CEL-based Integration Policy for Critical Multi-layered Applications

Pierre Pomiers

Abstract—
Tomorrow, advanced mobile robotic applications have to be able to cope with various situations and perform tasks in a dynamic environment. Furthermore, finding concrete applications in a wide range of user oriented industrial products, such systems, embedding several computing units, have to support both increasing demand of interactivity and number of non-critical pieces of hardware and software. To this aim not only advanced programming techniques, but also appropriate control architectures are required. This paper proposes a generic methodology, actually in use for our own products, that offers both the possibility to model such internally competing systems and the capability to coordinate them. A special parallel is drawn between our approach and the SynDEx[1] software methodology, developed by INRIA, on which our current implementation relies.

Keywords— **CEL, multi-layered applications, synchronous implementations, distributed executives, critical software**

I. Introduction

CLASSICALLY , and according to [1], we consider that real-time embedded systems react, within bounded delays, to input stimuli received from the environment by generating output reactions and changing their internal state. Considering such systems, both hardware and software reliability appear to be critical. Various techniques and tools exist, proposing solutions for implementing mono-component sequential software.

Nevertheless, when algorithms complexity increases, as well as the number of integrated sensors and actuators, classical sequential architectures become inadequate. First, because a single computing machine can only handle a limited number of external input/output ports and secondly, because the ratio between computation volume and the response time bound becomes too high. Thus, the convenient choice of parallel architectures is required to satisfy real-time constraints, distributing computation load, and to take into account the distributed nature of the system resources (sensors, actuators, computing units, memory).

Programming such architectures is an order of magnitude harder than with mono-component sequential ones, and even more when hardware resources must be minimized to match cost, power and volume constraints required for embedded applications [2]. To override the complexity of computing such application algorithms, the control software is built using a dedicated software developed by INRIA: SynDEx. Relying on the AAA[2] methodology [1] and language DC [3], SynDEx objective is to prototype and optimize implementations of parallel, distributed and heterogeneous application through three main steps: specification, validation and code generation.

P. Pomiers is with Robosoft S.A., Technopole Izarbel, 64200 Bidart, France. E-mail: pierre@robosoft.fr.

[1]Synchronized Distributed Executive

[2]Algorithm Architecture Adequation

II. Synchronous data flow graph formalism

Handling parallel distributed architecture raises the problem of scheduling application execution between the different components of the architecture. In non optimized solutions, client/server mechanisms must be integrating manually inside each component executives as it is shown on Figure 1.

Most of the time, centralized implementation is considered because it makes system synchronization and consistency be easily maintainable. Nevertheless, this relative facility for integration shadows non negligible constraints: system states are stored in one place, component or software clients are necessarily directly connected to the server, and, messages routing between client is realized by the central server. Hence, centralized systems are penalized by the cost of their communication loads. This remark is all the more evident when the number of both hardware and software components increases. Scalability problems can not find any satisfactory solution with this approach.

Supporting and scheduling fully distributed architecture is more of the synchronous approach resort. Advantages of such approach has already been proved as it allows verification of functional and temporal requirements, in the early stages of the development cycle. These languages rely on a special specification in which algorithms description is a set of equations that must be always verified by the program variables. This approach is inspired by formalisms familiar to control engineers like block diagrams (refer to [4] for further details).

Benefits of using a synchronous data-flow language for programming critical real-time systems appear clearly: the data-flow approach meets both the traditional description tools used in control domain and the ability to support formal design and verification methods [5]. Figure 2 shows the way reactive system algorithms are intended to be described with this formalism.

III. Implementing optimized distributed applications

A. AAA methodology overview

AAA methodology aims to find the best matching between an algorithm and an architecture, while satisfying constraints. For, it is based on graphs models to exhibit both the potential parallelism of the algorithm and the available parallelism of multicomponent architecture. Its implementation consists in distributing and computing a

