Upgrading the ContrACT Scheduler with Useful Mechanisms for Dependability of Real-Time Systems

Robin Passama, David Andreu, Bertrand Brun

Upgrading the ContrACT Scheduler with Useful Mechanisms for Dependability of Real-Time Systems

Bertrand Brun David Andreu Robin Passama

LIRMM, CNRS and Montpellier II University
161, rue Ada
34392 Montpellier Cedex 5 France
{brun,andreu,passama}@lirmm.fr

Abstract

Nowadays, current research in the development of a robot focuses on ensuring the safety of persons around the robot and on resilience to recover from errors. As part of the general problem, managing real-time aspects is of main importance regarding "fault tolerance" in robotic systems, to monitor and adjust control scheme execution or to guarantee execution of critical behaviors prior to less important ones. In this context, the paper describes our current work on the ContrACT middleware [10] to enhance its real-time scheduling service for instance to provide to users some means to observe and potentially react to temporal errors occurring during periodic loops execution.

1 Introduction

Robotic software is now one of the essential parts of robotic system development, therefore software control architecture design methods and concepts, often inspired by software engineering community, are necessary within a robotic project to enhance evolution, modularity and re-usability, and to avoid redesign costs [1]. In this context, we already proposed a control architecture engineering methodology, named ContrACT, and associated tools and middleware to improve such characteristics [10]. The central piece of ContrACT based architectures is the scheduler module, that allows for a fine grain decomposition, control and reuse of periodic schemes and/or algorithms.

Nowadays, current research in the development of a robot focuses on ensuring the safety of persons around the robot and on resilience to recover from errors. Some improvements of the ContrACT methodology to treat "dependability" aspects have been proposed [4], but one main concern is that ContrACT lacks for efficient middleware services to precisely observe temporal errors and to adapt the periodic behaviors adequately. Fortunately, since the scheduler module has been reified in the architecture, we have a great liberty to implement new services to achieve such a general objective.

In this context, this paper focus on our current work, whose aim is to provide new mechanisms inside the scheduler module so that the roboticians can have a better control over real-time aspects. We want to add new features to the scheduler:

- to adapt its execution regarding context,
- to guarantee some functional constraints (e.g. important schemes are executed prior to less critical ones),
- to allow for fine grain definition and observation of temporal events and,
- to provide supervision level some means to dynamically adapt the periodic behaviors used at runtime.

The scheduling algorithm is the cornerstone of the scheduler module. It ensures execution of periodic schemes given a set of real-time constraints. "Adapting scheduler execution regarding context" does not mean to directly improve this algorithm but to adapt some parameters it is based on in order to optimize its execution in function of periodic schemes executed at a given time of the robot mission. This can be achieved by observing periodic schemes' duration at runtime and by doing an online adaptation of schemes's related parameters according to well-defined rules.

One important concern, currenlty not managed into the scheduler module is the ability to define periodic schemes with different criticity levels ; a periodic scheme being an ordered subset of modules to be executed in the current context. Indeed, low level control (for instance sensor sampling or actuator command) often needs to be guaranteed before any other functionality can be executed or security command schemes (e.g. obstacle avoidance vs. trajectory following). The modification of the scheduling algorithm is in this case necessary to manage different levels of criticity among schemes.

It is interesting for roboticians to observe not only if but how their robots execute their behaviors. So we first need to implement an observation system allowing the decisional level of architecture to decide to take action if deemed necessary (ie according to observations that have been notified and rules of reaction that have been defined). For instance if a control law does not respect its timing time constraints, it may produce unstable behaviors, behavior which can be very risky for the robot. Supervision system must be notified of such situations to potentially perform an adequate reaction before any reaction can take place. Furthermore, depending on the periodic scheme implementing a given behavior, temporal errors can be more or less significant. So the precise observation of the periodic schemes execution is very important to characterize the current execution context of a robot, it is a mandatory feature before any adjustment of the behavior can take place.

Jointly to observation of temporal errors, a control levers on schemes execution are necessary to be able to react when they occur. The most important question to answer to is: what to do in case of faulty scheme? There is no unique solution since it depends on the nature of the scheme and its execution context. Nevertheless, depending on the nature of the algorithms (modules) used, a scheme overpassing its delay is more or less foreseeable and/or acceptable. For instance, an algorithm such as a reactive trajectory planner can have sometime long computation time compared to its mean computation time, but this should be acceptable in some execution contexts and maybe not in others. In the first case we can let it continue to make it end its cycle, while we could want to immediately interrupt it in the other case. ContrACT scheduler should be able to manage such situations.

We also need to implement a greater level of control of schemes execution. If this control is already possible in ContrACT scheduler by simply changing the schemes it schedules, the problem becomes harder when several simultaneously executed schemes are in interaction (they share part of their inputs/outputs), which is in fact the general case. Indeed, changing a periodic scheme may result in many reconfigurations of data flows between schemes: for instance commuting from a cartesian control law to another one (more precise, less time consuming, etc.) will also result in reconfiguring (parts of) the inputs (set point flow) and the outputs (sensor sampling flow) of the lower level articular control law. In the context of ContrACT, where schemes are decomposed into several modules, managing the correct reconfiguration of data flows can be an error prone scheme. That is why we need new mechanisms to easily define and control these situations.

The following sections of this paper will present all these points. First, Section 2 gives an overview of ContrACT and discuss about its limitations regarding observability, controllability and adaptability. Then Section 3 presents the improvement the scheduler module regarding its current internal limitations and Section 4 focuses on the evolution of periodic schemes themselves. Finally, Section 5 concludes this paper and discuss about future work.

