AN ADL CENTRIC APPROACH FOR THE FORMAL DESIGN OF REAL-TIME SYSTEMS

Sébastien Faucou, Anne-Marie Déplanche, Yvon Trinquet
Institut de Recherche en Communications et Cybernétique de Nantes - UMR 6597
CNRS, École Centrale de Nantes, École des Mines de Nantes, Universit de Nantes
France
firstname.name@irccyn.ec-nantes.fr

Abstract This paper presents the REACT project, dedicated to real-time system design. REACT aims at combining into an architectural design process some formal modelling and verification techniques and providing those corresponding tools. It emphasizes on the ADL of REACT (CLARA), and the validation of functional architectures using formal techniques.

Keywords: ADL, real-time systems, architecture design process, formal validation

1. Introduction

The increasing complexity of real-time systems (as regards not only their functionality but also their hardware and software components and the interactions and mappings between these components) in domains such as in-vehicle embedded electronic, robotics, field-devices control, or avionic leads to attach more and more importance to their architectural design step. *Architectural design* is not only about specifying (or even choosing) and assembling (logical, software, hardware, ...) components together so as to create a coherent and functionally correct system. It has also to make sure that some other extra-functional[1] requirements such as timeliness, reliability, safety, costs, etc. will eventually be met by the system under operation. Introducing V&V activities as soon as possible in the development process of computer-based control systems is now a widely accepted need. This implies several important consider-

[1] As in (ARTIST, 2003), we use extra-functional instead of non-functional. Indeed, the timing constraints for instance, are part of the definition of the functionalities of a real-time system and thus should not be qualified as "non-functional".

ations: (i) designers must have available a way (a language) to describe (to model) the system architecture (its constitutive parts and relations) at different levels of abstraction; (ii) verification tools for system analysis and stating on its performances from the architectural level (and still after) have to be provided; (iii) the architectural design process (transformation(s) from the functional architecture up to the runtime configuration) has to be integrated to form a continuous and traceable chain, even an automatic one. It should guarantee that the (functional and extra-functional) properties stated at the upper levels are always met at the lower levels.

The objectives of our present work are to combine into an architectural design process some formal modelling and verification techniques. The strong dependency between embedded software and its execution platform requires us to focus on techniques that take into account the operational characteristics of the system, so as to reason on its extra-functional properties. REACT, "REal-time Application Configuration Tool", is the name we gave to this project and to the corresponding toolkit we are thinking of building.

Real-time systems that are aimed, are embedded ones, using some RTOS or middleware as runtime platform. They may be distributed. However we only consider situations where the system hardware is already defined (either a ready-to-use commercial platform or a reused yet designed one): hardware architecture design or co-design are out of concern.

In this paper, we present only a few constitutive parts of REACT: its architecture description language CLARA[2]and the validation of functional architectures. It is built as follows: in section 2, we give an overview of the development process, from the high level design to the binary code synthesis. In section 3, we introduce CLARA, the ADL that we use to perform the description of embedded software systems. In section 4, we discuss the validation of architecture descriptions with regards to some functional and extra-functional properties.

2. Design process

The REACT project aims at offering a set of formal modelling and verification facilities in an unified framework for the rigorous architectural design of real-time systems. It does not cover the whole development process: we consider that the specifications are given. Our main goal is to produce an operational architecture that has been validated

[2] CLARA: Configuration LAnguage for Real-time Application (Durand, 1998).

against some functional and extra-functional requirements (especially timeliness). Before to present the process, we define some vocabulary:

We call *functional architecture* (FA) the structure of the system in terms of its *functions*, their behaviours, and the control and data flows between them (*function* denotes a logical computation block that has not to be refined at this design level).

We call *runtime platform* (RP) the hardware and low-level software layer (RTOS, device drivers, middlewares, communication protocols) that are used for the deployment of the application. All these elements are seen as a platform, accessible through a set of runtime services.

We call *operational architecture* (OA) the result of the mapping of the FA onto the RP. The functions are allocated to tasks and the tasks are allocated to processing nodes. The control flows and data flows are translated into appropriate invocations of RP services. At last, the configuration of the RP (e.g. priorities of the tasks and messages) is given.