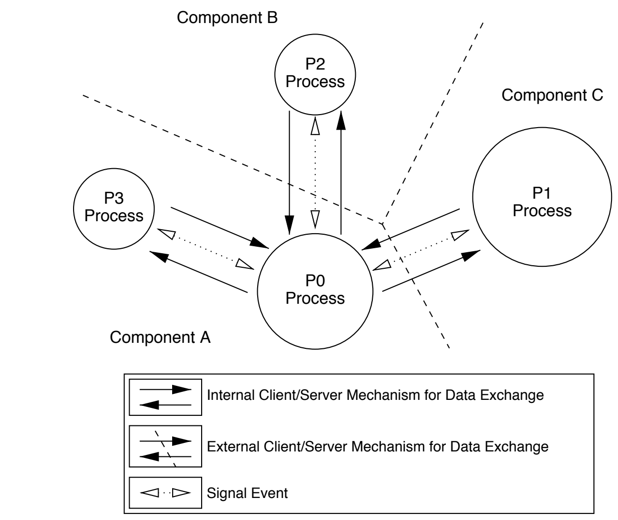
Component B
P2 Process
Component C
P1 Process
P3 Process
P0 Process
Component A
Internal Client/Server Mechanism for Data Exchange
External Client/Server Mechanism for Data Exchange
Signal Event

Fig. 1. Example of a centralized implementation

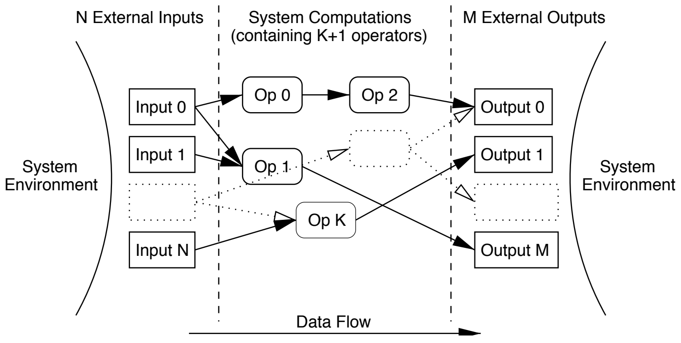

Fig. 2. Example of a reactive system algorithm described as a data-flow graph

static schedule of the algorithm data-flow graph on the system architecture graph while satisfying real-time constraints and optimizing resources allocation (refer to [6]).

In the case of concrete real-time embedded systems, by resources we mean the computing power of all the architecture operators (i.e. either processors or micro-controllers), as well as the data communication bandwidth available on the communication media interconnecting this set of computing units.

B. Generation of synchronous code

The result of both algorithm graph and architecture graph transformations is a set of optimized synchronized distributed macro executives. Generated macro executives are coded in a specific language, compatible with other synchronous languages compilers and tools, through the common format DC [3]. Afterward, binary executives, aiming to be run on the components of the described architecture, are automatically built from an hardware dependent executive kernel. It exists one executive kernel for each supported processor supporting various functionalities such as: memory allocation, data communications, synchronization and related hardware input/output supports.

Let us point the fact that each operation being a part of an executive kernel is said to be safe, what means that its coding has been done regarding to the hardware limits. The same way, the generalization of this assertion prove that any composition of such safe operations is safe. A direct consequence of this is that system architectures running such generated executives do evolve in a safe state.

C. Conclusion about SynDEx approach

Thanks to the steps of specification, verification inherent to co-design using SynDEx and, finally, thanks to automatic code generation it offers, most of the usual coding mistakes are avoided. This point is very encouraging, on one hand because it saves a lot of development time (generally spent debugging applications) and, on the other hand, because it assures the prototyped applications functioning to be safe. Last, SynDEx provides very small footprint executives, embedding only the minimum of code required for designed applications.

Nevertheless, this approach is not beyond reproach. This technique is extremely efficient while the complete application development process respects the same description formalism. This point has to be discussed. As mentioned above, data-flow graph techniques, provided by SynDEx to handle massively parallel architectures, focuses on static schedule of real-time systems and fixed-duration executions. If this is well adapted to distributed applications implemented with a single consistent execution layer, it is definitely not the case for more complex multi-layered application.

IV. Embedded multi-layered applications design

Using SynDEx, for monolithic massively parallel applications, appears to be really helpful, as it automatizes distributing and scheduling processes. Nevertheless, this approach does not fit multi-layered programming. Here, we propose an additional methodology, intended to be used implementing such multi-layered applications under the SynDEx synchronous approach. Afterwards, this integration approach will be generalized to multi-layered applications resulting of mixed modeling and programming methods, either synchronous or not.