2 Presentation of ContrACT

2.1 Architectural model

The programming model of ContrACT relies on the concept of module. A module is an independent (with it own context) real-time software task that reacts to a set of predefined requests and communicates with other modules by the means of ports (for more detail see [10]). Each port is a connection point of a module by which it can send or receive data or events. A port is defined by its data type, its name and its direction (input or output). Ports connections are made by binding one output port to one to many input ports, and connection is checked by verifying types of exchanged data. There are two kind of ports using two different publish/subscribe mechanisms : event ports (receiver is triggered) and data ports (receiver is not triggered). In a ContrACT architecture all modules are software tasks, but different classes of modules, and even some specific ones, derived from the previous module model are defined (see Figure 1):

- Synchronous modules are used to implement periodically computed algorithms and periodically performed sampling of sensors and command of actuators. They communicate between each other only using data flow communication.

- The scheduler module is the unique module implementing the applicative scheduling services. Its main role is to schedule synchronous modules according to a set of constraints defined on the module itself (duration of the execution) and schemes (i.e. on composition of modules): precedence constrainst, critical delay, period. To achieve this job it handles OS priorities (managment of preemption) and sends activation requests to synchronous modules. The second role of the scheduler module is to notify scheduling events to supervisors or asynchronous modules.

- Supervision modules are implementing reactive supervisory control in the architecture. There may be many supervisors but only their configuration change, not their implementation. Supervisors can be seen as specific asynchronous modules that control the execution (asynchronous or scheduled execution), the assembling (data or event communication) and the parametrization/consultation of modules according to events they receive. They can so receive events from synchronous and asynchronous modules but also produce events to other supervisors.

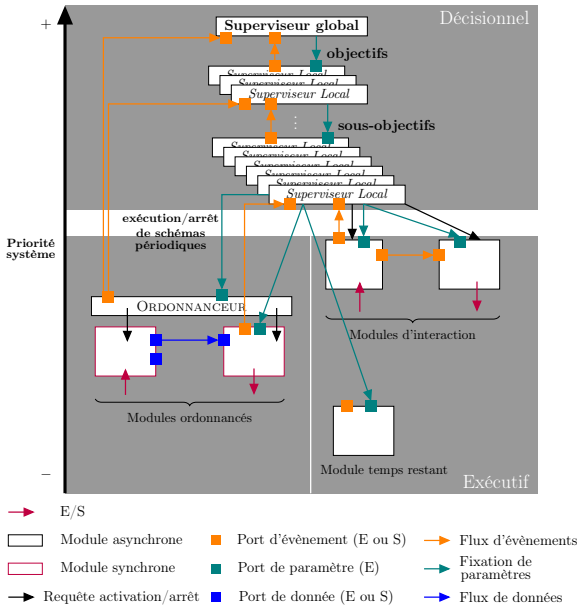

Figure 1: ContrACT architectural model

2.2 The applicative real-time scheduler

The applicative real-time scheduler is embedded in a specific module of the middleware, dedicated to real-time aspects management. This scheduler manages the execution of synchronous modules, dealing with periodical schemes. A periodical scheme describes an ordered sequence (composition) of synchronous modules to be executed (see Listings 1 to have an example) ; modules are ordered to respect causal relationship between them (described in `scheduling graph`), according to the usual "perception-decision-action" process. The applicative real-time scheduler controls the execution of synchronous modules (their corresponding tasks) above the scheduler of the real-time OS. To do so

it sends activation request to considered modules and plays with priorities it dynamically associates to the corresponding tasks according to the selected scheduling policy [7]. The scheduling policy used in ContrACT is Earliest Deadline (ED). ED algorithm gives the highest priority to the task which has the earliest deadline. The scheduling problem has been formalized taking into account two kinds of tasks: treatment tasks which contain an algorithm that will be executed on the processor, and acquisition tasks that are two-part algorithm which first sends a request to a device (e.g. sensor, communication link) and then waits for the response (to get the data). Constraints between tasks are described in terms of task ordering within a given scheme and tasks mutual exclusion (e.g. tasks of different schemes using common or exclusive resources, like interfering sensors for example) [6]. There is no pre-running validation of the scheduling setup to anticipate CPU saturation, the behavior of the scheduler consist in launching a scheduling setup and doing an online reporting to supervisors each time realtime constraints are violated. Thus the execution of a scheme will consist in the execution of a sequence of periodic modules until the end of the scheme. This suite of modules will begin its execution in the reception of the scheme by the scheduler and will have to end before the date specified by `CRITICAL_DELAY` while following the policy explained above. This sequence will be repeated with a period defined by `PERIOD`.

This real-time scheduler enables a fine grain decomposition of complex robotic algorithms (typically control loops) into individual real-time modules. Doing so, users obtain a better reusability for these algorithms because they can be composed differently according to the global algorithm to put in place. Furthermore, this approach allows to manage precisely real-time constraints execution and reaction to constraints violation, what is not possible when the global control loop is implemented as a periodic task of the real-time OS.

2.3 Limitation of ContrACT

Currently, scheduling algorithm's performance is strongly coupled with the nominal duration of synchronous modules, but these values are set quite empirically. This solution is maladjusted because either this duration is an overestimation and thus the system is sub-optimized (because the scheduler considers that the module is working during all this period), or this solution is sub-estimated and the scheduler will generate many notifications. The basic needs are:

- doing an online optimization of the scheduling parameters without loosing stability (regularity) of periodic loops. One big constraint is to avoid too many time parameters adjustements, otherwise the risk is that the scheduler module does an excessive number of reconfigurations and so "waste" too much CPU time.
- allowing the developpers to control more deeply the notification of scheduling events, in order to better discriminate situations and react only to important ones. For now the scheduler module only notifies a "scheduling invalid" event when a module excessively exceeds its nominal duration or when a scheme repeatedly exceeds its period but nothing can be configured by a user. The consequence is that the user cannot limit the generation of events to the situations he wants to handle.