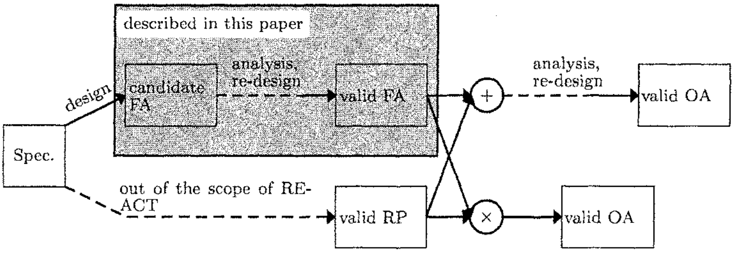


Figure 1. Overview of the design process of REACT.

As an entry point of REACT, we have defined an ADL, CLARA, dedicated to the description of the FA of reactive systems. It provides also some support for specifying timing constraints and properties. Thus, the design process (see fig.1) starts with the description of a candidate FA, which needs to be validated. For this purpose, we use Petri net (PN) theory to assess safety properties (eg. deadlock freedom). We also propose a high level consistency analysis of the timing constraints and properties (assessment of a necessary condition over the existence of a valid implementation). Given the results of these analyses, the designer can either validate the candidate and go on, or design a new one (generally not from scratch). As an output of these steps, a valid FA is defined, together with a set of timing constraints and properties that may be consistent. A complete illustration of this part of the process is given in the paper.

As stated in the introduction, we consider that the RP is already defined. To achieve the mapping of the FA onto the RP so as to produce an OA, we have explored a first direction and are presently working on a second one. We present both approaches hereafter (due to space limitation, they won't be discussed any further in this paper).

With the first approach (operator + on fig. 1, see (Faucou, 2002)), the mapping is made so as to "preserve the structure" of the FA. It targets especially the OSEK/VDX-based RP[3]. The active components are mapped onto OSEK tasks that interact, according to the connections described in the FA, through the services of a "CLARA middleware" (the configuration of which is extracted from the FA). The assignment of the tasks to the computing nodes and the configuration of the RP have to be done by the system architect. Such a mapping allows to preserve traceability between the levels (the FA and the OA). Nevertheless, it can potentially produce OA with a complex execution structure involving a lot of inter-task communications, the behaviour of which being hardly analysable. On the one hand, we have developed a simulation approach that takes into account the effective operational behaviour of the RP (including the middleware, OS and communication protocol). It is thus possible to observe for instance the impact of ISR executions on the scheduling of the tasks. It is also a good framework to play the "what-if?" game in order to tune the OA. On the other hand, we are studying the use of an extension of Time Petri Nets (TPN) to real-time scheduling (SETPN from (Roux and Déplanche, 2002)). These two approaches are supplementary.

The second approach (operator × on fig. 1) aims at defining algorithms and tools to generate a "valid by construction" OA ("valid" in reference to timeliness). Similar works are being presently driven in the "model integrated computing" community (Kodase et al., 2003). We target RP using fixed priority task scheduling and CAN protocol (ISO, 2003). To achieve our goal, we have identified several intermediate steps. In a first time, end-to-end control flows (transactions) are extracted from the FA. They give a precedence relationship between user-defined functions. In a second time, a task set is composed by grouping the user-defined functions (using some heuristics) of a transaction. This task set is the input of a tool that tries to find an allocation of tasks to processing nodes and priorities to tasks and messages, so that the resulting OA meets all the end-to-end timing constraints (the tool combines constraint programming and schedulability analysis (Cambazard et al., 2004)). For

[3] OSEK/VDX is a set of specifications for a real-time runtime platform dedicated to in-vehicle embedded systems. Homepage: `http://www.osek-vdx.org`.

more complex properties, it is possible to use (for instance) SETPN analysis.

The logical follow-up of the works exposed above concerns code synthesis: given a set of files containing the source code of user-defined functions and the description of a valid OA, synthesise the code of the tasks, as well as the configuration files for the RTOS and communication protocols. A tool has also to be developed to ensure the compliance between the code of the user-defined functions and their model (used for the design of the OA). Presently, these problems are not explored within REACT.