A. Preliminary definition

For a best comprehension of what follows, it is now important to well define what actually is a multi-layered application. Each complex application, either in field of embedded systems or not, possesses a variety of features or execution modes [7]. Each operation or composition of operations, used for implementing these application parts, have to be organized in coherent levels of criticality. Hence, each of such defined levels are adjusted to their specific functional layer needs. This is the basic concept of applications consistent execution layers.

It is important to remark that we are not talking about abstraction layers. Abstraction layers are generally used for describing the different levels of use of an application

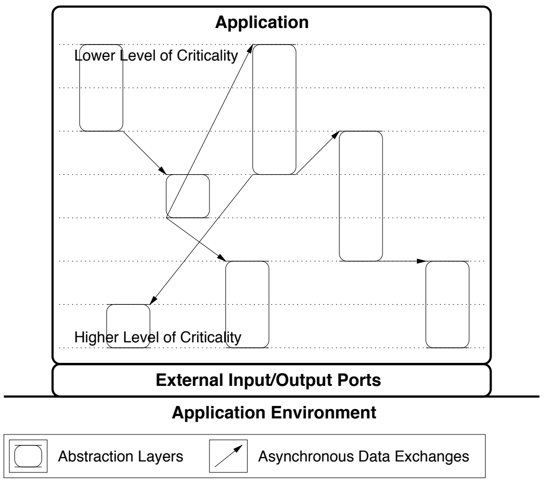
Application
Lower Level of Criticality
Higher Level of Criticality
External Input/Output Ports
Application Environment
Abstraction Layers
Asynchronous Data Exchanges

Fig. 3. Abstraction layers splitting strategy

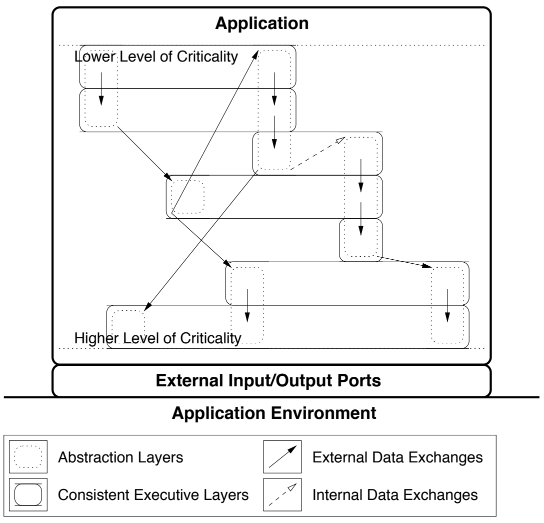
Application
Lower Level of Criticality
Higher Level of Criticality
External Input/Output Ports
Application Environment
Abstraction Layers
Consistent Executive Layers
External Data Exchanges
Internal Data Exchanges

Fig. 4. CEL splitting strategy

in terms of software engineering. CEL[3] based approach is really different. In our case, splitting a given application does not result from any mind representation of the application, but results from runtime constraints considerations. This lead to a marginal meaning: indeed, as shown in Figure 3, classical abstraction layers can cover several levels of criticality, while, in Figure 3, CELs are depicted including several pieces of abstraction layers into the same level of criticality. As a last remark, do keep in mind that a CEL execution is not constrained to only one architecture component, but may be distributed over all the hardware architecture. Consequently, a CEL has to be understood as a potentially massively parallel and heterogeneous part of the application.

B. Consistent execution layers internals

As far as abstraction layers splitting is concerned, each pieces of application executes over multiple criticality levels which are subject to different constraints (either real-time or not). For instance, it is the case with abstraction layers including several different real-time periods or including reactions to multiple events under interrupts. Consequently, when implementing abstraction layers using synchronous languages, it results, at compilation time, sequential or pseudo-sequential executives, fully synchronized either on the lowest internal period or less frequent event. Such a software inevitably runs with no regard to application criticality constraints.

In order to respect application constraints, SynDEx should be used only for implementing pieces of software of the same criticality level. Including the remarks given above, benefits of splitting applications into CELs is now obvious. Indeed, a given application can be represented as a set of consistent software layers, what allows each layer (CEL) to be implemented using SynDEx, independently of from the others layers.

