*Equipe TRAME, CER ESEO*
4 rue Merlet de la Boulaye - BP 30926
49009 Angers cedex 01
jerome.delatour@eseo.fr

ABSTRACT. *Model Driven Engineering (MDE) raises the level of abstraction of the development life cycle by shifting its emphasis from code to models and model transformations. According to the well-known principle of separation of concerns, MDE advocates the isolation of business concerns from their technical achievement. The idea is that the business concerns can be modeled independently from any platform concerns. Therefore, business models are not corrupted by technical concerns. In this way, the main part of the development becomes an activity upstream, dedicated to business concerns through the elaboration of the application model that abstracts away technical details, i.e., the so-called Platform-Independent Model (PIM). The transformation of a PIM into a Platform-Specific Model (PSM) is then achieved when introducing into the PIM the technical considerations depending on the chosen platform. We applied MDE to develop software systems for Aibo which is one of several types of robotic pets designed and manufactured by Sony. The objectives was (1) to analyze advantages provided by models in this specific robotic context, (2) to measure the maturity of MDE provided technologies and (3) to highlight the limitations of the approach . We present in this paper the approach we follow for applying MDE for Aibo and we present the results we obtained.*

KEYWORDS: *Model Driven Engineering, Robotic, Ingénieurie Dirigée par les Modèles, Aibo, Domain Specific Language, Langage dédié*

1. Introduction

Model Driven Engineering (MDE) raises the level of abstraction of the development life cycle by shifting its emphasis from code to models and model transformations. According to the well-known principle of separation of concerns, the MDE advocates the isolation of business concerns from their technical achievement. The idea is that the business concerns can be modeled independently from any platform concerns. Therefore, business models are not corrupted by technical concerns. This is strongly anticipated as a way of simplifying the construction of systems. In this way, the main part of the development becomes an activity upstream, dedicated to business concerns through the elaboration of the system model that abstracts away technical details, i.e., the so-called Platform-Independent Model (PIM). The transformation of a PIM into a Platform-Specific Model (PSM) is then achieved when introducing into the PIM the technical considerations depending on the chosen platform.

The MDE approach is quite often applied for large scale industrial systems. Indeed, MDE standards such as UML (Unified Modeling Language) and on-the-shelf transformations such as “UML to EJB” have been defined to specify and to build those large systems. However, the MDE approach has been defined to specify and to build any systems, whatever their size and their business domain. Standards such as the MOF (Meta Object Facility) and technologies such as model transformations and model validation can be used to define domain specific modeling language that can be used to build any.

In order to check if MDE can really be used to build systems other than large scale industrial systems, we tried to apply it to build software systems for Aibo which is one of several types of robotic pets designed and manufactured by Sony. Aibo is a reactive system that can move, see, hear and speak. Aibo software can be developed in order to make Aibo react (move and speak) when it receives events (hear and see). Building software for Aibo is a complex task as (1) Aibo is a complex platform with a lot of joins and sensors and (2) no industrial development environment is provided by Sony. Moreover, Aibo should be considered more as a family of robotics pets rather than a single robot. Each member of the family has its specificities. Aibo software should be then reusable as much as possible for all members of the Aibo family. For all these reasons, we considered that Aibo was an ideal platform to check if MDE principles can be applied to build all kinds of systems.

The next section of this paper presents MDE concepts used to define domain specific modeling languages. Based on those concepts, section three of this paper presents how we applied MDE for Aibo. Then our conclusion is presented in section four.

2. MDE Approach

This section presents the MDE principles and core concepts. Despite its relative youth, the MDE approach defines a long list of acronyms (MDA, PIM, PSM, CIM, MOF, EMF, QVT...) and slightly adapts the meaning of some existing concepts (model, transformation, meta-model...) to its needs. Moreover, different implementations of the MDE approach exist: « Software factories » from Microsoft [Greenfield 04], MDA (Model Driven Architecture) by OMG [Soley 02], EMF(Eclipse Modelling Framework) which is the MDA approach of Eclipse [Budinsky 03], MIC (Model Integrated Computing) from the DARPA [Frake 97] ... Each of them defining their own vocabulary and techniques, it is still a challenge to reach a general consensus on the MDE core concepts.

Therefore, we present the definitions that reach a large agreement in the MDA community. These definitions are based on the works made by the french group « AS MDA » [AS MDA 06].

2.1 Models, meta-models and meta-metamodels

The notion of model is quite old and has been defined before MDE. "A model represents reality for a given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality." [Rothenberg89]. It is a set of statements about something which is under study (the system). For instance, a map of the world is a model of the world. Depending on the authors, this relationship between a system and a model is called either, « RepresentationOf », « RepresentedBy », « Describes »... As stated by J.M. Favre, it is very important to understand that this relation is actually defined between systems: being a model or a system under study is a relative notion, not an intrinsic property of an artefact. In fact these notions are roles that a system can play with respect to another system [Favre 04]. It is obvious to consider the map of a world as a system and make a model of it (for instance, a sketchy drawing of this map).

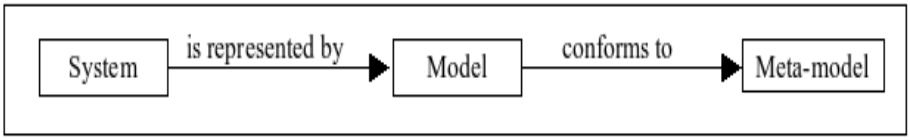


Figure 2.1: Relationship between system, models and meta-models

In MDE, another relationship is required for the models: they must conform to a meta-model. A meta-model is a precise specification of the considered concerns the model will represent. A meta-model of the map of the world could be the legend of

the map. In other words, a meta-model could be seen as a way to define the sets of possible models (a model of a set of models). A rough analogy with the language theory could be that a meta-model is one possible representation of the grammar of a language.

In each trend of the MDE, there is an attempt to define a meta-metamodel: a common language for defining meta-models. In MDA, they define a four-level modeling stack (sometimes called MDA pyramid) where:

- M0 is the real world,
- M1 is the model level,
- M2 is the meta-models levels, UML is a meta-model. In the next section of this article, a meta-model of the AIBO language will be presented.
- M3 is the meta-metamodel, it is self-defined and called MOF (Meta Object Facility).

A rough analogy with the language theory has been established in figure 2.2. The Extended Backus Norm Form (EBNF) could be seen as a meta-metamodel as it can define the grammar of different languages.