In the next sections, we illustrate the process exposed here, from the first design of the FA to its validation.

3. CLARA: the ADL

According to (Medvidovic and Taylor, 2000), an ADL is a language that provides features for modeling a software system's conceptual architecture. Classically, its building blocks are *components*, *connectors* and *configurations.* Within the context of REACT, we have defined an ADL, CLARA, to describe the functional architecture of reactive systems (Durand, 1998). CLARA stands for "Configuration LAnguage for Real-time Applications".

While defining CLARA, we paid a special attention to the description of the control flows. Compared to other ADLs, it allows to express complex synchronisation and activation laws. It provides also some support for the description of the behaviour of the components and for the expression of real-time requirements (timing constraints) and properties (time budgets).

We illustrate[4] the concepts and abstractions of CLARA through the design of the FA of a small reactive embedded system (see shaded frames). The parts of the text related to this design are embedded within coloured panels. Moreover, to facilitate the global understanding, we give on fig. 2 its final functional architecture.

3.1 The components

Five component families are proposed: activity, occurrence generator, shared resource, shared variable and system. Within each of the four first families, the architect must define at first a set of types that will be used (through instantiation) in the description.

[4] CLARA has both a textual and a graphical syntax. In this paper, we use mainly the graphical one.

Terms of the problem: We consider a simple feedback control loop: two sensors measure the value of the controlled variable (its value is not spatially homogeneous) and an actuator sets the manipulated variable to the computed value. Moreover, the state of the controlled process has to be displayed to an operator (HMI).
The control loop must be triggered every $10ms$, with an end-to-end deadline equal to the period (stringent constraint). The HMI must be updated every $20ms$, with a deadline equal to the period (soft constraint: a period on two can be missed).

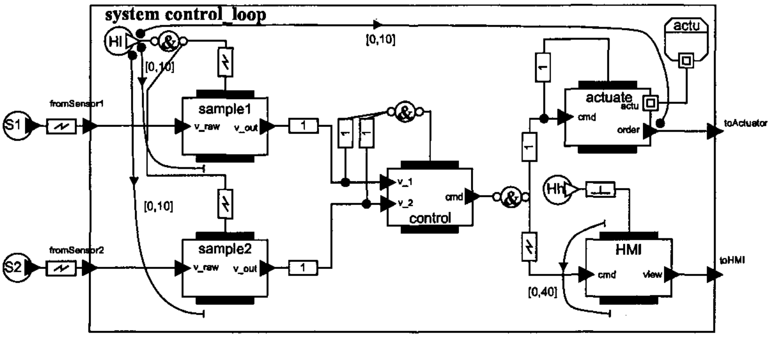


Figure 2. Description of the control loop in CLARA

The *activity* family denotes active components. It can be either *atomic* or *composed* (in our example, we only use atomic activities). For short, atomic activities are the basic building blocks of the architecture and composed activities introduce abstraction levels through hierarchy and encapsulation.

An activity has two interfaces: control and interaction. The *control interface* has only two ports: *start* (input port) and *end* (output port). *start* is used to attach an activation law; *end* is used to signal the end of the execution. From a behavioural point of view, they control the transition from "not operational" to "operational" state [5]. The *interaction interface* is a user-defined set of directed exchange ports (to transfer data or signals) and a set of access ports (to access shared resources or variables). The graphical representation of an activity is given figure 3: cmd and order are data exchange ports, actu is a resource access port.

When it becomes operational, an (atomic) activity executes a finite sequence (fig. 4). The actions can be user functions for which an execution time budget [6] has to be given (a closed interval that will be used for

[5] The execution of an activity instance is not reentrant.

[6] If it is planned to use an heterogeneous runtime platform, budgets are couples (function, processor).

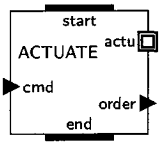


Figure 3. Graphics of an activity

```
name ACTUATE;
sequence {
read(cmd);
access(actu.get);
call(translate_command,0.5,1);
access(actu.release);
write(order);
}
```

Figure 4. Behaviour of an activity

verification purpose[7]). To ensure consistency between the model and the implementation, the budget becomes a requirement (i.e. the "Best Case Execution Time" and the "Worst Case Execution Time" of the function must be within the interval). Other actions are invocations of interaction services (eg. *read(cmd)* or *access(actu.get))*. The invocation of the control services (on port *start* and *end*) are implicit.