Let us remark that, CELs splitting can be easily automatized. Let A be a software application composed of several abstraction layers named $\mathrm{F_i}$ where $\mathrm{i} = \{0, ..., \mathrm{N}-1\}$. Assuming application A covers P criticality levels $\mathrm{C_j}$ where $\mathrm{j} = \{0, ..., \mathrm{P}-1\}$. Let us give the mathematical expression of application A compsition in case of abstraction layers splitting.

$$\mathrm{A} = \bigcup_{\mathrm{i}=0}^{\mathrm{N}-1} \mathrm{F_i} \tag{1}$$

Each abstraction layers $\mathrm{F_i}$ covers at leat one criticality level. Hence we obtain the following equation.

$$\mathrm{F_i} = \bigcup_{\mathrm{j}=0}^{\mathrm{P}-1} (\mathrm{F_i} \cap \mathrm{C_j}) \tag{2}$$

Now, we can give a new expression of application A composition:

$$\mathrm{A} = \bigcup_{\mathrm{i}=0}^{\mathrm{N}-1} \left(\bigcup_{\mathrm{j}=0}^{\mathrm{P}-1} (\mathrm{F_i} \cap \mathrm{C_j}) \right) \tag{3}$$

which is equivalent to:

$$\mathrm{A} = \bigcup_{\mathrm{j}=0}^{\mathrm{P}-1} \left(\bigcup_{\mathrm{i}=0}^{\mathrm{N}-1} (\mathrm{F_i} \cap \mathrm{C_j}) \right) \tag{4}$$

Equation 4 describes application A composition as the union of each software component (of the same criticality level) sub-set. Let us call these software sub-sets $\mathrm{E_k}$, with $\mathrm{k} = \{0, ..., \mathrm{P}-1\}$, such as:

$$\mathrm{E_k} = \bigcup_{\mathrm{i}=0}^{\mathrm{N}-1} (\mathrm{F_i} \cap \mathrm{C_k}) \tag{5}$$

application A composition expression becomes:

$$\mathrm{A} = \bigcup_{\mathrm{k}=0}^{\mathrm{P}-1} \mathrm{E_k} \tag{6}$$

[3]CEL is the denomination we use for consistent execution layer

Equation 5 bring the proof that all applications composed of abstraction layers get an equivalent application representation, resulting from a criticality level splitting. Hence, each inconsistent [4] appllication may be transformed into a set of independent consistent software layers (CELs). Each software layers of such equivalent CEL-based applications now fits synchronous implementation requirements.

Hence, CELs splitting method prevents from blocking interactions between inconsistent parts of software. As a result, the global implementation of an application becomes consistent as it fully respects the various application internal constraints. Figure 5 shows a representation of a CEL and its interaction infrastructure. Let us note that no external synchronization is possible, synchronization mechanisms are only allowed for internal CEL algorithm. Thus, a CEL can only exchange data this its environment asynchronously, that is to say without the possibility to block the distant CELs execution. This functioning well respects the local execution consistency, as well as, by extension, the global application execution consistency. About CEL communications, there is no limitation concerning supported media types: for instance industrial buses, serial lines, network connection, as well as, shared memories may be used.

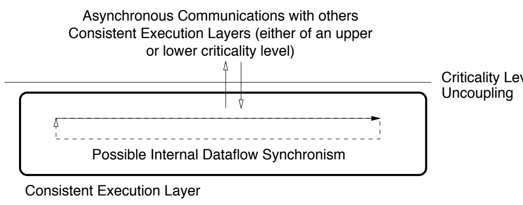
Asynchronous Communications with others Consistent Execution Layers (either of an upper or lower criticality level)
Criticality Level Uncoupling
Possible Internal Dataflow Synchronism
Consistent Execution Layer