The ContrACT scheduler notifies events to the supervision level, but being simply informed of temporal errors occurence is an information too low to discriminate implicated modules among all active schemes. For instance, we can not know if the error comes from a long calculation or of a module that loop indefinitely. Thus we want to distinguish the faults of the modules (critical delay outdated) and of schemes (periodicity impossible to respect by example, duration of modules $>$ the period of scheme). We also want to identify precisely which modules and schemes involved and to define the "severity" of the fault.

As explained in section 2.2 a prerunning validation of the schedule setup is not done, to date, to anticipate CPU saturation. An "a priori" verification would allow to avoid structurally faulty situations (i.e. situations when scheduling parameters values

are intrinsically not coherent, resulting in a configuration that cannot be scheduled). The big interest at runtime is to avoid too many unecessary notifications by producing a single specific event. The contextually faulty situations (i.e. situations when modules execution time spontaneously derives for any reason) would be so the only ones managed with the default mechanism (see previous paragraph).

Another limitation is that all schemes are "functionally equals", therefore it is impossible to define those that are considered "critical" (control law for the securing of the system) and that must run before. For instance, we might want to maintain an observation loop (for instance to store the articular positions of a tele-echograph robot in order to pair them with ultrasond images memorized) while controlling the robot. This loop can have the same frequency that the loop of the probe by example. If the processor resources are too limited, the articular control loop and the probe control loop must take priority to the observation loop. Nothing allows it at present.

And finally in the case of the multischeme (see Figure 2), when one wants to make two or more schemes interact between each-others, modules contained in schemes have to be correctly bound. This operation requires that developers know modules inside the schemes which is against reuse principle.

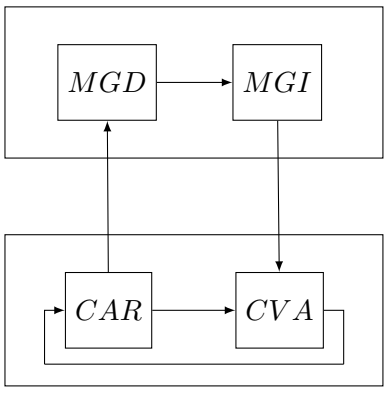

Figure 2: Example of two interacting periodic scheme used in the PROSIT robot control architecture

We present with listings 1, 2 and 3 the implementations of Figure 2. In this example we create two periodical schemes (Listing 1 and 2). Then when the `doLocalArticularAndEulerPositionRobotControl` is call it execute the two rules which realized the connections and activate schemes. We see that into the supervisor we realize two others connections (`subscribe modules CAR:0 -> MGD:0 * 5;` and `subscribe modules MGI:0 -> CVA:0 * 1;`) to connect schemes between them.

```
periodic schema doArticularControl(){
  realtime{
    PERIOD=10ms;
    CRITICAL_DELAY=4ms;
  }
  modules{
    CAR();
    CVA();
  }
  scheduling graph{
    CAR -> CVA;
  }
  communications{
    CAR:0 -> CVA:1 * 1;
    CVA:0 -> CAR:0 * 1;
  }
}
```

Listing 1: The scheme doArticularControl which manage the articular loop

```
periodic schema doSondControl(){
  realtime{
    PERIOD=50ms;
    CRITICAL_DELAY=50ms;
  }
  modules{
    MGI();
    MGD();
  }
  scheduling graph{
    MGD -> MGI;
  }
  communications{
    MGD:0 -> MGI:0 *1;
  }
}
```

Listing 2: The scheme doSondControl which manage the sond

```
supervisor SCR{
  :
  :
  doLocalArticularAndEulerPositionRobotControl
      (){
    rules{
      ROBOT_ARTICULAR_POSITION_CONTROL:
      [elapsed=1ms]
          activate schema doArticularControl()
      [endof(ROBOT_ARTICULAR_VELOCITY_CONTROL)]

      ROBOT_SOND_POSITION_CONTROL:
      [elapsed=200us]
          subscribe modules CAR:0 -> MGD:0 * 5
              POSITNOW;
          subscribe modules MGI:0 -> CVA:0 * 1;
          activate schema doSondControl()
      [endof(ROBOT_ARTICULAR_POSITION_CONTROL)]
    }
  }
  :
  :
```

```
}
```

Listing 3: The supervisor which connect schemes in a multischeme instance

Even more complex is the management of reconfiguration of data flows when some scheme change. It requires that supervisors adequately reconfigure these flows, which can be an error prone task. For instance, in figure 2 (extracted from ANR PROSIT project) direct geometric model's (MGD) input flows of the probe referenced control scheme is bound to CAR module's (sensor/actuator access) output flows or the articular control scheme. If the probe referenced control scheme changes the disconnection of CAR module with the MGD module and eventually its reconnection with the new one used has to be redefined. Considering that these reconfigurations can be requested by different supervisors and applied to many interacting scheme (in the general case) the situation can become really "uncontrolled" in terms of complexity for the developers. So, we need to provide new mechanisms to automate as far as possible the reconfiguration of flows between schemes, while of course keeping consistent links between them.

3 Proposed scheduler ameliorations

3.1 Adapting duration parameters of modules

During an execution, the execution period of different ContrACT modules can vary. For example, particle filter is an algorithm which estimates Bayesian models per sampling methods. As the particules number is more or less important during execution, and considering that this number greatly conditionates the duration of a module, its duration can vary in an important interval during runtime.

It was planned in Cotama architecture (ContrACT's predecessor) to adapt the execution times in the modules using the following formula [5]:

$$val_{estimated} = average + 1,5\sigma_n \quad (1)$$

This formula computes "safe" average.