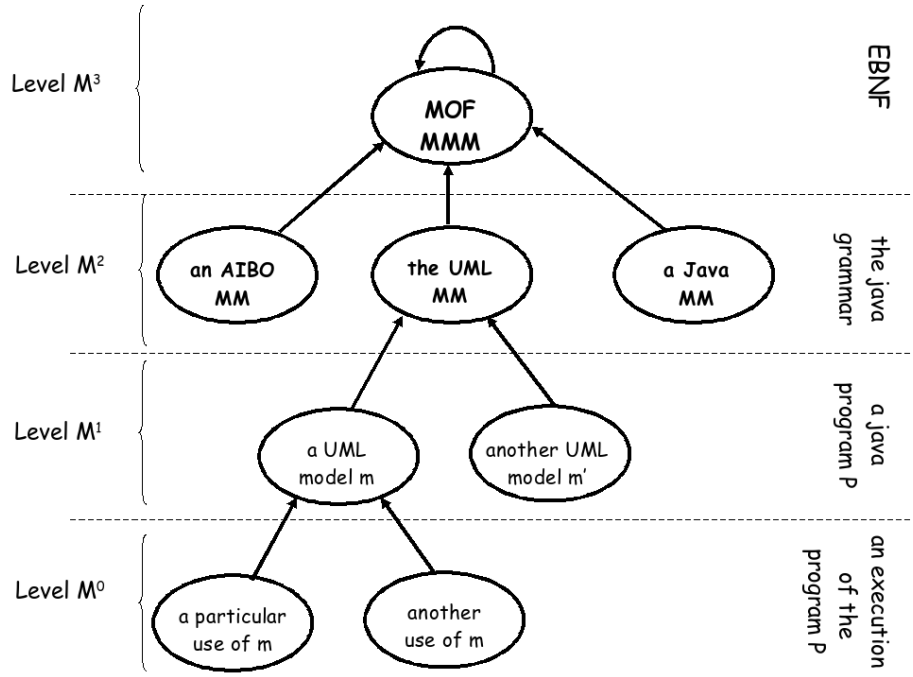


Figure 2.2 : MDA pyramid, adapted from [Bezivin 04]

MDA and other MDE trends facilitate the definition of DSL (Domain Specific language [Czarnecki 00]). It consists basically in defining a consistent meta-model and, thanks to the normalized MDA infrastructure, a model editor (textual and, in certain case, graphical) could be generated. The models are stored in an XMI format (an XML variant) and could be exchanged between MDE tools.

2.2 The notion of Platform

The MDA guide [MDA03] provides a generic definition of the platform concept "A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented". This is a very large high-level definition that leaves a wide scope for interpretation. There are actually several undergoing works in order to precise the notion of platform [Paulo 04] [Delatour 05].

A practice exists for differentiating the platforms which are dependent on a technology (the PSM: Platform Specific Model) and those which are independent (the CIM: Computational Independent Model and the PIM: Platform Independent model). This notion of « Platform independence » is difficult to define and is quite relative. The distinction between a PIM and a PSM is not clear-cut and will be dependent on whether one wants to consider different sets of target platforms. Although often used, these notions of PIM and PSM should be more clarified

2.3 Model transformation

In MDE, the translation from a PIM to a PSM is generally described as a transformation. A transformation consists in generating a set of target models from a set of source models. Target and source models must conform to meta-models. The transformation itself is a model that conforms to a meta-model. There is a huge variety of transformation languages [Czarnecki 03].

One can use traditional languages. For instance, a transformation could be written in Java using a library in order to facilitate the manipulation of the models. Several libraries have been defined in order to manipulate elements from model repositories (JMI for MDR, a MOF compliant repository, EMF for Eclipse framework...).

One can use dedicated transformation languages. There are a lot of different approaches, from declarative or imperative paradigm, based on graph grammar theory, based on template approach... In MDA, the QVT (Query, Views and Transformations) language [QVT 05] has been normalized in order to specify transformations. To the best of our knowledge, there is no implementation of QVT,

currently some transformation languages implement parts of it, but none are totally compliant.

Model transformations are not restricted to the translation of a PIM to a PSM, and a huge variety of transformations could be imagined (from the refinement of a PIM to the code generation). Thanks to the MDE facility for importing and exporting existing models (stored for instance in various textual format or in XML form), it is possible to define a transformation from a specification model to a validation model (for instance, in a petri net formalism). Therefore, it is possible to use the validation tools developped outside the MDE world. For a researcher, it is a gain of time that will be profitably invested in the definition of transformation rules. Indeed, developing its own transformation and parsing technology to different everchanging file format could be a time consuming activity, Moreover, the MDE community is starting to share their meta-models and transformations, available in meta-model zoos and transformation zoos.

3. MDE for Aibo

3.1. Approach

The figure 3.1 presents the MDE approach proposed to build Aibo Software.

Three meta-models have been built (M2 layer):

- The Robot meta-model defines concepts used to specify robots characteristics (joins and sensors). Thanks to this meta-model, models of the different members of the Aibo family can be elaborated. This meta-model has been used to elaborate the model of the ERS-7 member of the Aibo family. The ERS-7 is the last member manufactured by Sony and is the robot we used for running our software. For sake of simplicity, we won't present this meta-model in this paper. The complete meta-model is available in [AiboDev 06].
- The Validation meta-model defines concepts used to specify Aibo consistent and inconsistent states and transitions. For instance, if Aibo is seated, it should not be allowed to walk. This meta-model is used to elaborate different validation models that contain inconsistent states and different strategies (transitions) to gain consistent states. For instance, if Aibo is seated and has to walk, it should stand up first. For sake of simplicity again, we won't present this meta-model in this paper. The complete meta-model is available in [AiboDev 06]
- The Behavior meta-model defines concepts used by developers to specify the Aibo software they want to build. This meta-model can be

considered as the Aibo programming language. This meta-model is detailed in the following section.

In this approach, two kinds of users have been identified. One user is responsible of the elaboration of the robot models and the validation models. This user customizes the MDE environment for Aibo. This user is called the framework level user (M1: framework in figure 1). The other user is the Aibo developer. He/she elaborates models of the behaviors of Aibo. Those models have to be consistent with the robot models and the validation models. For instance, the behavior model cannot target joins and sensors which are not specified in the robot model. The behavior models cannot specify inconsistent behaviors (identified in the validation models).

Once the behavior model is completely elaborated, it can be automatically transformed into code thanks to predefined templates. The code generation step is presented in the following section.