All control and interaction service invocations are synchronous. They can be blocking, depending on the behaviour of the connector attached to the port (see below). This enforces the designer to give a complete specification of the application control flows.

Specification of the activities: The FA contains 5 (atomic) activity (see fig. 2):

- sample1 and sample2 (two instances of the same type) read the controlled variable value on v_raw, translate it into a computation-friendly format and write the result on v_out;
- control computes the command from two values (read on v_1 and v_2) and writes the result on cmd;
- actuate controls the actuator. It reads the command on cmd, translates it into an actuator-friendly format and writes the result on order;
- HMI produces (a part of) a synoptic of the controlled process and sends it to an external display equipment. It reads the newly computed command on cmd and writes the new view on view.

The *occurrence generator* (OG) family denotes data or signal sources, which can be part of the system or its environment. Their interface is made of a single output signal or data port (signal ports are white triangles, data ports are black ones). The associated graphics is a circle containing the name of the instance and an output port. An OG can be periodic or sporadic. A periodic OG can only produce signals. Its behaviour is defined through its *period* attribute (cycle time). A sporadic OG can produce signals or data. Its behaviour is given as a sequence

[7] Closed intervals implicitly forbid the use of blocking calls in user functions.

of dates (resp. a sequence of couples (date, value)) that denotes the absolute signal production dates (resp. the absolute data production dates and the data values).

Specification of the OG: The control loop has a period of 10ms: it is measured by a periodic OG *Hl*. The HMI has a period of 40ms: the periods being different, we will use another periodic OG (*Hh*).
Furthermore, there are two sporadic data sources in the environment (one for each sensor) modelled by *S1* and *S2*. We don't define their behaviour and we don't need it (for much of the analysis work to perform) because the control flows of the control loop are not synchronised with the production dates of these OG.

The *shared resources* and *shared variables* denote "passive" entities. Their interface is composed of a single access port (containing subports *get* and *release* for resources and *read* and *write* for variables). The graphics are an oval for a resource and an octagon for a variable, decorated with the instance name and the access port. The access policy is an attribute. The set of predefined values contains mutual exclusion, write exclusive / read many, etc.

Specification of the shared resources: We use a shared resource to control the access to the actuator (access policy: mutex; name: actu). Although it is not shared by several activities, it is included: (i) to illustrate the concept of shared resource in CLARA, (ii) to reference its name in the description of the behaviour of actuate, (iii) to anticipate further extensions of the architecture (for instance adding an activity that uses the same actuator).

The last component family is the *system* family, used to define the boundaries and the interface of the control system under design. Each architecture contains exactly one system component. In fig. 2, the system is named *system_control_loop*.

3.2 The links

In CLARA, a *link*[8] is used to connect a set of output ports to a set of input ports. Beyond to specify which components interact, it states the interaction policies that are used in terms of control flow between the "producers" and the "consumers". A link is built from a set of basic building blocks that allow the specification of very simple as well

[8] In the literature, "connector" is used rather than "link" . However, as "connector" denotes a specific CLARA object, we use "link".

as very complex policies[9]. These blocks are: protocols, connectors and operators (not used in this paper).

A *protocol* has a producer hook and a consumer hook. Each one is associated to a service (production or consumption) and is attached to one (and only one) connector (see below). A protocol synchronises its producers with its consumers according to a specific policy. At the present time, a set of pre-defined protocols is proposed: rendez-vous, transient, blackboard, blackboard with consumption and mailbox. Graphically, it is a small rectangle with a specific symbol inside. The example uses: mailbox (symbol: a number that is the size of the box), blackboard (symbol: a lightning) and transient (symbol: a peak in the middle of a flat line).