When a user defines a module he must emprically specify its nominal execution delay. This will lead, as we previously said, to a sub utilization of the system, because the nominal execution delay may be far longer than the mean execution delay. Being currenlty "lost" (not used by the scheduler), the time interval between both delays could benefit to another module.

The basic need to be able to do so is to set up a system of statistics on the execution of modules, and that is what we have primilarly implemented: the scheduler records execution runs of each synchronous module (start stop dates) over a sliding temporal horizon and incrementally compute simple statistics (means, standard deviation). From these statistics, we get the information useful to automatically adjust execution times, but also useful to the decision-making level (for instance to decide if the robot has to be put in emergency or degraded modes).

Concerning a global dependability manament system, the scheduler can obviously only allow the consulting of these statistics and cannot do more "reasonning", first for efficiency reasons, second for respecting separation of concerns. The dependability management system should include specific modules to do so, for instance as explained in [3].

After the observation of the evolution of executions durations of the modules. We propose here an algorithm that will allow to adapt the execution times of the modules at runtime. It's not really an action on which supervisors can interact. But this is still a control action on the scheduler modules.

To achieve this adaptation, we use the formula (1) presented in [5] to compute the execution time of the module using the various statistics collected throughout their performances.

Algorithm 1 presents how this adaptation is performed.

- First, the decision of adaptation is taken only every X cycles (where X is either computed or

given directly "in hard", as actually).

- Second, the computed tendency of a module execution defines if this duration time strongly derives from the mean time.
- Third, the last test is a kind of filter : the adaptation of current execution time ($M.running\_time$, algo 1) will take place only if the "securized mean" (M.avg, computed according to formula 1) defines a sufficient variation around current execution time value. This variation is defined using a threshold delay (Δ) that applies above or below current execution time.

Algorithm 1: Algorithm determining when to adapt, 3rd version

```
Data: M module to adapt
if M.counter ≥ X then
    if M.trend ↗ or M.trend ↘ then
        if M.avg > M.running_time + Δ or
        M.avg < M.running_time − Δ then
            adapt
    M.counter ← 0
else
    M.counter ← M.counter + 1
```

3.2 Priority levels for the schemes

During the evolution of a robot in its environment, it may have to enter into an emergency behavior for various reason. Such a behavior is implemented with one or more schemes, that consequently must run in preference to schemes used to implement other less critical behaviors. From this observation it brings up a need to establish a system of priorities between schemes.

We initially thought to establish two levels of priority as in [8]. But we do not know if using two priority levels will be sufficient for a given application. So we have instead kept the solution of hierarchical levels of priority. We so allow the users to define oriented relationships between schemes, according to a priority constraint. If we have two schemes A and B, A with higher priority than B, then we write $A \rightarrow B$. From these constraints we defined a graph (that is not necessarily made of connected components, see fig. 3).

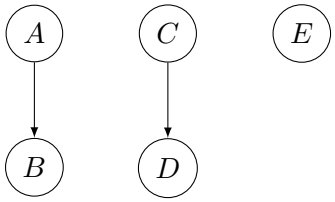

Figure 3: Relationship between five schemes

In figure 3, readers can remark that a scheme can have no relation with the others. If two schemes have no direct or undirect relationship, it may mean that they will not be simultaneously active or that they apply to completely disjoint controlled systems (arms vs legs). However, there can't be no cycle in the graph. Thus we can't write $A \rightarrow C$, $C \rightarrow D$ and $D \rightarrow A$. It won't have a meaning.

The graph of relationship defines relatives priorities between schemes. Nevertheless, the implementation of schemes priority management requires global priorities for schemes. Indeed the internal algorithm is based on static levels of priority to avoid any ambiguity at runtime or any extra unnecessary computation. To compute global priorities for schemes, we create a vertex "source" that is connected to all the roots of the schemes relationship graph, as shown in Figure 4.

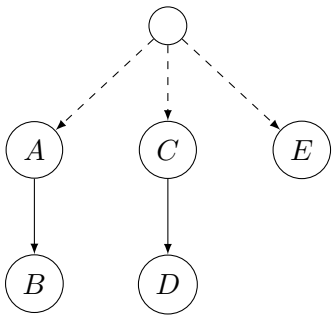

Figure 4: Adding the source vertex in our graph

Once the tree built, we perform a breadth-first search (BFS) to compute the distances of nodes from the "source". These distances represent global priorities of schemes (see figure 5).

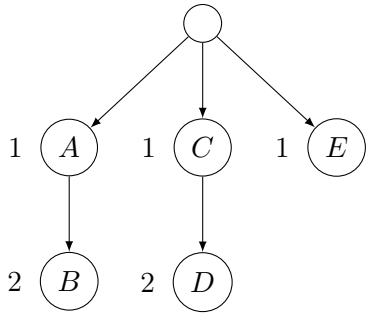

Figure 5: Relationship between tree height and priority level

On the graph 6, we have the scheme C with a distance of 2 because the BFS algorithm looking for the minimum distance from a vertex (here, "source" vertex), but it should be 3 because user had specified $B \rightarrow C$. The conflicts comes that we have a transitive arc AC. This is why, we propose algorithm 2 to reduce the transitivity. This algorithm is performed before the calcul of the schemes priority levels.

