Integrating the Synchronous Paradigm into UML: Application to Control-Dominated Systems

Charles André, Marie-Agnès Peraldi-Frati, and Jean-Paul Rigault

Laboratoire Informatique Signaux et Systèmes (I3S)
Université de Nice Sophia Antipolis CNRS UMR 6070
{andre,map,jpr}@i3s.unice.fr
http://www.i3s.unice.fr

Abstract. The Synchronous Paradigm proposes an abstract model integrating concurrency and communication, deterministic thus simple, semantically well-founded thus suitable for formal analysis, producing safe and efficient code. However combining this model with the object-oriented approach is still challenging. This paper explores how an UML-based methodology can be set up, making it possible to use the Synchronous Paradigm in combination with other (more classical) techniques to develop control-dominated systems. It addresses the issue of representing behavior in a semantically sound way using the synchronous models, of relating behavior and structure, and of mixing synchronous and asynchronous behavior though an extended notion of (ROOM-like) «capsules», the synchronous islets. We also briefly mention the extensions and modifications in the UML meta-model necessary to support this methodology.

1 Control-Dominated Systems

Control-dominated systems are frequent in real time applications, especially in embedded ones (automobile, air and space, process control, integrated manufacturing, robotics...). These systems display a unique combination of characteristics. They are usually multi-modal, alternating their functioning between several operating modes. They are reactive which implies to deal with concurrency, communication, and preemption. They have to satisfy stringent constraints (correctness, response time) thus they have better be deterministic. Finally they may be life or mission critical and consequently they should be submitted to formal verification and validation.

The reactive nature and its consequences are the major reasons why the behavior of these systems is delicate to handle. A way of reducing the behavioral complexity is to have them evolve through successive phases. During one phase, only the external events which were present at the beginning of the phase and the internal events that occurred as a consequence of the first ones are considered. The phase ends when some stability (fixed-point) has been achieved (or when some external "clock" decides that it is over).

We call such a phase an *instant*. Indeed, during such an instant, the time seems to be suspended (the external events are frozen). An instant-based system will be called a *synchronized system*. The so-called synchronous models are among these instant-based schemes.

Atomicity of reactions is often necessary in reactive systems. The UML "run-to-completion" is a way to satisfy such a requirement. The synchronous models satisfy it, "by construction".

These dominated control systems are not only delicate behavior-wise. The complexity of modern systems makes it even more necessary to analyze, design, implement, and verify these systems in a structured (hierarchical) way, hence the need for enforcing the relationship between their structure and their behavior.

Thus we need to represent the architecture of the system as well as its behavior, in an integrated way. Combining powerful and semantically sound models of behavior with the object-oriented approach is certainly a path worth considering. The UML *de facto* standard, which proposes both behavioral and architectural points of view, is certainly the notation of choice.

This paper is about extensions and modifications to UML to integrate the power of the Synchronous models. The following section (2) describes the common concepts of synchronized models and presents the definition of their ideal (limit) instance: the Strictly Synchronous Model. Section 3 is the core of the paper; it deals with extending UML to support the synchronous paradigm. The next section (4) illustrates the approach on an example. We finally conclude and propose some forthcoming developments.

2 "Synchronized" Behavioral Models

2.1 Common Concepts

Signals and Events Our systems react to input events by producing output events. An event corresponds to the occurrence of a *signal*, and a signal may carry a value

Instant A synchronized system evolution concentrates into successive instants which are defined system-wide[1]. An instant is characterized by a Begin Of Instant (BoI) and an End Of Instant (EoI). BoI and EoI are in monotonic order and stricly alternating (instants cannot intersect). The input events are frozen at BoI; the output events are available at EoI. Between two instants (more specifically, between EoI_i and BoI_{i+1}) nothing happens. The notion of an instant gives a clear meaning to simultaneity (everything which happens during the same instant), and to past and future. "Strict future" designates the next instant(s).