Fig. 5. Generic model of a CEL

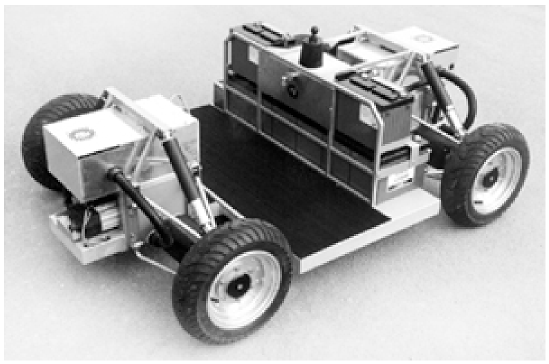

Fig. 6. The RobuCar robotics platform

C. Interfacing multiple CELs

Establishing interaction between CELs is something easy. As CELs executions are not conditioned by any CELs executions, asynchronous communication interface is nothing but inspired by DMA[5] mechanism. In order to prevent CEL from handling fine data transfers between its own internal memory and others CELs memories, a DMA-like shared memory unit is added to CEL communication layer.

CEL communication interface units include their own transfer scheduler. The so called data transfer scheduler is partially independent from CEL instructions scheduler. Indeed, in order to be programmed and initiated, data transfer scheduler requires calls to instructions scheduler, but does not interact with it anymore during the transfer. This allows a real parallelism between computations and communication sequences.

Transfers are realized by an automata, able to access directly local CEL memory content and to transfer contiguous data from (resp. to) external input (resp. output) communication media devices. In order to realize such data transfer, CEL is normally able to access all communication hardware available on the operator it is handled by. Nevertheless, we recommend intentionally to use only shared memory devices. Let us justify this choice.

Unlike buses or point-to-point communication media, shared memory is the only mechanism that allows data transfer whitout requiring hardware interrupts handling for reading or writing operations. This remark is very important. As it is explained above, data exchanges between CELs are realized asynchronously, with respect to each CELs critical functioning. Basically, if some data were transfered asynchronously under interrupts, each incoming data would stop execution of the receiving CEL until transfer stops. This would, inevitably, lead to a non-predictable schedule. On the opposite, considering shared memory devices, CELs integrity is fully preserved. Indeed, each data transfer falls into two part: reading operations, handled by the receiving CEL, and writing operations, handled by the emitting CEL. Hence, even mutually asynchronous, such reading and writing operations are scheduled on their respective CELs, avoiding any unpredictable behavior. Now, remains to find a strategy for routing data between CELs. To this aim, let us detail CEL communication layer nature.

As mentioned above, let $\mathrm{E_k}$, where $\mathrm{k} = \{0, ..., \mathrm{P}-1\}$, be software sub-sets of homogeneous criticality levels. Each sub-sets $\mathrm{E_k}$, representing the CEL of criticality level k, is could be seen as a set of executives $\mathrm{e_{k_i}}$ $(\mathrm{i} = \{0, ..., \mathrm{n}\})$ attached to a set of computing units $\mathrm{O_j}$ $(\mathrm{j} = \{0, ..., \mathrm{N}\})$, as expressed below:

$$\begin{array}{rrl} E_k = & & \{e_{k_0|O_0}, ..., e_{k_i|O_0}\} \\ & \cup ... \cup & \{e_{k_j|O_J}, ..., e_{k_l|O_J}\} \\ & \cup ... \cup & \{e_{k_m|O_N}, ..., e_{k_n|O_N}\} \end{array} \quad (7)$$

where $\mathrm{e_{k_i|O_j}} = \emptyset$ if it exists no executive $\mathrm{E_k}$ running on computing unit $\mathrm{O_j}$. Let Ω_k be the computing units set concerned by CEL $\mathrm{E_k}$ execution:

$$\Omega_\mathrm{k} = \{\mathrm{O_h}, \mathrm{O_i}, ..., \mathrm{O_j}\} \quad (8)$$

It becomes possible to determine a computing units subset supporting, simultaneously, two CELs execution (for instance CEL $\mathrm{E_p}$ and CEL $\mathrm{E_q}$):

$$\Omega_{\mathrm{p|q}} = \Omega_\mathrm{p} \cap \Omega_\mathrm{q} \quad (9)$$

[4] by inconsistent we mean of different criticality levels
[5] Direct Access Memory