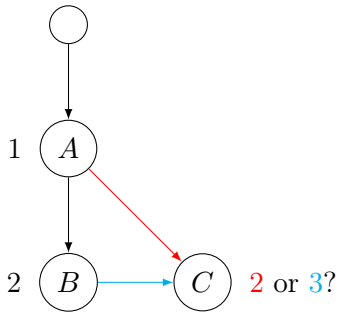

Figure 6: Conflicting priorities when $A \rightarrow C$ and $B \rightarrow C$

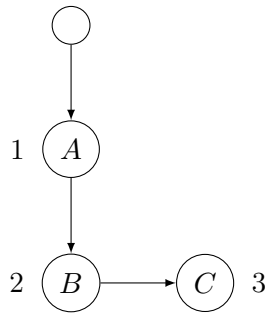

Figure 7: The changes on the figure 6 after a pass of algorithm 2

The research for the levels of priorities of schemes is realized during the phase of compilation of the project. When the scheduler has to choose a module to be executed, it will begin by looking for the module with the strongest priority (thanks to the algorithm ED) but in the schemes of stronger priorities (those who have the smallest value in our figures). If the scheduler does not find modules in the list of the schemes of stronger priority, then it will look in that just down and so on until find a module to be executed.

Algorithm 2: Removing transitivities

Data: G the graph of priorities
Data: s the starting vertex (the source)

$q = CreateQueue()$
$setParent(s, s)$
$mark(z)$
$Enqueue(q, s)$
while $\neg Empty(q)$ **do**
  $x = Dequeue(q)$
  while $haveChild(x)$ **do**
    $z = nextChild(x)$
    $setParent(z, x)$
    if $\neg mark(z)$ **then**
      $Enqueue(q, z)$

3.3 A system for controlling the observation of and the reaction on temporal events

In the current version of ContrACT scheduler, the temporal event detection mechanism is based on nominal delays of modules, which are, in practice, an overestimation of the real execution delays. Due to the automatic adaptation of modules duration in the new scheduler version (cf. section 3.1), we no more need a nominal delay except for initialization purposes. Nevertheless, we still want to define a temporal constraint on a module, either for checking scheme schedulability (see section 3.4) or for temporal event generation. The temporal constraint of a module can be defined as "the maximum amount of time the module can execute before being considered as faulty". We assume that each module defines a

default temporal contraint and that each schema can redefine the temporal constraints of its modules.

From this start point, it is important to classify possible temporal events and to reify them as simple concepts. We classify temporal event into three levels (cf. figure 8), from the less to the most "critical" one : delays, faults, exceptions.

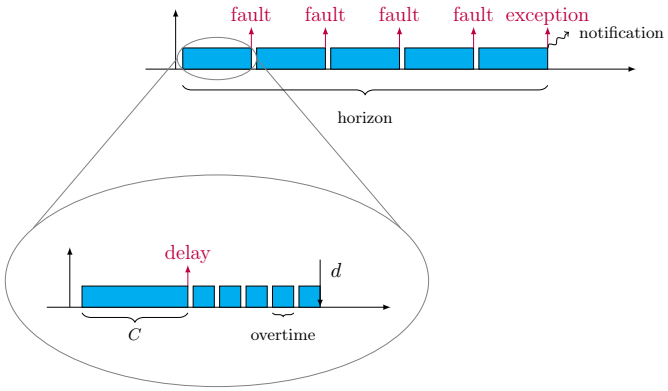

Figure 8: Diagram presentating the concepts of Delay, Fault and Exception

The **delay** is the first level. A temporal delay occurs when a run of a given module is longuer than its adjusted "secured" duration. Regarding delays management, decisionnal level has no mean to act on this type of events. The scheduler computes additional time given to a module based on its temporal constraint value (max, see formula 2) and the adjusted duration ($adapt$, formula 2).

$$add\_time = (max - adapt) \times \frac{1}{delay\_allowed} \quad (2)$$

$delay\_allowed$ is the maximum numbers of time which we grant additional time to the module. It is among five in the architecture.

This approach gives additionnal time to modules proportionnaly to their remaining delay until reaching their temporal constraint. In other words, a module with a bigger remaining execution time will have of course more additional time but it will generate at maximum as many temporal delays. We think this is a reasonnable approach to avoid too many unnecessary tasks switches (between scheduler and late module), because the delay is also an event that awakes the scheduler.

The **fault** is the second level of temporal event. When the scheduler detects an excess large number of delay for one module (defined by $delay\_allowed$ in equation (2)), then the module is in temporal fault. Unlike delay, the temporal faults can be subject to an action by the user. Indeed, when specifying the scheme it can choose among three types of reactions:

- The module continues its execution, as if nothing had happened. This reaction is useful for managing modules using some class of algorithms that are known to be temporally unbounded or with big delay variations.
- The module immediately interrupts (abort and restart from beginning at next start) its execution. This reaction is equivalent to an Abort strategy in [2]: a STOP request is sent to the module that consequenlty stops as quickly as possible. This strategy is so related to the management of what is commonly called "hard real time tasks".
- The module will be interrupted just before next run if not terminated yet. This is the same behavior as previously but delayed to let the module the possibility to execute in rest time if possible.

Other strategies could be imagined in the future.

The **exception** is the last level. A temporal exception characterizes an important situation that must be reported to the decisionnal layer. This situation is determined in function of a repartition of temporal faults among a memorized temporal horizon. Currently, this repartition is really simple since we define the number of authorized faults over a given temporal horizon (in number of cycles). If this condition is broken the scheduler sends a notification to the supervisors that asked for it. As we presented in the section on observability, we trigger an temporal exception by a combination of errors over a cycle time. This combination allows us to define rules. According to these rules we can inform the decisionnal layer if an exception occurred, and the decisionnal layer can dynamically configure the scheduler to

adapt temporal exception generation to the context. For example, the user can define a "hard real time" rule with a couple mistake time equal to 0 at any horizon. In contrast, it may define a rule with 10 faults on a time horizon of 30 cycles to behave much more flexible. With these notifications, supervisors are notified when one of their modules does not meet these time constraints. We could add to the system mentioned above more features for instance:

- to define the repartition of faults in terms of successives errors over a given horizon.
- to anticipate future tendencies.