A *connector* connects ports to protocol hooks. A simple connector is just a wire (concerning both graphics and behaviour). For more complex connexions, complex connectors have been defined:

- conjunctive connectors (a circled &) for "1 port to n hooks": production (resp. consumption) requests are broadcasted to all protocols; a single acknowledgement is delivered to the caller when all protocols have acknowledged.
- selective connectors (a circled vertical dash) for "n ports to 1 hook": each production (resp. consumption) request is delivered to the protocol and the acknowledgement is sent to the original caller (concurrent requests are serialised).
- hybrid connector, which is (graphics and behaviour) the "merging" of a conjunctive connector and a selective connector.

3.3 The configurations

In the ADL ontology, a configuration is a bipartite graph of components and connectors that describes (a part of) the architecture of the system. The system of fig. 2 is a configuration. As CLARA targets real-time reactive systems, it supports the expression of real-time constraints at the configuration level. More configuration-level facilities might be offered in the future, depending on the needs detected during the on-going case-studies.

A real-time constraint is expressed on events that are observable at the architecture level: production or access request and acknowledgement. A constraint can be:

[9] There is a list of link patterns that are forbidden because they lead to structural deadlocks.

Specification of the links: At first, we specify the data exchange links.

- from sample_1.v_out to control.v_1 (simple link): we use a 1-mailbox protocol: every produced value must be consumed before the delivery of the next one. We do the same with sample_2.
- from control.cmd to actuate.cmd and HMI.cmd (complex link, conjunctive connector on producer side): on the one hand, actuate must consume every command (before the production of the next one) so we use a 1-mailbox protocol; on the other hand, HMI reads the value "when it wants" and is allowed to loose some occurrences so we use a blackboard protocol.

Then we specify the activation laws.

- Hl output signal activates sample_1 and sample_2 (complex link, conjunctive connector on producer side): a blackboard is used and a timing constraint will require that every occurrence is consumed; control is activated each time there are new values on v_1 and v_2 (complex link, conjunctive connector on consumer side); actuate is activated each time there is a new value on cmd (simple link).
- Hh output signal activates HMI (simple link): a transient protocol (without memory) is used. Thus, some occurrences can be lost (HMI must be waiting for the signal to catch it). A timing constraint will require that two consecutive occurrences are not lost.

At last, we specify that the system asynchronously reads the values produced by S1 and S2, using blackboard protocols.

- absolute: the first occurrence of an event must occur in [dmin, dmax] where dmin and dmax are dates;
- relative: the delay between two consecutive occurrences of an event must me in [dmin,dmax] where dmin and dmax are delay;
- causal: the delay between the i^{th} occurrence of a source event and the i^{th} occurrence of a target event must be in [dmin, dmax] where dmin and dmax are delay.

The graphic is a curved line between the involved ports. At the extremities of the line, a bullet denotes a req (request) event, a dash denotes a ack (acknowledgement) event. The interval labels the line. For causality constraint, an arrow indicates the direction. This notation is sufficient for end-to-end deadlines and simple real-time constraints and can be used by non specialists. However, it lacks the expressiveness of TCTL, the possibility to express probabilistic QoS requirements, ...

4. Validation of the functional architecture

An architecture provides with a comprehensive description of the system. This description must be validated before to engage the next design

Expression of the timing constraints: There are three constraints:

- the control loop execution must complete at most $10ms$ after its last activation (stringent constraint). This is a causality constraint between Hl.out.req and actuate.order.req;
- to avoid the lost of occurrences of the control loop clock, the execution of sample1 and sample2 must complete at most 10ms after the clock period: two causality constraints between Hl.out.req and sample1.end.cnf and between Hl.out.req and sample2.end.cnf;
- the HMI activity must not lost two consecutive occurrences of Hh.out. We translate this constraint as a deadline on its execution: once operational, it must finish before $40ms$ (twice the period of Hh). A deadline is a causality constraint between start.cnf and end.cnf.

step. For critical system, the use of formal methods in the validation process is mandatory. These methods can be used only if (a subset of) the ADL has a formal semantics. Moreover, it makes sense only if a rigorous approach is followed for the continuation of the design process, to ensure that the successive refinement steps (up to the binary code) preserve the properties stated at the upper level. The operational semantics of CLARA is given by means of (time) Petri nets (TPN). We will first introduce (informally) this semantics and expose how a CLARA architecture is translated into a TPN model. Then, we will show some analysis possibilities on our example. For PN, useful definitions and theory can be found in (Murata, 1989). For TPN, see (Berthomieu and Diaz, 1991).