Broadcast and Combine During an instant, the various (concurrent) parts of the system can produce and absorb events. The events (external as well as internal) are broadcast (that is their target is not explicitly designated and an event is available for any process waiting for it in the same instant). When the same signal is emitted several times during an instant, it is *combined*: there is only one occurrence, and the value of the signal is a commutative and associative combination of all the values corresponding to the individual emissions (the combination function is usually part of the signal declaration). Of course, not all signals are capable of combining their values. Multiple simultaneous emissions of some signals may be forbidden.

[1] Well, in fact for this part of the real system that is to be considered synchronized.

Semantics The semantics relies on the search for a (unique) fixed-point during each instant. For the imperative synchronous languages and models, the semantics can be *constructive*: it is established by propagating positive facts (e.g., the effective presence of a signal) and possibly negative ones (e.g., the certainty of the absence of a signal) during an instant.

2.2 Variants

Strict Synchronous The strict synchronous model is the limit case for synchronized systems. It corresponds to $BoI = EoI$. Thus an instant has a null duration, the reaction is instantaneous, the instant-wise simultaneity is true simultaneity. It is supported by both imperative languages like Esterel [7], and data flow ones like Lustre and Signal [17].

The strict synchronous model proposes both a model of concurrency and a model of communication. The concurrency model is deterministic, hence simple, and yet powerful; it is abstract (no need for extra mechanisms like semaphores, monitors...); and it is semantically well-founded. Signals abstract communication: broadcast and combine make it simple to communicate, efficient instantaneous protocol dialogs are easy to set up (if not to understand).

As a consequence, a synchronous program may be viewed as a set of (very) lightweight processes, composed in parallel and communicating through broadcast signals. The simplicity and well-foundedness make it possible to consider formal analysis. Tools can produce safe, efficient and compact code.

However there are a number of drawbacks. First the semantic fixed-point does not always exist; thus some programs are (semantically) incorrect. This is due to “temporal paradoxes” also known as “causality cycles”. These can be detected at compile-time but are not easy to avoid. They are indeed a programmer’s nuisance. Second, strict synchronous languages are good at expressing control, but less at ease for general data manipulation; the formal analysis methods do not handle values very well either. Last, since they imply a sort of centralized coordination (centralized clock), the strict synchronous systems are difficult to distribute.

Relaxed Synchronous Models For practical reasons as well as to cope with the drawbacks of strict synchrony several works have proposed to relax the synchronous model hypotheses.

Execution Machines: A first and compelling practical reason to relax arises when trying to put the synchronous model into practice. Indeed the “real world” is not synchronous, our processors are not infinitely fast, and reactions cannot be absolutely instantaneous. So we need to define *execution machines* that adapt the inherently asynchronous application events to their ideal pure synchronous counterparts. The responsibility of an execution machine is to collect the external input events, to impose BoI and EoI, to decide which external input events will be available for the synchronous program during an instant, and to propagate the output events to other parts of the system.

When designing synchronous execution machines, two important questions must be answered: is the strict synchronous model a reasonable approximation of the reality? is

the machine fast enough (compared to the flow of input events) so that its reaction may be considered of null duration? This is a performance analysis issue, related to "real" time[2]. Discussion about this is out of the scope of this paper. Interested readers may refer to papers presenting software implementations [4] or hardware solutions [6].

Note that there is no need for a *schedulability analysis*, at least within the synchronous program itself. In fact, the control generated by the compiler is a static scheduler of the actions to be performed by the reactive system. By a simple analysis of the generated code, given a target architecture, it is possible to compute the best and the worst case execution time for each reaction [5].

Another reason to relax the model is to cope with the two main drawbacks of pure synchrony: existence of "causality cycles" and difficulty to apply to distributed systems. When relaxing, BoI and EoI are no longer equal ($BoI_k < EoI_k$ of course), and the fixed point semantics has to be revisited.

Since the construction of the fixed-point semantics is based on propagating facts (section. 2) one can obtain several ways of relaxing the model by choosing which facts to propagate, and when they are propagated. These relaxed models demonstrate that the strong hypotheses of the Strict Synchronous Model can be weakened without losing the notions of instant, signals, and broadcast/combine communication. Besides being more realistic, these models avoid the unpleasant temporal paradoxes due to instantaneity. Thanks to their instant-based semantics, they remain relatively tracktable and mostly deterministic, even for complex systems. This point of view has been adopted for long in hardware: synchronous circuits are far easier to design than their asynchronous counterparts. We mention here some of these extensions.