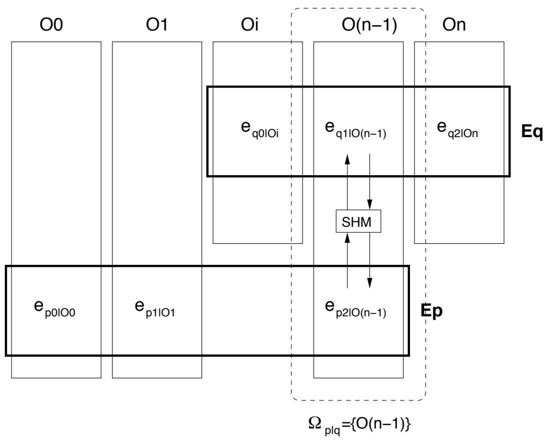

Fig. 7. Data exchange example between two CELs $\mathrm{E_p}$ et $\mathrm{E_q}$, with $\Omega_{\mathrm{p|q}} \neq \emptyset$

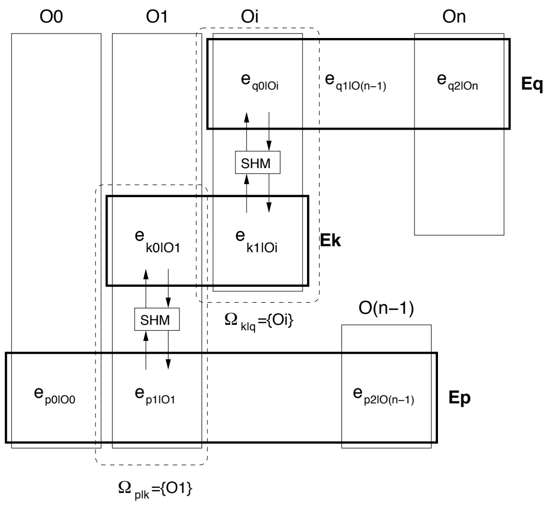

Fig. 8. Data exchange example between two CELs $\mathrm{E_p}$ et $\mathrm{E_q}$, with $\Omega_{\mathrm{p|q}} = \emptyset$ and $\Omega_{p\wr q} \neq \emptyset$

This last equation is helpful to determine how data should be routed between two CELs. Assuming $\Omega_{\mathrm{p|q}}$ is not empty, if CEL $\mathrm{E_p}$ and CEL $\mathrm{E_q}$ are consecutive, data trasfert is possible directly, from one CEL to the other, via one of the computing unit listed by $\Omega_{\mathrm{p|q}}$ (please, refer to figure 7). Else, if CEL $\mathrm{E_p}$ and CEL $\mathrm{E_q}$ are not consecutive, $\Omega_{\mathrm{p|q}} \neq \emptyset$ indicates it exists at least one computing unit capable to forward data from CEL $\mathrm{E_p}$ to $\mathrm{E_q}$. On the other hand, if $\Omega_{\mathrm{p|q}}$ est empty, we need to find one or several intermediate CELs $\mathrm{E_k}$ such as:

$$\Omega_{\mathrm{p\wr q}} = \Omega_{\mathrm{p|k_i}} \cup \Omega_{\mathrm{k_i|k_j}} \cup ... \cup \Omega_{\mathrm{k_h|q}} \neq \emptyset \qquad (10)$$

that is to say, such as each $\Omega_{\mathrm{p|k_i}}$, ..., $\Omega_{\mathrm{k_i|k_j}}$, ..., $\Omega_{\mathrm{k_h|q}}$ sub-sets are not empty. Then, $\Omega_{\mathrm{p\wr q}}$ figures the computing unit set need to route indirectly data from CEL $\mathrm{E_p}$ to CEL $\mathrm{E_q}$ (please, refer to figure 8). If no CEL, neither group of CELs, allows to find a solution to this routing problem (i.e $\Omega_{\mathrm{p\wr q}} = \emptyset$), CEL $\mathrm{E_p}$ and CEL $\mathrm{E_q}$ are considered to be isolated.