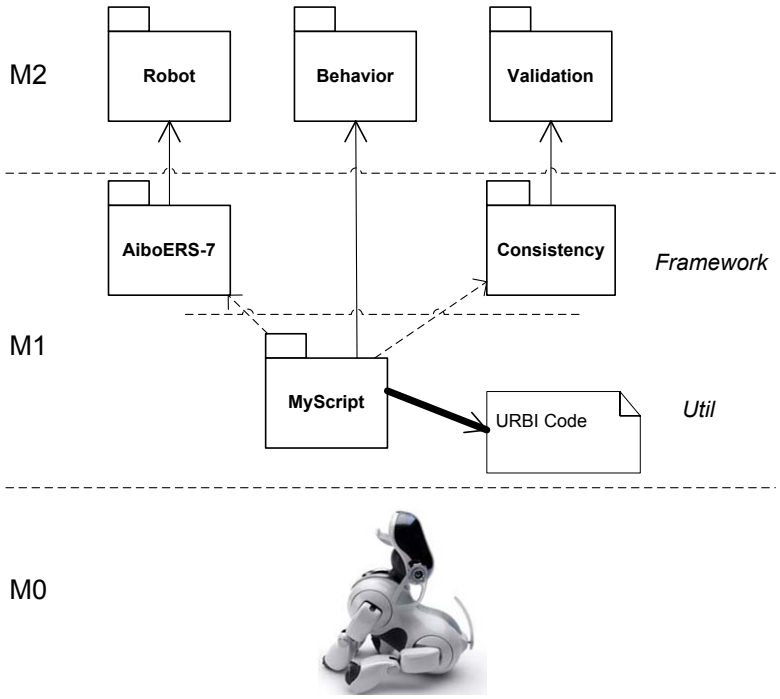


Figure 3.1: MDE Approach for Aibo

3.2. The Aibo meta-model

The figure 3.2 presents a simplification of the Aibo Behavior meta-model. The whole meta-model is available in [AiboDev06].

Basically, this meta-model defines that a Aibo behavior is a kind of sequential script composed of blocs of Aibo actions. Aibo actions are linked to joins and sensors of the Aibo robot model. For instance, the "walk" is specified as a bloc composed of several Aibo actions making Aibo moves its legs in order to walk. Blocs can be controlled by classical control blocs (if, loop, while, etc.). It should be noted that, for sake of simplicity, only the Loop meta-class is presented in figure 2.

Events received by Aibo are considered as condition of the "if" statement. For instance, when a developer wants to specify how Aibo react when it see its ball, he/she has to use the "if" statement with a kind of "ballVisible" condition. Pre-defined event that can be used in condition are specified in the robot models.

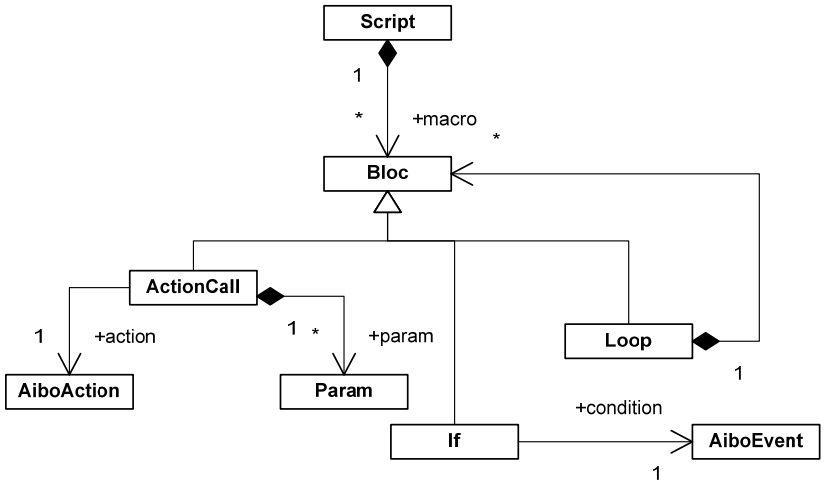


Figure 3.2: Simplification of the Aibo behavior meta-model

In order to validate this meta-model, three behaviors have been considered. Those behaviors are representative enough of behaviors developers usually want to specify:

- **A dance**. There are several dance competitions with Aibo. The dance behavior mainly consists in playing a song and chaining different dance steps.
- **A defense**. Aibo is sometime used as a defense pet. The defense behavior mainly consists in moving in a place looking if somebody moves and then bark.

- **A labyrinth escape**. Aibo can evolve in unknown places. Therefore, it has to know how to escape all places. There are several labyrinth escape algorithms. We choose one consisting in running parallel to walls.

All those behaviors have been completely modeled thanks to our meta-model, see [AiboDev06] for models of those behaviors.

3.3. Aibo to URBI transformation

There are few programming languages targeting the Aibo platform. The best one is probably URBI (Universal Real-time Behavior Interface) [URBI07] which is a scripting language specially dedicated to robots.

Programming with URBI is quite easy as it provides primitives for controlling joins and sensors of the robot. Moreover, URBI provides real-time primitives allowing actions to be performed in specific time frames.

URBI code generation from Aibo models can be seen as a kind of Model-to-Tex transformations (cf. section 2). To realize this transformation, a set of rules have been defined. Each transformation rule concerns a specific meta-class of the Aibo Behaviour meta-model, and targets a specific URBI primitive. The complete set of these transformation rules can be downloaded from [AiboDev 06].

The rules definition is not of particular interest since the Aibo behavior meta-model is script oriented and really fits to URBI. Thanks to them, URBI code can be automatically generated.

The rules have been validated on the three models of the behaviors presented in the last section. For each model a complete URBI code has been generated. This code was then deployed and run into Aibo ERS-7.

4. Conclusions and perspectives

The objectives of our study was (1) to analyze advantages provided by models in this specific robotic context, (2) to measure the maturity of MDE provided technologies and (3) to highlight the limitations of the approach.

In robotic we have used models in order to specify the robot characteristics, the consistent and inconsistent states of the robot and its behaviors. Thanks to the definition of the three corresponding meta-models we were able to specify links between them and to develop automatic operations (validation and code generation). We can say that MDE definitively are useful and offer advantages all specific domains.

Even if we have not presented the realization in this paper, the whole approach has been completely prototyped. EMF (Eclipse Modeling Framework) has been

used to realize editors for each meta-model. Thanks to those editors, user can graphically elaborate models. The validation module has been developed in Java and the code generation with MofScript [MofScript07]. The prototype realized clearly shows that MDE technologies are completely mature.