Via these new concepts and mechanisms we try to provide a frame to detect various situations:

- Blocking module: normally, it should be detectable from the time when there will be a temporal succession of faults that will systematically cause a temporal exception.
- Running an unusual branch of the very long calculation: this may generate some errors on some temporal cycles, but, a priori, should not generate exceptions, unless the module is in real time and in strict if this behavior is quite normal.
- Preemptive sporadic supervisors or other modules: this is the same case as for the execution of a long branch. The problem is especially stringent for modules. Indeed, supervisors will inevitably run and they will bring this kind of problem.
- Adapting too vigorous execution time: it should, a priori, only lead to delays and faults that should be reabsorbed in the next adjustment.

3.4 Checking the schedulability of the scheme

In this section we will look at various verifications that can do the scheduler to ensure that the modules meet their temporal constraints.

A computation of the new CPU load is put in place each time a new scheme is inserted in the schedule and a verification process can then apply: if the CPU load is too high then some scheme have to be removed from the schedule. This is the way we can quickly test that the schedule is coherent. To compute the CPU load we use the formula (3). There are other formulas for calculating this charge, but this one has the advantage of being simple and thus to provide a result quickly.

$$U_n = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{3}$$

Rather than using the value of 100% load as the threshold, we will instead use a value, defined according to the experiments. Indeed, a CPU load of 100% computed by our formula can correspond to a load of 110% in reality, since there are also the activities of the modules with greater priority (scheduler module, supervisors and asynchronous modules) to take into account.

Algorithm 3 presents our CPU load verification:

Algorithm 3: precompute the CPU load

Data: $Config$ configuration to schedule
Data: S scheme to schedule

add S to configuration $Config$
$load =$ calculating the load for $Config$
if $load > threshold$ **then**
    notify supervisors that the scheme is not sequenced

In our precomputation algorithm, we can add the consideration of priorities between scheme. Thus when we compute the CPU load, we start the highest priority to lowest priority. Algorithm 4 presents the changes.

We can notice three things in our algorithm 4:

- If the addition of a higher priority scheme is theoretically not schedulable with the current configuration, then a scheme of lower priority will not be scheduled.

- The order scheduling for schemes of same priorities is determined by the algorithm already in place in ContrACT.
- If an overload is detected, the scheduler sends a notification to the adequate supervisors (those who asked to execute the scheme that are now no more scheduled).

Algorithm 4: precomputation of the CPU load with consideration of priorities schemes

Data: $Config$ configuration to schedule
Data: L the list of priority lists sorted in descending order of priority level
Data: S the scheme to schedule

$load = 0$
Add S at the list of priorities $l \in L$ which corresponds to it.
for $l_i \in L | i = 1..n$ *with* $1 > n$ *for priorities* **do**
  $U_{l_i} = \sum_{j=1}^{n} \frac{C_{l_{ij}}}{T_{l_{ij}}}$
  $load = load + U_{l_i}$
  if $load > threshold$ **then**
    notify supervisors that the scheme is not sequenced
    return

Since statistics on synchronous modules are available, they can be used to compute a good estimation of the CPU load, in order to have a value closer to reality. Of course the modules's delay adaptation can be more or less important depending on the execution context of the robot, that's why modules statistics have to be bound with adequate scheme that characterize most of their execution context. So, the goal is to be as close to realistic values as possible to avoid the CPU load verification process to be too constraining.

For now the CPU load check is always made immediately, whether it is based on nominal values (time constraints values) or adjusted values. Nevertheless, one improvement would be to let the scheduler adjusting all delays (if needed) before doing the check N cycles later, in order to take decision only on realistic values. But what is problematic with this option is the configuration of the number N of cycles that could lead, if not well adjusted, to a significant number of temporal faults.

4 Proposed schemes ameliorations

This section presents the new way schemes are reified in the ContrACT environment. The basic idea is to transform schemes into independently reusable blocks of code, as for modules. It is achieved by providing an interface to schemes' inputs and outputs (see figure 9). An interface is defined the same way as modules's ports.

N.B. The examples presented here are derived from two nested control laws (see Figure 2), the first "cartesian" and the second "articular". For notational convenience we will use letters to name the modules.

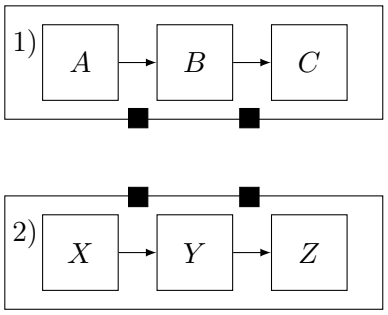

Figure 9: Adding interfaces on schemes

In a scheme, we have two types of connections: Connections between modules and connections between a scheme and a module. The first case is realized by the system implemented in ContrACT and detailed in [10].

Figure 10 shows the second case in which there are two examples of connections between modules and a scheme. An input interface can be connected to an input port and vice versa, as shown on the figure.

4.1 Internal switching

The reification of schemes will allow us to simplify scheme switching. Thus, when adding a scheme, one

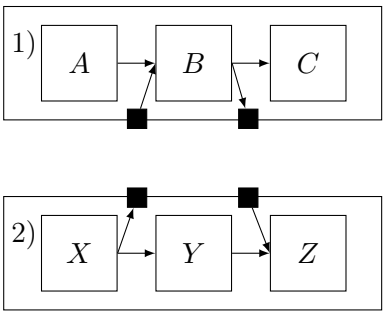

Figure 10: Connections between modules and schemes

must first create it. To do this the user specifies the constituent modules. Next, he connects the modules together (see Figure 9), then connects the modules that must communicate with the outside to the scheme in which it is located's interfaces (see Figure 10).

With the internal connections of scheme realized, we connect them with each other as shown in Figure 11.