V. Evaluation scenarios

Each point of the method discussed here has been comforted with a number of applications ranging from critical sensoring and control to user oriented software. One of the most representative implemented system is a Robosoft own industrial product named RobuCar (refer to Figure 6). RobuCar is a robotized platform, specifically designed for urban applications and automated transport , composed of four totally independent wheels (i.e. one DC motor and one steering servo per wheel). Common car driving security matters, together with mechanical related ones, led to the actual RobuCar hardware architecture. The vehicle embeds one Motorola MPC555-based control board per wheel and an Intel Pentium-based industrial computer. All architecture components interconnected through two CAN buses and two serial lines. An Ethernet wireless link can be added to allow communications with network oriented application parts running on off-board components. Basically, the RobuCar set of applications has to cope with mixed features such as: path planning, wire guidance, image processing, teleoperation, down to critical actuators control and security procedures.

As an example of typical multi-layers interfacing requirements, we propose to describe some aspects of one RobuCar application, implementing teleoperation and distant monitoring. Figure 9 show the interconnected CELs network corresponding to the application described below. Let us notice that CELs are numbered relatively to their own criticality level: from CEL 0, the most critical one, to CEL 7, the less critical one. CELs 0, 1, 3 and 5, distributed over heterogeneous components, are designed with SynDEx, while remaining CELs are freely developed using miscellaneous programming methods. The following functionalities are studied (each mentioned sub-set may result from an abstraction layers partitioning):
Motion control: first, this process is in charge of the user command acquisition and command filtering. Then, such computed commands are diffused over the distributed control boards. Finally, low-level motor and steering servo controls are performed one each independent control board. This process corresponds to operations sub-set {CA, CF, CD, MC, SC} (refer to Figure 9 for abbreviations meanings).
Security control: this process consist of a watchdog, electrically linked to convenient vehicle signals, and emergency stop handlers (one per wheel) able to drive brakes and inhibit incoming motion commands (sub-set {WD, ES}).
User operating level: this level handles user actions and reports them to the control levels (sub-set {UI, HL}).
System monitoring: this part of the application aims to collect internal status of each control boards and to diffuse data toward distant components. Thus, status information may be accessed by two applications: one in charge of monitoring and updating a log data base, and an other one, implementing the graphical user interface (sub-set {ST, SU, SD, SM, GI}).

Results we obtain obviously reach our expectations and show that CEL system integration strategy well fits very