Even if code can be completely and automatically generated from models, we are not completely sure if it is better to elaborate an Aibo behavior model rather than directly write URBI code. Indeed, our Aibo behavior meta-model is maybe to much script oriented. This is, for us, the clear challenge of applying MDE: define the ideal domain meta-model.

5. Acknowledgment

The authors whish to thank Aleksandra Dworak, François Nizou and Nicolas Ulrich for having defined and realized this approach during the AiboDev competition.

Bibliographie

[AiboDev 06] http://www-src.lip6.fr/homepages/Xavier.Blanc/courses/CAR/AiboDev2006/

[AS MDA 06] AS MDA, « L'ingénierie dirigée par les modèles. Au-delà du MDA », Traité IC2, série Informatique et Systèmes d'Information, Hermes, 2006

[Budinsky 03] Budinsky, F., Steinberg, D., Raymond Ellersick, R., Ed Merks, E., Brodsky, S. A., Grose, T. J., « Eclipse Modeling Framework», Addison Wesley, 2003

[Bezivin 04] Bézivin, J., « In Search of a Basic Principle for Model Driven Engineering ». CEPIS, UPGRADE, The European Journal for the Informatics Professional V(2):21—24, 2004

[Czarnecki 00] Czarnecki, K., Eisenecker, U., « Generative Programming: Methods, Tools, and Applications », Addison-Wesley, 2000

[Czarnecki 03] K. Czarnecki and S. Helsen, « Classification of Model Transformation Approaches » Workshop on Generative Techniques in the Context of Model-Driven Architecture, OOPSLA'03, 2003

[Delatour 05] Delatour J., Thomas F., Savaton G., Faucou S., « Modèle de plate-forme pour l'embarqué », IDM '05, www.plantemde.org/idm05, 2005

[Favre 04] Jean-Marie Favre, « Foundations of Model (Driven) (Reverse) Engineering», several articles (episode 1, 2, 3), www-adele.imag.fr/~jmfavre, 2004

[Frake 97] Franke H., Sztipanovits J., Karsai G., « Model-Integrated Computing », Hawaii Systems of the World Manufacturing Congress, Auckland, New Zealand, 1997

[Greenfield 04] Greenfeld J., Short K. with Cook S., Kent S., (foreword by Crupi J.) « Software Factories, Assembling Applications with Patterns, Models, Frameworks and Tools », Wiley Publishing, 2004

[MDA 03] OMG, MDA guide version 1.1, http://www.omg.org/mda/, June 2003

[MofScript 07] http://www.eclipse.org/gmt/mofscript/

[Paulo 04] Joao Paulo Almeida, Remco Dijkman, Marten van Sinderen et Luis Ferreira Pires, « On The Notion of Abstract Platform in MDA Development », IEEE International Enterprise Distributed Object Computing Conference (EDOC'04), p 253-263, IEEE Computer Society, 2004.

[QVT 05] OMG, « MOF QVT Final Adopted Specification», OMG Document ptc/2005-11-01, http://www.omg.org/docs/ptc/05-11-01.pdf, 2005

[Rothenberg89] Jeff Rothenberg, "The Nature of Modeling.", Artificial Intelligence, Simulation, and Modeling, L.E. William, K.A. Loparo, N.R. Nelson, eds. New York, John Wiley and Sons, Inc., 1989, pp. 75-92.

[Soley 02], Richard Soley and the OMG Staff Strategy Group, « Model Driven Architecture », www.omg.org, 2002

[URBI 07] http://www.urbiforge.com/

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.

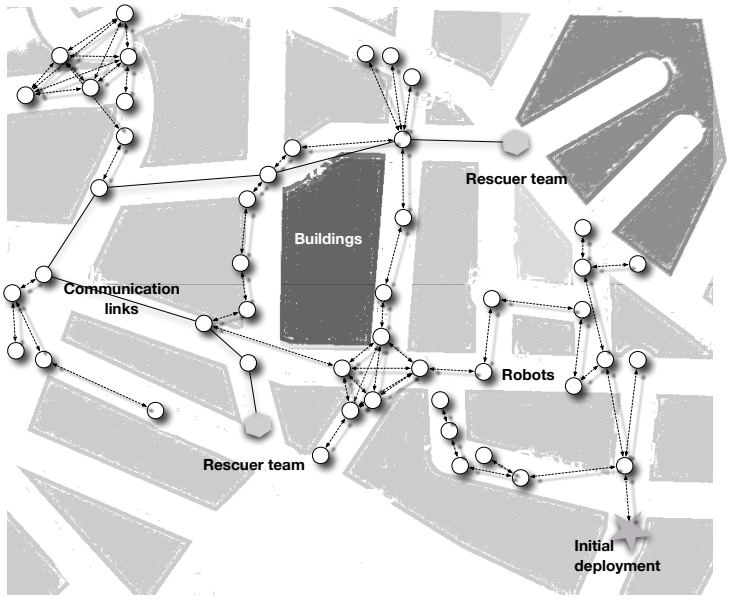


Figure 1: AROUND project

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.

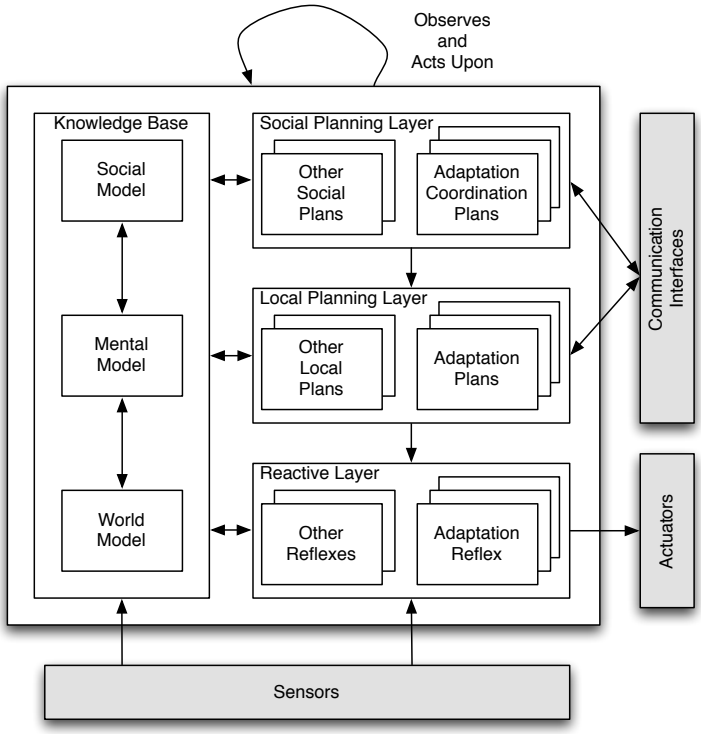


Figure 2: Abstract Adaptive Robot Control Architecture

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.