4.1 TPN model of a CLARA architecture

The translation from a CLARA description to a TPN is done in two steps. At first, every entity (activities, shared variables, protocols, connectors, etc.) is associated to a TPN pattern. If the behaviour of the entity is predefined or defined through simple parameters (e.g. shared resources), a predefined pattern is used. For more complex entities (activities and aperiodic occurrence generators), a pattern is generated from the textual behaviour description. Then, a global TPN is built by merging the elementary patterns, according to the composition rules specified in the CLARA description. A prototype tool has been designed that performs the translation (the TPN follows the input format of ROMEO[10]).

[10] ROMEO: `http://www.irccyn.ec-nantes.fr/d/en/equipes/TempsReel/logs/software-2-romeo`

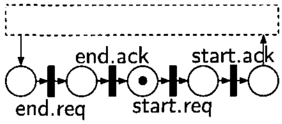


Figure 5. Control interface

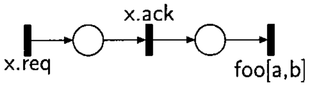


Figure 6. **read**(x); **call**(foo,a,b)

Fig. 5 shows the TPN pattern associated to an activity. The visible transitions model the (implicit) interaction on the start and end ports. The dashed box is to be replaced by a pattern corresponding to the activity behaviour (see fig. 6): an interaction on port x gives rise to a pair of transitions (x.req: interaction request and x.ack: interaction acknowledgement); the invocation of the user-defined function foo gives rise to a single timed transition foo[a,b] where [a,b] is the time budget allocated to the function foo.

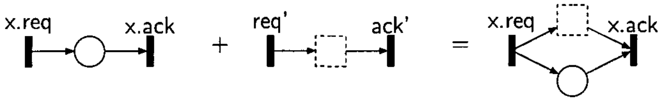


Figure 7. Connection of a port to a link

Fig. 7 illustrates the merging step. The x.req and x.ack transitions are respectively merged with transition req' and ack' of the connector attached to port x. The same mechanism is used to make the connections between all the entities.

The TPN corresponding to our example has 91 places and 71 transitions. This is obviously "big". This is a consequence of the "naïve" translation performed by the tool. Indeed, most of the places and transitions are withdrawn by the usual static reduction rules (Murata, 1989) applied before analysis (presently, the reduction is handmade).

4.2 Validation of the candidate design

At this design level, the validation concerns some functional properties and the consistency between the timing constraints and the allocated time budgets. To achieve these goals, we use presently the tools ROMEO, CADP[11] and TINA[12]. ROMEO computes (among other things) the marking graph of a PN. TINA computes (among other things) its structural properties. CADP allows to perform a wide set

[11] CADP: `http://www.inrialpes.fr/vasy/cadp/`

[12] TINA: `http://www.laas.fr/tina/`

of analysis on labelled transition systems (ROMEO and TINA can output the marking graph of a PN in CADP format).

At first, the timing informations are discarded and we consider the classical PN theory. The goal is to state properties on the FA that will be verified by any further correct refinement (a system the behaviour of which is simulated by our PN for the events that are observable at the FA level). This "weak" equivalence relation limits us to the analysis of safety properties. As an example, we will use deadlock freedom analysis.

Then, the usual reduction rules are applied onto the PN. It does not only reduce the size of the model but also produces a bounded PN (clock modeling produces unbounded marking when the time is not taken into account). Notice that this transformation preserves: liveness, safeness and boundedness (for the places still present in the reduced model). Concerning our example, the reduced PN has only 42 places and 25 transitions. Its marking graph has 26,124 states and 136,204 transitions. There is no deadlock state. Some more complex properties might be verified using CADP model-checking facilities (e.g. "for each pair of input value, there is exactly one actuation"). They can be carried along the design process as long as they can be expressed as safety properties.