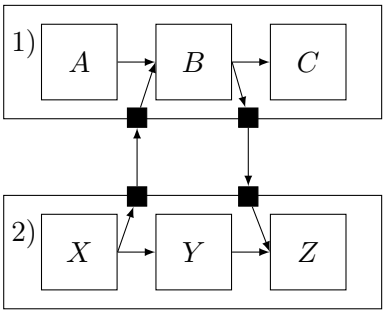

Figure 11: Connections between schemes

When switching schemes, we start by cutting the connections to the scheme's interfaces. Once the connections have been cut, we replace the scheme by an other one which has its internal connections satisfied. Finally, we reconnect the interfaces between these schemes. The reconnection of interfaces will depend on the dependencies that we will discribe below.

The benefit of this method compared to the current version of ContrACT is mainly the possibility to switch schemes. Because we no longer have to specify which modules communicate with which, we only have to specify the dependencies of the scheme. Dependency resolution (discussed in the next section) takes care of making the connection just before scheduling.

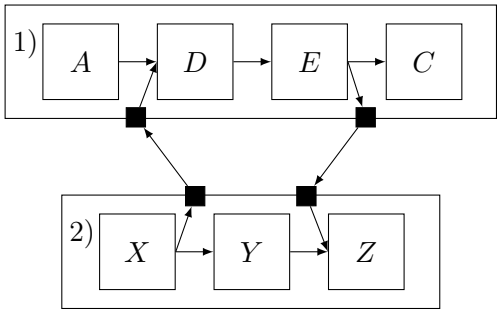

Figure 12: We replace the B module by two others D and E

To switch from Figure 11 to Figure 12 we will proceed in five steps:

1. Create the new scheme (connect the modules together and the scheme) ;
2. Delete connections between the scheme we want to replace and other schemes and unschedule it ;
3. Send a schedule request for the new scheme ;
4. Resolve dependencies between interfaces ;
5. Schedule the new scheme.

Step 1 is achieved through the software provided by ContrACT which will have to add the management of the reification of schemes. Step 2 is not necessary if the aim is to add the scheme instead of replace it. Step 3 is detailed in section 3.4. Step 4 will be detailed in the section 4.2. The last step is performed by ContrACT's scheduler in which we added the event management and the implementation of priorities between schemes.

4.2 Expression and Resolution of dependencies between schemes

In this section is presented the way interfaces of schemes are described and used in the resolution of dependencies.

Schemes' interfaces are described the same way as modules' ports : they have a direction (input or output), a data type (e.g. GPS data, articular coordinates, navigation data) and a name. The main difference is in the use of the name : while this name is not used when connecting ports (only data type equivalence is checked), the name of a scheme interface is an important identifier for dependencies resolving process. So, when two or more interfaces are connected, their type but also their name is checked for equivalence, and of course there is a verification of their respective direction (one interface as ouptut for N interfaces as inputs). Name checking is important to understand the role of the interface, the type of data it produces or consumes being not sufficient to resolve dependencies. For instance the name "articular_velocity_arm2" could represent the articular velocity of the arm number 2 of a two armed robot. When resolving dependencies, the system has to take care of names to make schemes produce and consume adequate data.

Basically, the dependencies resolution is a simple binding of interfaces identifiers in function of their name, type and direction. According to the direction of the interface, the dependency is provided by the scheme (outgoing) or if it is requested by the scheme (incoming). Jointly to names and types, the system knows at any moment which scheme required or provide a given data. As readers can understand, this resolution process requires a global table of exchanged data (identified by name and type) and associated schemes that produce and consume it. This phase can not be realized at the level of supervisors, because they just have a partial vision on schemes that are available in the system. Therefore the dependency resolution has so to be done at level of the scheduler module, since it is the only module that centralizes information on periodic schemes. The scheduler performs dependencies resolution during the precomputation phase, each time a scheme is launched, stopped, or switched.

The first step of the resolution so consists in, for all active scheme, checking if each requirement (input interface) is satisfied by exactly one providing (output interface). Resolutions errors come when more than one active scheme produce the same data and when at least one requirement is not satisfied. Algorithm 5 presents this check applied to each scheme (each dependency must be considered as named, typed and directed). Once this first step of resolution process is done, the resolver directly binds modules's ports together according to 1) the schemes' internal connections between internal modules' ports and its involved interfaces and 2) the connection between schemes interfaces.

Algorithm 5: Algorithm checking the dependencies of a scheme

Data: LO the list of dependencies as output

```
for s ∈ all active scheme do
    for d ∈ s.input_dependency do
        for lo ∈ LO do
            if d.already_connected then
                disconnect S.input_dependency
                notify supervisors we can't
                resolved dependency
                S.input_dependency
                return
            else if ¬d.already_connected and
            lo = d then
                connect lo to d
                d.already_connected = true
        if ¬d.already_connected then
            notify supervisors we haven't
            resolved dependency
            S.input_dependency
            return
```

If the dependencies resolution fails, for instance because a dependency is missing in the architecture (no scheme provides it at a moment of the mission), then the active scheme whose dependency is not satisfied is removed from the current schedule. And supervisors that are listening to such scheduler events are notified.

From this first simple resolution process, we observed (see Figure 13) that links not satisfied may be correct in some cases. Indeed, in our example, module B does not use data supplied by scheme 1 input interface (e.g. algorithm based on other data).

In this case, the fact that its port is not connected to the scheme input interface does not matter. Reversely, module Z shall need data provided by schema 2 input interface (e.g. command law requiring a set point). So, there is an error if module B's output port is not connected to the interface of scheme 1 or (as in figure 13) if module Z's port is not connected to the interface of scheme 2. Indeed, it should for example mean that a command law module will not receive set points, what is certainly a design error.