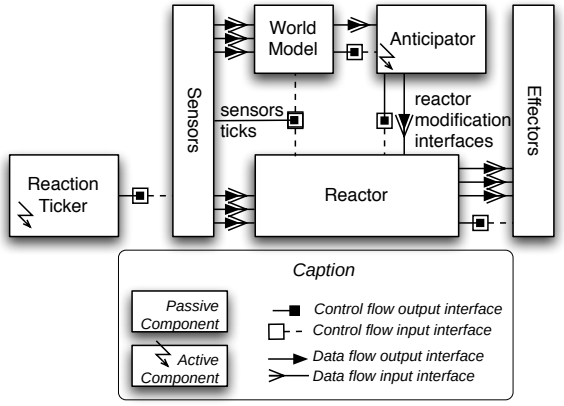


Figure 3: Maleva-based Anticipatory Agent Architecture

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.

Design Principles for a Universal Robotic Software Platform and Application to URBI

Jean-Christophe Baillie

***Abstract*— The number of generic robots available today on the market is increasing significantly. While this represents a great opportunity to carry on research using advanced robotic devices, including humanoid robots, there is still a serious limitation: all these robots are totally incompatible in term of software, and they are generally quite hard to program, lacking the proper abstractions and properties needed in robotics. We will present here a discussion on possible design rules for such a universal software platform and review a few already existing platforms with these criteria in mind. We will also introduce the URBI platform which is the result of our effort in an attempt to design a universal platform following these guidelines.**

I. INTRODUCTION

Looking at what happens today in Japan, Korea and the US, it appears that the number of generic robots available on the market will be increasing significantly in the coming years [9]. By ”generic robots”, we mean robots not specifically designed for an industrial task but general purpose robots, commonly usable in labs for research. Many of these robots, including humanoid robots, integrate cameras, wifi and on-board computers, and most of them are available for an affordable price. This is a great opportunity for research but while the hardware is making progresses, there is still no clear generic software platform emerging that could make these robots compatible and bring the benefits of cross-robot compatibility and component reuse. This is a major issue and several studies have highlighted the benefits of having such a universal software platform [4], [8], [5], [6].

Of course, robot programming can be done in C++, even in C, and to a certain extend every software platform is formally identical. But we believe that the debate should not be between using Java vs C++, or using a given technology vs another. The important question is to think about good properties of such a universal platform that would facilitate its adoption by a vastly heterogeneous community, mostly knowledgeable and sometimes already on-track with a software platform of their own. How could such a paradox be solved: to have a universal but also very flexible platform, which still brings value added in the process and does not stand just as an empty shell, while providing a clear orientation towards robotics. This will be the topic of our discussion in this article.

We claim that the URBI software platform developed in our lab makes a significant progress in the direction of such a universal platform. We will shortly present the key characteristics of URBI and compare it to other existing software platforms.

This work was supported by Gostai S.A.S., France

J.C. Baillie is with the National Institute of Advanced Technologies, Cognitive Robotics Lab, 75015 Paris, France jean-christophe.baillie@ensta.fr

II. WHAT ARE THE KEY REQUIREMENTS FOR A UNIVERSAL ROBOTIC SOFTWARE PLATFORM?

The challenge that we are going to investigate is: how could we design and promote a universal robotic software platform in the research community, what are the key requirement for such a platform?

As a guideline, there are three main questions that we should keep in mind when we investigate the potential features: are these features necessary, acceptable and usable by the community?

- Being *necessary* is an obvious requirement, however sometimes overlooked by “over-featured” solutions that can drawn the user in useless complexity.
- Being *acceptable* means that the community can realistically accept to use this feature which by essence creates certain constraints or enforces a point of view. For example, forcing the use of the C language exclusively or forcing the use of a very constrained class-based architecture will not be acceptable by many users.
- Finally, *usability* means that, however necessary and acceptable the feature is, it should be relatively easy to understand and integrate. The risk is otherwise to have a kind of ”complexity barrier” that prevents the widespread usage of the platform.

Breaking one of the three above requirements might be one the main reasons why we have not seen any successful platform emerging yet.

We detail below four key characteristics that we believe are required for a successful universal platform and whose implementation should fit with the necessary/acceptable/usable constraints presented above.

A. Flexibility

Being flexible means that the platform should be working with any operating system, be interfaced easily with any programming language and of course be suitable for any type of robotic application.

A non flexible platform will typically enforce several design choices and will, to a certain extend, constrain the user into a predefined way. While this has the advantage to improve general coherence and structure, it is also generally not acceptable (in the sense defined above) by the community. In fact, many research topics are precisely about finding a good architecture, a good paradigm or structure for robotics.

One can always bypass these constraints but at the cost of heavy custom-made layers. The difficulty here is that most platform developers claim to have a flexible solution, in the sense that it is possible to do anything with it, as long as one is ready to put enough effort in the process. A C++ library, like OPENR on the Aibo[10], is a good example. Here, the notion of *usability* that we have introduced before plays a central role. The flexility must be easy to implement, natural (using well known concepts, like objet oriented programming) and possibly transparent: no need to read a long documentation to know how to use it, and no need to develop, port or recreate preliminary tools/interfaces to benefit from it.

Note that non-flexible platforms have been imposed already in other domains, like desktop computers, but there was already a quasi monopolistic player to enforce it, which is not the case today for robotics. So we believe that a non-flexible platform (in the *usable* sense) will unlikely become a widely used universal robotic platform.

B. Modularity

Modularity adds the possibility to plug components in the platform to extend its capabilities. This is necessary to be able to start cooperations between labs and the industry, by developing and reusing modules for robotics. It is the basis of a software industry for robotics. The need for modularity is widely recognized, see for example [11], [15], [12]. Modularity is also imperative to move towards plug&play hardware components for robotics.

Again, this constraint will be acceptable only if it is very easy to deploy. Anyone should be able to develop components and the unavoidable structuring constraints should be reduced to the minimal, for the solution to be usable.