We will now use the structural analysis of the PN, performed by TINA. We expect our system to have two periodic end-to-end transactions: control loop and HMI update. To validate this assumption, we look after the T-semi-flow generating sets (a positive T-semi-flow denotes a cyclic behaviour of the system). Five positive T-semi-flow generating sets exist. All of them are feasible (i.e. there exists at least one run from the initial state whose firing vector corresponds). Three of them are artefacts of the model; The two others correspond respectively to the cyclic execution of the control-loop transaction and the HMI transaction. If we let aside the artefacts,the description corresponds to our expectations. The artefacts are caused by the modelling of periodic occurrence generators: as we do not take time into account, the "y.expire" transitions (corresponding to the clock expiration) can be fired from any marking. If we try to remove the artefacts by controlling the transitions, we reduce the set of possible implementations (the behaviour of which will be simulated by the model): we would make the strong hypothesis that some part of the transaction is always executed within one period of the occurrence generator. Such a reduction of the design space is obviously not desirable at the FA level since it could potentially exclude all the valid solutions.

Let's consider the consistency between the timing constraints and the time budgets. It is clear that -at this level- it is not possible to assess the timing correctness: it cannot be done without taking into account the operational characteristics (mapping of functions to tasks, of tasks

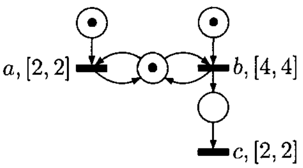
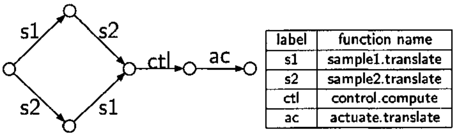


Figure 8.



label	function name
s1	sample1.translate
s2	sample2.translate
ctl	control.compute
ac	actuate.translate

Figure 9.

to nodes, scheduling policies and parameters, etc.). However, a first analysis can be driven to check that there may be an implementation, with these time budgets, that could meet the constraints (*i.e.* we check a necessary condition). We have to find a "best case" for the execution time of the sequence (of transitions) bounded by e_s (the "starting" event of the constraint) and e_c (the "closing" event). "Best" means that every possible implementation will produce highest or equal execution times.

First, notice that the study of the state class graph of the TPN doesn't give us a best case from an operational point of view. To get convinced, consider a system with two concurrent tasks T_1 and T_2. T_1 executes the action a and completes. T_2 executes b then c and completes. Actions a and b need a shared resource and are mutually exclusive. The TPN of fig. 8 is a model of this system (where one can see the execution time of each action). If the deadline of T_1 is 10 and the deadline of T_2 is 7, the analysis will show that T_1 always meets its deadline whereas T_2 always misses its deadline. Nevertheless, this system is schedulable, e.g. with a fixed priority scheduler and $prio(T_2) > prio(T_1)$.

Let's go back to our consistency checking. Because we don't know the operational characteristics of the system, the only information that we can take into account is the precedence relation between the executions of the user-defined functions of a same transaction and their execution time budgets. Thus, we must extract the graph of the precedence relation from the PN marking graph. Then, the edges corresponding to the invocation of user-defined functions (involved in the constrained transaction) are weighted with the upper bounds of the function time budgets. If no information is known about the RP, we have to compute the value of the longest path in the graph (to take into account an optimistic true parallelism). In case of a mono-processor RP (we consider this hypothesis for our example), we just have to sum up the weights (the execution sequence is a sequential chain) and compare the result to the upper bound of the constraint. Any implementation for which the computation times actually reach the upper bounds of the execution time budget will inevitably executes this sequence of functions with a *higher or equal* execution time (in the implementation, the execution will be delayed at least by the overhead of the RTOS services, and may be by the network and/or the execution of some transactions of higher priority). Thus, timing consistency between constraints and budgets occurs *when the value of the path is less or equal than the upper bound of the constraint.*

For our example, the work is trivial for three of the constraints (they involve only one atomic activity and thus no concurrency). For the fourth one (between, HI.out.req and actuate.order.ack), the problem is a bit more complex. Even if it can be done "by hand", we illustrate how to use tools to automate the work.