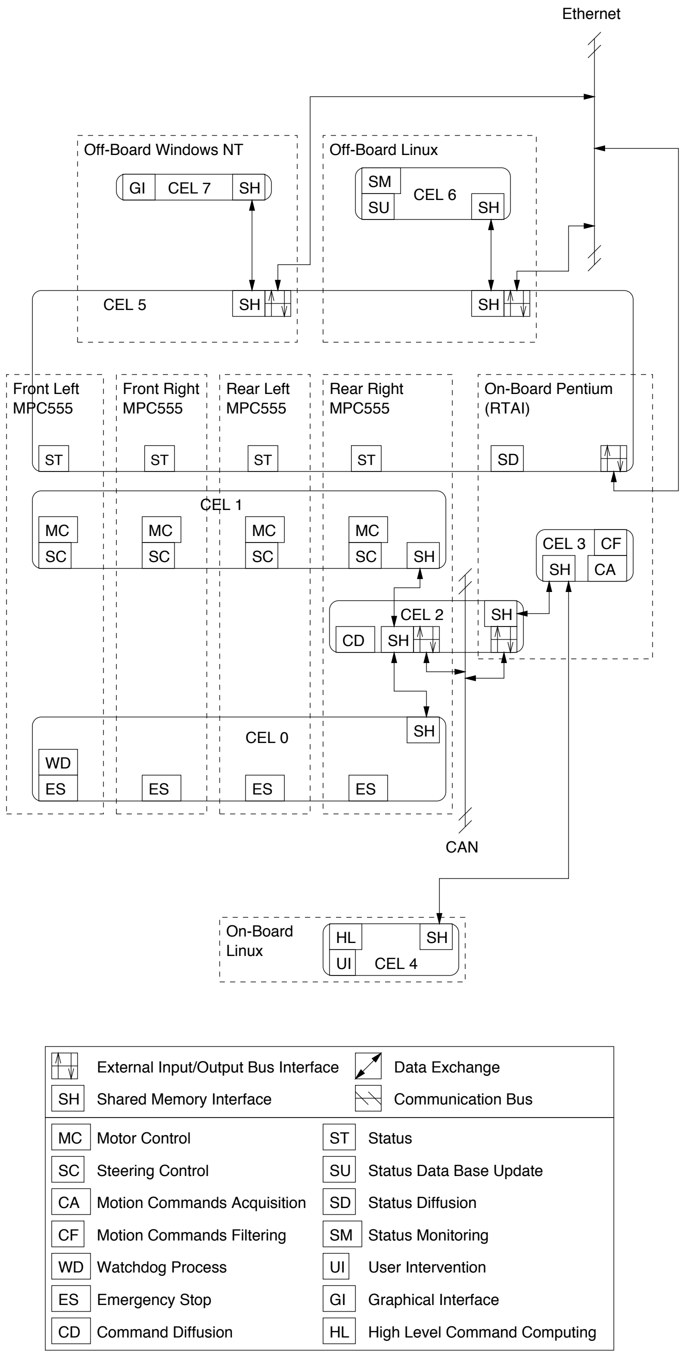

Fig. 9. CEL interpretation of a RobuCar application

constrained embedded robotics applications needs.

First, because, considering the multiple internal process behaviors, such well described software architecture appears to be particularly flexible.

Secondly, because the resulting interpretation, also solves the raised problems of supporting implementation methods heterogeneity much more than the single hardware heterogeneity. Indeed, here, stand-alone generated critical synchronous executives and various less critical operations (either supported by user oriented operating systems or not) accomplish the challenging task of mutually sharing asynchronously the same application environment variables, safely and without constraining each others.

Finally, relying on SynDEx for implementation operations, generated executives appear to have an extremely small footprint. For instance, considering RobuCar case, binaries for MPC555 boards (resp. RTAI PC) are smaller than 4 kilobytes (resp. 9 kilobytes). None of the commercially available solutions, related to distributed embedded systems, provide user with such performances. Indeed, these approaches focus more on classical kernels and operating system structures, leading to larger binaries: approximately 20 kilobytes for micro-controllers, up to 1 Megabytes for real-time applications on PC computers.

Conclusion

The approach we discussed in this paper, brings improvements to what is proposed in [8] for classical distributed centralized robotics oriented architectures. Benefits are twofold. First, application distribution and scheduling steps supporting heterogeneous and massively parallel architectures becomes now possible and fully automatized. Secondly, this approach may lead to a future extension of the AAA methodology. Indeed, if criticality level criterions are introduced for elementary operations and operators, an appropriate heuristic would enable to automatically partition applications into CELs. To this concern, in our approach, a new sub-set of criterions, including: qualification of operators activity predictability, operators interrupts handling latency, operations execution priority levels and communication channels reliability. This constitutes a major enhancement. For, as software implementations quality is usually subject to user viewpoints and backgrounds, by such an extension we prevent users from taking not efficient decisions concerning application split, and enable to focus more on global application description.

References

[1] T. Grandpierre, C. Lavarenne, and Y. Sorel, "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors," in *7th International Workshop on Hardware/Software Co-Design CODES'99 - Rome, Italy*, May 1999, pp. 74–78.
[2] B. Shirazi, A. Hurson, and K. Kavi, "Scheduling and load balancing in parallel and distributed systems, IEEE computer science press," 1995.
[3] N. Halbwachs, *The declarative code DC, version 1.2a*, Vérimag, Grenoble, France, October 1995.
[4] N. Halbwachs, *Synchronous programming of reactive systems*, Kluwer Academic Pub., 1993.
[5] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and verifying real-time systems by means of the synchronous data-flow programming language lustre," *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.
[6] Y. Sorel, "Massively Parallel Computing Systems with Real Time Constraints - The Algorithm Architecture Adequation Methodology," 1994, pp. 44–54.
[7] Christian Schlegel and Robert Wörz, "Interfacing Different Layers of a Multilayer Architecture for Sensorimotor Systems using the Object-Oriented Framework SMARTSOFT," in *Proceedings of the 3rd European Workshop on Advanced Mobile Robots EUROBOT'99 - Zürich, Switzerland*, September 1999, pp. 195–202.
[8] Klas Nilsson and Rolf Johansson, "Integrated Architecture for Industrial Robot Programming and Control," *Robotics ans Autonomous Systems*, vol. 29, no. 4, pp. 205–226, December 1999.