C. Powerful abstractions for robotics

Robotics is unlike classical computer related domains. The main difference is that robots have to explore hypotheses, runs things in parallel, react to events at various levels and generally speaking integrate vastly more software technologies than a regular PC. Among the important technical differences, parallelism and event-based programming are two core requirement in any robotic application. The underlying platform must support these abstractions natively.

Thread-based libraries on top of C++ or other languages do not offer the appropriate abstraction needed for robotics. The limitations of threads have been widely discussed already [16] and it is now recognized to be a major issue in parallel application development, especially now that we can see quad-core processors on the market. The main limitation of threads in their most primitive declination is that memory safe locking and code synchronization, which is are very challenging problems, are left to the programmer. This rapidly lead to bugs, even for careful and experienced programmers.

Something new is needed to handle parallelism at the right level of abstraction, just like object-oriented programming brought something new to modular development. The benefits of the abstractions available in C++ today compared to C are now widely recognized, even if some resistance existed at the beginning, and we claim that such a paradigm shift is needed for robotics as well, to integrate parallelism and event-based programming at the core of the platform.

Introducing some smart C++ classes to handle parallelism will raise the problem of flexibility. Theses classes will not be available in Python, Java or Matlab and, even if it is possible, a important effort would have to be done to provide a unified set of abstractions for all possible languages. A intermediary programming scripting language, with proper interfaces with other languages and integration of parallel and event-driven abstractions is the direction we have chosen. More and more, scripting languages are used as glue between other compiled languages. Python or Ruby are two famous examples. What these languages lack however for the moment is the necessary abstractions adapted for robotics, at least with a sufficient level of *usability*, as described above.

D. Simplicity

The above requirements generally tend to give birth to very complicated software architectures which are confusing and prevent the users to adopt them. Simplicity for *usability* is a requirement of both the modular component architecture and of the core platform technology in general.

It is of course hard to assess what is difficult and what is simple, it will highly depend on the level of experience of the user. This criteria can only be enforced by following some common sense principle: reuse notions that exist, limit the number of abstractions, build layered architectures where the complexity of the knowledge necessary to perform a task is proportional to the complexity of the task, etc.

It is however possible to measure the level of simplicity reached once the platform has been released by looking at the average user appreciation. This measure should be done by any platform developer to validate his approach.

III. Adoption strategies

Beyond the technical questions mentioned above, the question of the promotion of the platform is a generally underestimated problem. The issue is not only to have the platform adopted by the research community but also by robot manufacturers. Relying on standardization comities is a long and uncertain process, and releasing open-source version on the Internet is generally not enough to attract the proper attention on the solution, at least not in a short time. In fact, some marketing is required.

Active promotion and continuous development, maintenance and support are typically demanded by robot manufacturers. This support can be achieved by a gathering of universities granted with appropriate funding and political will, or by a private company spin-off from the lab and supported by the community. We believe that individual initiatives or small lab projects will not be sufficient to efficiently promote a universal software platform for robotics. The issue of ”adoption strategies” should not rely only on

the quality of the solution proposed, but deserves its proper attention and financing.

IV. The URBI Platform

The URBI platform, developed initially by J.C. Baillie[1] in the Cognitive Robotics Lab of ENSTA (Paris), is an attempt to design a universal robotic software platform that follows the above guidelines.

The URBI platform is based on a client/server architecture, built on top of a new programming language called URBI. It is possible to log into the robot with a simple telnet client and start to enter URBI code to control the robot and the software inside the robot. More advanced graphical and developer-oriented tools are also available.

We detail here the main characteristics of URBI. This is a very brief introduction, the reader is invited to get more information in [2], [1], [3].

A. Why a new programming language?

There are already so many programming languages, why should we add a new one to the list? The novelty of URBI is that it brings new abstractions to handle parallelism and event-based programming, directly integrated into the language semantics. A component architecture is also directly integrated in the language via a familiar object oriented approach.

In terms of the general look & feel (types, control structures, objects, etc), URBI is fairly similar to other scripting languages like Ruby, with a syntax based on C/C++ to make it as familiar as possible to new users (*usability* constraint).

1) Parallelism: Parallelism is not handled via threads or callbacks in the language, instead URBI introduces four command separators: the classical semicolon, and the new comma, pipe and ampersand. While the semicolon has its usual serial semantics, the other separators add more possibilities, including parallelism, to state how two commands should be executed. Figure 1 illustrates the four semantics.

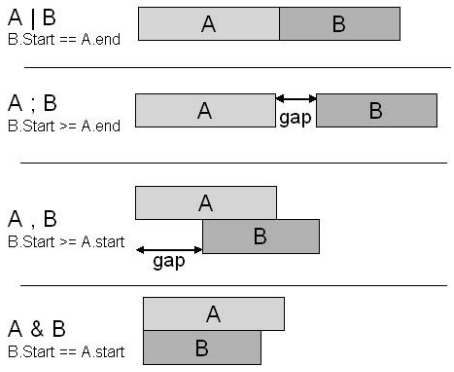


Fig. 1. Four types of command separators in URBI

Note that separators like pipe or ampersand are *strict*, since they enforce respectively an immediate serialization or parallelization of the commands. This can be used to precisely describe synchronization processes. Comma and semicolon however are *loose*, and the semantics allows for a gap to exist.

One can build complex serial/parallel constructs very easily. In the following code, C will start as soon as A and B have both finished:

```
{ A & B } | C;
```

Threads are not required as they are transparently handled by the URBI scheduler running in the language kernel interpreter, thus providing the right level of abstraction needed for parallelism.

2) Event-based programming: Another important abstraction in URBI is how events can be handled in the language. Consider the following code:

```
at (condition) action;
```

The action part will be executed once, each time the condition becomes true. Another event handler is `whenever`, which will execute the action in loops whenever the condition is true and as long as the condition remains true. These two commands are a bit similar to `if` and `while`, but they remain in background and will constantly monitor the condition to trigger the action if necessary.

It is also possible to emit events with parameters to send signals between codes running in parallel or between clients. The example below will trigger `action(42)`:

```
at (myevent(x)) action(x);

emit myevent(42);
```

We will see in the next section an example that combines parallel and event-based programming together with the component architecture.

B. The UObject component architecture

URBI is an object oriented language, so you can use and define objects as usual. What is more important is that you can also plug C++ objects into the language very easily, which is the base of the component architecture available in URBI. The C++ object will then be visible and usable as any other object in URBI; it is called a ”UObject”.