First, we know that the control loop transaction is cyclic and has no transitional mode. Thus, we can limit our study to the paths in the marking graph that correspond to the first instance of the transaction (any further instance will exactly have the same set of runs). At first, we hide all the labels that are not useful (i.e. not corresponding to e_s, e_c or any function of the transaction where e_s is the source event and e_c the closing event of the constrained sequence). Then, we extract the paths that match $i^*.e_s.(\sim e_s \& \sim e_c)^*.e_c$ (a path starting with a sequence of silent actions, then e_s, then any action that is not e_s and not e_c, then e_c) using CADP. We forbid the paths containing more than one occurrence of e_s in order to eliminate the interference of some other instance of the transaction. From this set of paths, we obtain the labelled transition system (LTS) shown on fig. 9. The length of the chain is 6 ms and the upper constraint is 10 ms. We conclude that the solution space may contain some valid implementations.

We now have a candidate FA, together with a set of extra-functional characteristics, that have been validated. We have shown that it will not deadlock and that the values of the extra-functional properties seem to be consistent.

As stated in section 2, the next step is to map the candidate FA onto the RP, so as to obtain a candidate OA. This candidate OA has to be validated too, especially with regards to extra-functional properties that can be assessed only at this level. However, due to space limitation, we will not detail this stage in this paper.

5. Conclusion

In this paper, we have described the goals of the REACT project. We have exposed (i) the process that it adopts for the architectural design of real-time systems; (ii) its ADL CLARA; (iii) the validation of CLARA architectures through formal analysis techniques.

In (Faucou et al., 2004) (extended version of this paper), a comparison is done between CLARA and some related projects (all of them being discussed in other papers included in this volume): MetaH/AADL (Binns and Vestal, 2001), COTRE (Farines et al., 2003) and EAST (Debruyne et al., 2004). Although the development of REACT is certainly less ahead than these projects, we have highlighted some of its specificities. Hence, the link mechanism of CLARA allows to easily describe complex multi-components synchronisation patterns and enforces the designer to specify and validate the control flows at the architecture level (obviously a good practice for real-time system design) . Moreover, compared to MetaH/AADL or Cotre, REACT can be used at a higher design level (FA rather than SA). This allows us to investigate the synthesis of "valid by construction" operational architecture and to propose in the future a coherent and automated toolset for the rigorous design of real-time systems.

References

ARTIST (2003). Component-based Design and Integration Platforms. Technical Report W1.A2.N1.Y1, ARTIST - Advanced Real-Time Systems - IST project.

Berthomieu, B. and Diaz, M. (1991). Modeling and verifications of time dependent systems using time Petri nets. *IEEE TSE*, 17(3).

Binns, P. and Vestal, S. (2001). Formalizing software architectures for embedded systems. In *EMSOFT 2001*, volume 2211 of *LNCS*. Springer.

Cambazard, H. et al. (2004). Decomposition and learning for a hard real-time task allocating problem. In *CORS/INFORMS Joint International Meeting.*

Debruyne, V. et al. (2004). EAST-ADL, an Architecture Description Language, Validation and Verification Aspects. In *IFIP 2004 WADL.*

Durand, E. (1998). *Description et vérification d'architectures d'application temps réel: CLARA et les réseaux de Petri temporels.* PhD thesis, École Centrale de Nantes.

Farines, J. et al. (2003). The COTRE project: rigorous software development for real-time systems in avionics. In *27th IFAC/IFIP/IEEE WRTP'03.*

Faucou, S. (2002). *Description et construction d'architectures opérationnelles validées temporellement.* PhD thesis, Université de Nantes.

Faucou, S. et al. (2004). REACT: an ADL centric approach for the rigorous design of real-time embedded systems. Technical report, IRCCyN. (to be published).

ISO (2003). *ISO 11898 : Road Vehicles - Controller area network (CAN).* ISO.

Kodase, S. et al. (2003). Transforming Structural Model to Runtime Model of Embedded Software with Real-time Constraints. In *DATE'03 Designer's Forum.*

Medvidovic, N. and Taylor, R. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE TSE*, 26(1).

Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, 77(1).

Roux, O. and Déplanche, A. (2002). A T-time Petri net extension for real-time task scheduling modeling. *European Journal of Automation (APII-JESA)*, 36(7).

SESSION 2: SPECIFICATION AND DESIGN