Synchronous Objects: This was one of the first attempt to mix the object-oriented paradigm with the synchronous approach [10, 2]. Synchronous modules are used to define the behavior of reactive sub-systems. Each sub-system is abstracted as a class whose methods are related to signals (receiving or emitting), and whose behavior is expressed by the synchronous code. Instances of these classes (synchronous objects) can be inter-connected to realize more complex reactive systems. The behavior of the whole system can be still a synchronous behavior, provided some assembly rules are respected. F. Boulanger developed a technique to schedule the different objects so that to ensure an instant-based behavior. This scheduling can be either static (a very efficient solution) or dynamic (a convenient solution for rapid prototyping).

Weak Synchronous Systems: F. Boniol [9] defined a process algebra called CoReA (COmmunicating REactive Automata) based on a synchonous semantics of concurrency and a weak synchonous semantics for communication between concurrent processes. Communications take one time unit of a global clock. Boniol's weak synchrony have been successfully applied to distributed applications built on a RTLAN (Real-Time Local Area Network). For instance, L. Blanc in his thesis [8] considers *strict synchrony* for *local communication* and *weak synchrony* for communication over the network. Such characteristics guarantee deterministic behavior for the distributed system from a logical and a temporal perspective.

[2] as opposed to the logical time defined by the event flow in the synchronous model

The quasi synchronous approach: It has been ntroduced by P. Capsi and R Salem [12]. The system is considered as a set of sub-systems locally synchronous that exchange information via a shared memory. Exchanges are periodic. There is no "global clock" in this system as the authors consider that clocks are "quasi equivalent". They have an abstract view of the asynchronism by modeling the "clock drift". They propose an algorithm for the distribution of LUSTRE code that guarantees disjoint acquisition phase and transitions executions.

Reactive Objects: They have been ntroduced by F. Boussinot et al. [11]. The reactive object model is an object-based formalism matching the reactive paradigm. Methods can be invoked using instantaneous non-blocking send orders, which are immediately processed (that is, processed during the current instant); moreover, a method cannot execute more than once at each instant. In this model, a process[3] is executed until it blocks, waiting for some event which is not yet known to be present in the current instant. Only the positive fact are propagated during an instant, possibly unblocking a waiting process. The instant terminates when all processes are blocked.

In this paper we shall not deal with integrating the relaxed models into UML. We shall only concentrate on the integration of the strict synchronous model. Indeed the latter should be considered as a *foundation model*. All relaxed models should exhibit the following suitable property: if there is a strict synchronous solution[4], there should be (at least) one relaxed solution. Thus it is important to first consider the impact of integrating the ideal model into UML, before looking at its variants.

3 Integrating the Synchronous Paradigm into UML

3.1 Relating Behavior and Structure

The strict synchronous model is well-adapted to structuring the control aspect of a system, but fails at expressing architectural issues. The modules, or processes, or parallel branches that are found in the synchronous languages are not objects, nor are they associated with objects.

There have been several early attempts to add object-oriented capabilities to the synchronous model [10, 2]. Besides the gain in expressing the architecture and the relationship between architecture and behavior, they make it possible to represent systems that are still control-dominated but that exhibit a significant data processing activity.

3.2 UML and Control Systems

The Unified Modeling Language (UML) is now the *lingua franca* of (object-oriented) software engineering and, in the real-time domain as well as in all the other domains

[3] Remember these are very lightweight processes, with a granularity of possibly a few basic statements.

[4] which is unique by definition

of the software industry, using the UML is becoming a key issue. UML offers multiple points of view on the system, and some of these points of view are rather universal.