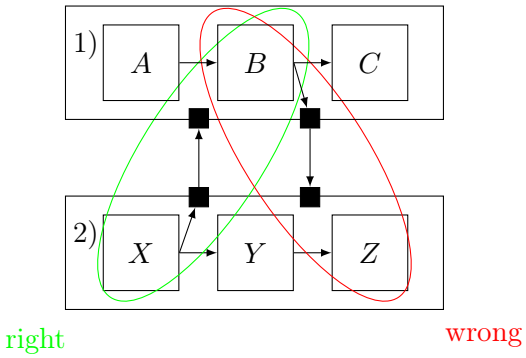

Figure 13: Links between schemes

This observation leads us to define the concepts of "mandatory and optional" ports and interfaces. This concept is not new in the frame of software engineering [9]: it consists in defining connection constraints for ports. A mandatory port or interface must be connected, while an optional may be let unsatisfied, the appreciation is left to the user. During the scheme's design phase, the connection of mandatory ports (between themselves and with scheme interfaces) is statically checked. On the contrary, mandatory interfaces connection is dynamically checked when resolution is checked.

But the notion of "mandatory and optional" does not contain all the possibilities that the user might want to express. For example, one might want to specify that either the interfaces 1 and 2 are mandatory or the interfaces 1 and 3 are mandatory in function of which dependencies are offered by the currently scheduled schemes.

To realize these concepts we introduce the possibility to constrain which interfaces and ports are mandatory through first-order logic. So if we want to express the example above we write:

$$(interface1 \wedge interface2) \otimes (interface1 \wedge interface3)$$

All possible combinations in the first-order logic can be expressed using interfaces as variables.

We think that these improvements on schemes' structure, allowing an automation of schemes reconfigurations, will greatly help the users to precisely control their system, notably control laws commutation. This is also an important characteristic for the management of fault tolerance mechanism defined in [4], in which possible scheme commutations are numerous.

5 Discussion and Conclusion

We are adding to ContrACT various mechanisms to observe, adapt and control real-time properties of ContrACT-based architecture in a simpler and more precise (finer grain) way than currently. To this end, regarding current limitation of ContrACT, we are enhancing the ContrACT applicative scheduler module with many new features: observing, memorizing and computing statistics on modules execution times; CPU load checking; temporal faults reaction and configuration of the generation of temporal exceptions; schemes priority control; schemes commutation; automatic reconfiguration of inter-scheme relationships. Some of these features are already implemented (duration observation, automatic adaptation) while others remain to be completely specified or implemented.

Concerning the reification of schemes, once implemented, it should simplify the development of interactions between different control laws. Indeed, the user will not have to realize the connections between modules of different scheme, but only to define the connections between the modules of the same scheme. Then the scheduler will take care of resolving dependencies between schemes. The automatic management of scheme dependencies and internal switching, coupled with scheme priorities

management is in our opinion, a main improvement to precisely control reaction/adaptation when reconfiguration is required (for instance after a temporal exception notification): the adaptation (e.g. internal switching of a scheme to commute control laws) defined in a supervisor can precisely target a given periodic loop (scheme) while keeping consistent relations between active loops (with no more efforts as compared to current version) and without perturbing more critical parts of the system.

With this new approach, it is possible to put in place a system in which adaptation decision can be decomposed into a hierarchy of independently reusable supervisors, each one managing a specific responsibility in control (e.g. articular control, cartesian control, teleoperation, visual servoing, etc.). In terms of dependability, we hope this improvement will be useful in fault tolerant architectures [4], notably to program supervisors that manage fault recovery processes.

References

[1] D. Brugali and M. Reggiani. A Roadmap to crafting modular and interoperable robotic software systems. In *SDIR05, Principles and Practice of Software Development in Robotics, ICRA workshop*, Barcelona, Spain, Apr. 2005.

[2] A. Cervin. Analysis of overrun strategies in periodic control tasks. In *16th IFAC World Congress*, Prague, Czech Republic, 2005.

[3] B. Durand. *Proposition d'une architecture de contrôle adaptative pour la tolérance aux fautes.* These, Université Montpellier II - Sciences et Techniques du Languedoc, June 2011.

[4] B. Durand, K. Godary, L. Lapierre, R. Passama, and D. Crestani. Using adaptive control architecture to enhance mobile robot reliability. In *TAROS'10: 11th Conference Towards Autonomous Robotic Systems*, pages 54–61, Plymouth, Royaume-Uni, Sept. 2010.

[5] A. El Jalaoui. *Gestion Contextuelle de Tâches pour le contrôle d'un véhicule sous-marin autonome.* These, Université Montpellier II - Sciences et Techniques du Languedoc, Dec. 2007.

[6] A. El Jalaoui, D. Andreu, and B. Jouvencel. Auv control architecture for control management of embedded instrumentation. In *4th IFAC Symposium on Mechatronic Systems (Mechatronics'06)*, Heidelberg, Germany, Sept. 2006.

[7] A. El Jalaoui, D. Andreu, and B. Jouvencel. Contextual management of tasks and instrumentation within an auv control software architecture. In *IEEE/RSJ IROS*, Beijing, China, Oct. 2006.

[8] H. Kooti, D. Mishra, and E. Bozorgzadeh. Reconfiguration-aware real-time scheduling under qos constraint. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ASPDAC '11, pages 141–146, Piscataway, NJ, USA, 2011. IEEE Press.

[9] A. Leicher, S. Busse, and J. Süß. Analysis of compositional conflicts in component-based systems. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Software Composition*, volume 3628 of *Lecture Notes in Computer Science*, pages 67–82. Springer Berlin / Heidelberg, 2005.

[10] R. Passama and D. Andreu. Contract: a software environment for developing control architecture. In *6th National Conference on Control Architectures of Robots*, page 16, Grenoble, France, May 2011. INRIA Grenoble Rhône-Alpes.