UObjects can be plugged, but they can also be unplugged and then ran remotely as autonomous executable programs, taking the address of the URBI Engine server as a parameter. This allows to integrates distributed object management directly in the language itself. For *usability*, note that, as we already explained, the C++ code of the UObject is the same whether it is plugged or ran in remote mode, it does not even need to be recompiled and it can be switched at runtime (relocatable objects).

CORBA, RT-Middleware or openHRP objects can also be seen from within URBI, like in fact most existing distributed object architecture, thus making URBI a central platform to integrate various technologies.

In summary: URBI plays the role of a central coordinator/hub for a set of plugged or remote heterogeneous objects, with parallel and event-based built-in capabilities. Shared memory and message passing can be done simply via the language structures.

Example

The example below can be used on the Aibo robot (there is a free version of URBI for Aibo available on www.urbiforge.org), and it will track the ball whenever the robot sees it:

```
whenever (ball.visible) {
  headPan.val += camera.xfov * ball.x
  &
  headTilt.val += camera.yfov * ball.y
};
```

`headPan` and `headTilt` are two UObjects controlling the head motors, and `ball` is a UObject detecting and localizing the ball in the image. Typically, `headPan` and `headTilt` will be plugged in the robot like "hardware drivers", while `ball` or other computer intensive objects can be running remotely outside of the robot. The UObject code is exactly the same in both cases (it is not even necessary to recompile it) and the way to use the object inside URBI is identical, which brings important features in term of transparent distributed object management and usability.

Note that the UObject architecture provides ways to get a notification whenever a specified variable is modified in URBI. For example, the `ball` UObject requests a notification on `camera.val`, thus making `ball` aware of any new image to process.

This simple example shows also how URBI enables to control motors by assigning the 'val' attribute of a motor object in the language. This is very simple and can be used in educational applications.

The UObject components in C++ have been designed with simplicity in mind. A limited set of small changes is necessary to port an existing C++ class into a UObject. Extensions to Java and C# are in development, to extend the *acceptability* of the feature.

C. Advantages of the client/server approach: liburbi

The client/server approach allows to interact with URBI through a socket connection, from any existing programming language. Liburbi is a set of GPL libraries for various languages like C++, C, Java, Matlab, Ruby, Python and others, that allows to create connections to an URBI Engine, send commands and receive messages asynchronously. Liburbi is designed with simplicity in mind, as a proxy to the URBI Engine and it does not add new abstractions or complexity to the platform, which is essential for *usability*.

In the most degraded case, URBI can be seen as a simple driver for the robot hardware, and can be used with liburbi from within any programming language to provide a common interface to any robot.

D. Advanced features for robotics

In addition to parallelism, event-based programming and the object oriented UObject component architecture, URBI brings several unique features that we briefly review here. These features are direct consequences of the parallel nature of the language and would not be semantically well defined otherwise.

A simple assignment in URBI can target a variable to reach a value in a given time or at a given speed, or set a sinusoidal oscillation on it. The assignment is not instantaneous anymore and can be run in parallel with others:

```
  neck.val = 10 time:450ms
& leg.val  = -45 speed:7.5
& tail.val = 14 sin:4s ampli:45;
```

Variables have a blend mode which specifies how conflicting simultaneous assignments should be handled - an extension of the concept of 'mutexes':

```
x->blend = add;
x = 1 & x = 3;
//now x equals 4
```

Any portion of code can be prefixed with a tag. It is then later possible to stop, freeze, unfreeze this code from anywhere using the tag name, which brings powerful features to control the flow of execution of parallel codes:

```
mytag: { some code };
stop mytag;
freeze mytag;
unfreeze mytag;
...
```

Hierarchical tags and multi-tags are also available.

More details about the advanced features of URBI can be found in the tutorial or reference manual [2].

E. Limitations

Currently embedded versions of URBI are limited to robots who have a reasonably powerful CPU embedded (typically ARM7 or more). It is written in C++ and must be compiled for the hosting platform. This limitation is counterbalanced by the fact that most robots can be controlled remotely via a serial link or wifi connection, URBI running on a PC or a Mac on the side. The general trend in robotics is also to have more and more linux based robots with sufficient CPU onboard.

Another limitation of URBI is the need to develop a set of hardware UObjects for each new robot. This is similar to the problem of driver development. Most of the time however, the robot comes with a simple C API that can be wrapped inside C++ UObjects in a few days. URBI is not an Operating System and will rely on existing interfaces to the hardware, thus limiting the driver development problem.

The real time aspect in URBI is dependent on the underlying operating system. If the OS is capable of real-time scheduling, then putting URBI in the highest priority

will allow to guarantee a first level of real-time features in URBI (at the level of URBI commands, not plugged UObjects). The second stage is to include in URBI some real-time oriented features, like priority flags for commands. This is in development for version 2. Finally, work has to be done on integration of UObjects in the real-time framework, and integration with existing real-time platforms like RT-Middleware.

Finally, the current limitation of URBI is its relatively limited adoption and youth. There are currently nine different robots that run URBI and one simulator (Webots), and about 20 to 25 universities that use it on a daily basis. One spin-off company of our lab is currently promoting URBI towards robot manufacturers to extend its coverage.

V. Comparison of URBI with existing platforms

URBI has been designed with the four above key constraints in mind: flexibility, modularity, parallel abstractions for robotics, and simplicity. Other attempts are currently made in the same direction and we will shortly review some of the most significant here.

A. Player/Stage

Player/stage is a client/server based platform built on top of C++. It is widely used in mobile robotics applications, in particular with the Pioneer robots.

As a C++ library, it does not bring new abstractions in term of parallelism or event programming: the user must use threads and an event loop of his own. Flexibility is also limited by the choice of a specific programming language, but the client/server architecture enables in principe other language interfaces, with a question mark on *usability*.

There is no distributed component architecture in player/stage, except C++ built-in objects.

B. Microsoft Robotics Studio

The recent announcement by Microsoft to offer a universal robotic interface has been perceived as a very good sign of the maturity of the robotics industry. It represents a serious effort in the right direction for more advanced abstractions for robotics, supported by a major software actor capable of active promotion. It is however relying on a .NET architecture which needs Microsoft Windows either on the robot or on a remote computer, raising questions in terms of flexibility. Generally speaking, several users have reported that MRS remains relatively complex to master for the moment.

C. CORBA

CORBA, the distributed component architecture developed by OMG, provides much flexibility as it has been ported to many type of languages. CORBA has no particular abstractions available for parallelism, except the object level parallelism, and events are handled with a usual asynchronous object message passing mechanism.