However, when it comes to modeling specific properties of real-time systems, and in particular their behavior, the models proposed by UML are too general to fit the requirements. Of course several real-time extensions to UML (or "profiles") have been proposed or are underway (see for instance [19] for a (non) exhaustive list). Let us mention two of these, among the most influential ones. Real-Time UML (RT-UML) [14] demonstrates how UML can be used to model real-time systems. Although a complete method is described, almost no extensions to UML are introduced (except the notion of Timing Diagrams which is perfectly orthogonal to the UML models). The models are simulable and executable. RT-UML is implemented in the Rhapsody commercial tool (from I-Logix). UML for Real-Time (UML-RT) [23] goes somewhat further. It is an UML adaptation of the ROOM method [22]. It introduces class stereotypes (capsule, protocols, ports, connectors) corresponding to frequent real-time entities and extends the UML collaboration diagrams to represent the structural view of the system (structure diagrams). Here again the models are simulable and executable and UML-RT is implemented in the Rose Real Time tool (from Rational).

ACCORD/UML [16] proposes another interesting approach where a system is a collection of active and passive real-time objects. The messages exhanged among the objects may be temporally constrained. They correspond to method call of signal sending.

All these extensions and adaptations of UML do not rely on the Synchronous Model. To represent the behavior of the system, they use the classical UML means: state machines (a variant of StateCharts [18]) and interaction diagrams. These models have a non deterministic semantics and thus they are not convenient for representing stringent constraints and, even worse, not adapted for performing formal property analysis.

3.3 Adapting UML to Comply with the Synchronous Model Hypotheses

In a recent work, a team from Dassault Aviation [20] proposed a close integration between the Strict Synchronous Model and the UML. They defined object-oriented extensions for the Esterel language (Esterel++, a pre-processor to pure Esterel). They used SyncCharts, a graphical companion to Esterel [1], for modeling the behavior of objects and they borrowed from ROOM/UML-RT the notions of capsules, protocols... as special kinds of classes but with a somewhat different semantics, more suited to the synchronous world. They also adopted the ROOM/UML-RT notion of a structure diagram and they adapted Rational Rose to represent these concepts and to generate Esterel++ code. The SyncCharts, which cannot be represented or edited within Rational Rose, are drawn, simulated, and verified using the Esterel Studio commercial tool [15]. The corresponding combination of tools is now routinely used within the Dassault company.

Dassault's approach is indeed a first and promising attempt, but more work needs to be done to have a smooth integration of the synchronous paradigm into UML. The rest of this subsection proposes some tracks for this.

We consider here the architectural representations of UML (class and structure diagrams) and the behavioral ones. The other points of view of UML (use case, component,

deployment) are either usable unchanged or their semantics is not precise enough to be impacted by the real-time and synchrony issues.

Class Diagram As far as the description of classes is concerned, the UML diagrams can be adapted easily to the Synchronous model. We need classes for reactive objects and at this end we borrowed the notion of a «capsule» from ROOM/UML-RT [23]. It is a very convenient model for representing objects with complex communication. A port is typed by a «pluget»[5], a special kind of «interface» consisting of a list of signals.

All these extensions are well known, they are not specific to the synchronous model, and they can be realized using UML stereotypes. To go further, we specialize these concepts for the synchronous case:

- «S_signal» (Synchronous signal) is a stereotype denoting a class of synchronous signals that can be broadcast and combined. UML does not seem to support broadcast *per se*; signals can be sent to several targets at the same time, but apparently the targets have to be designated (diffusion list as opposed to general broadcast). And when it comes to signal combination, this is a notion UML is oblivious of. Thus «S_signals» are not a specialization of the Signal meta-class.
- «S_capsule» (Synchronous capsule) is a stereotype denoting a «capsule» which has a specific port (its «clockPort»); through the clockPort, an «S_capsule» object will receive its begin and end of instant signals.
- «Islet» is a stereotype designating a collection of «S_capsule»s sharing the same «clockPort» generator; thus the islet provides the (synchronous) context; all its «S_capsule»s have the same notion of an instant and the «Islet» has the responsibility to provide BoI and EoI.