One of the limitation of CORBA is that it is perceived as a complex solution and it lacks of a central coordination mechanism to interface the different CORBA objects. The same applies to RT-Middleware currently developed in Japan.

Note that since URBI allows to dynamically create methods on an existing object at runtime, it is possible to create a UObject that will connect to a CORBA or RT-Middleware object, read the IDL and create appropriate hooks as URBI object methods, to transparently interact with the remote object. Effectively, this allow to transparently access CORBA or RT-Middleware components, seen as regular objects in URBI.

D. Others

Many others platforms exist, like Tekkotsu [6], Marie [7], Orocos [11], Orca [12], ERSP [13] or Pyro [14]. Some are based on a C++ library approach and rely on a rather complex architecture. Some others, like Pyro, are notably more simple and based on python, but do not include native abstractions for parallelism or distributed objects at the moment.

While each of these platforms brings its own clear benefits, and do a great job in their specific domain, none of them at the moment fits altogether with all the four requirements that we have detailed.

VI. Conclusion

While there is still no clear leader in the field of universal robotics software platform, the number of candidates is increasing, like the number of robots. Reminding us of the beginning of the 80's and the Personal Computer revolution, it is likely that one major platform will stand out in the future. But unlike computer users in the 80's, robot specialists and end-users are today more knowledgable and aware of the importance of design principles in order to have a successful, flexible and adapted platform. We have presented four key guidelines that should drive the development of such a platform: flexibility, modularity, parallel abstractions for robotics, and simplicity. We have claimed that each of these characteristics is necessary and should be acceptable and usable to make it suitable in a universal platform. Finally, we have introduced the URBI platform which is the result of our efforts towards such a universal platform. We also summarized briefly some others candidates by comparing their approach to the four criteria that we have stated.

URBI is currently available for free on www.urbiforge.org, with documentation and a user forum. It is available in particular for the Aibo and Mindstorm robots. We hope that URBI can contribute to the development of robotics in research and in the industry.

References

[1] J.C. Baillie, "URBI: Towards a Universal Robotic Low-Level Programming Language", *in Proceedings of IROS'05*, 2005, pp 820-825.
[2] J.C. Baillie, "The URBI Tutorial" *www.urbiforge.org/tutorial*, v 1.2, 2007.
[3] urbiforge, "URBI Technology website, with documentation and software exchange platform " *www.urbiforge.org*, 2004-2007.
[4] G.M.Biggs and B.A.MacDonald, "Specifying Robot Reactivity in Procedural Languages", *in Proc. IEEE/RSJ Int. Conference on Intelligent Robots and Systems*, 2006.

[5] G.M.Biggs and B.A.MacDonald, "A survey of robot programming systems", *in Proceedings of the Australasian Conference on Robotics and Automation, CSIRO*, Brisbane, 2005.

[6] David S. Touretzky and Ethan J. Tira-Thompson, "Tekkotsu: A framework for AIBO cognitive robotics", *in Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, 2005, Menlo Park.

[7] Ct, C., Brosseau, Y., Ltourneau, D., Raevsky, C., Michaud, F., "Robotic Software Integration Using MARIE", *International Journal of Advanced Robotic Systems - Special Issue on Software Development and Integration in Robotics*. Vol.3, No.1, 55-60.

[8] Richard T. Vaughan, Brian Gerkey, and Andrew Howard, "On Device Abstractions For Portable, Resuable Robot Code", *in Proceedings of the IEEE/RSJ International Conference on Intelligent Robot Systems*, 2003, pp 2121-2427.

[9] UNECE - United Nations Economic Commission for Europe, "2004 World Robotics Survey", *www.unece.org/press/pr2004/04robots_index.htm*, 2004.

[10] Sony Corporation, "Open-R SDK for Aibo robots", *www.openr.aibo.com*, 2005.

[11] OROCOS, "Open Robot Control Software", www.orocos.org.

[12] A. Makarenko, A. Brooks, T. Kaupp, "Orca: Components for Robotics". *IEEE/RSJ International Conference on Intelligent Robots and Systems*, (IROS 2006) Workshop on Robotic Standardization

[13] Evolution Robotics, "ERSP: http://www.evolution.com/products/ersp/"

[14] Blank, D.S., Kumar, D., Meeden L., and Yanco, H., "Pyro: A Python-based Versatile Programming Environment for Teaching Robotics.", *Journal of Educational Resources in Computing (JERIC)*.

[15] A. Mallet, S. Fleury and H. Bruyninckx, "A specification of generic robotics software components: future evolutions of GenoM in the Orocos context", *in International Conference on Intelligent Robotics and Systems*, Lausanne (Switzerland), 2002.

[16] John K. Ousterhout, "Why Threads Are A Bad Idea (for most purposes)", *Talk at the 1996 USENIX Technical Conference*, January 25, 1996, http://home.pacbell.net/ouster/threads.pdf

List of speakers

Name	Institution	Title
F. Ingrand	LAAS	*Activities in the « GDR Robotique »*
P. Curlier	SAGEM	*The EUROP Network*
J.L. Farges	ONERA	*Mission Management System for Package of Unmanned Combat Aerial Vehicules*
G. Verfaillie	ONERA	*A generic architectural framework for the closed-loop control of a system*
G. Haik	Thalès	*Really Hard Time Developing Hard Real-Time: Research Activities on Component-based Distributed RT/E Systems*
N. du Lac	INTEMPORA S.A.	*The RTMaps platform applied to distributed software development*
C. Séguin	CERT	*Formal assessment techniques for embedded safety critical system*
L. Petrucci	LIPN, Université Paris XIII	*A formal approach to designing autonomous systems: from Intelligent Transport Systems to Autonomous Robots*
J.-C. Chaudemar	SUPAERO	*Z and ProCoSa based specification of a distributed FDIR in a satellite formation*
F. Ingrand	LAAS	*Incremental Construction and Verification of Robotic System using a Component-Based approach*
X. Blanc	MoVe, LIP6	*Benefits of the MDE approach for the development of embedded and robotic system*
R. Passama	OBASCO, École des mines de Nantes	*A modelling language for communicating architectural solutions in the domain of robot control*
S. Stinckwich	GREYC - Université de Caen	*On Adaptive Robot Control Architectures*
J.-C. Baillie	GOSTAI	*Design Principles for a Universal Robotic Software Platform and Application to URBI*
G. Sassatelli	LIRMM	*HS-Scale: a MP-SOC Architecture for Embbeded Systems*