Note that an «Islet» is a regular «capsule», but not an «S_capsule» in general[6]; thus the «Islet»s communicate asynchronously; only within an «Islet» is the communication synchronous.

Figure 1 shows a simplification of extended UML meta-model to support the previously described stereotypes (our work is currently based on the UML 1.4 meta-model). In particular we do not show or detail the hierarchy of «port»s and «S_signal»s.

Structure Diagram Since an «Islet» is composed of «S_capsule»s communicating through their «ports»s, the connections and renaming of signals must be represented by a specific diagram. Such diagrams already exists in ROOM and UML-RT, and there is a proposal to introduce them in UML 2.0. They are called *structure diagrams*.

We need only one extension to the ROOM-like structure diagrams: a notation to represent broadcasting and combining «S_signal»s. It may look like a sort of bus (see figure 6 in next section for an example).

[5] UML-RT/ROOM denotes the same notion by the term «protocol»; we avoided it, since,first, it appears to us as confuse-wise and, second, there are some differences between «pluget» and UML-RT protocols. One of these differences is that our «pluget» are hierarchical: they can contain other plugets, not simply signals.

[6] although it can be an «S_capsule», but with a *different* clock reference

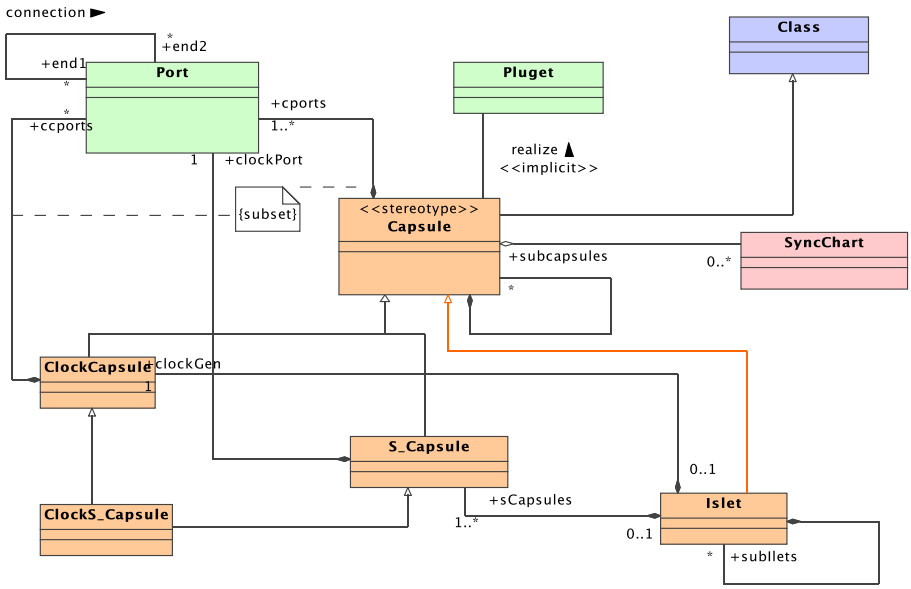


Fig. 1. A UML meta-model for the class stereotypes to support the strict synchronous model

Note that structure diagrams may be used either to represent the instances constituting a (sub-)system (an extended object diagram, showing the interconnections) or the internal organization of the elements constituting an aggregate. In the latter case the components represent roles more than instances. They may be subject to substitution (according to some UML generalization).

Another useful extension would be to represent template capsules, with type parameters as well as integer constants. There should also be a way to represent duplicated elements. We are currently working on such extensions.

Behavioral Models Of course the main impact of the synchronous model is onto the UML behavioral points of view. The latter are of two kinds: the first one describes the internal behavior of (the instances) of a class. The second represents the exchange of messages between class instances.

Intra-objects: UML uses state machines, a variant of Harel's StateCharts [18] to describe the internal state behavior of a class. For our purpose, there are at least two main drawbacks with this state-machine model.

The first drawback defeats UML state-machine use in the strict synchrony hypothesis: these state machines are inherently asynchronous and consequently non deterministic. Their semantics is given in term of waiting queues, without any indication on how and when the events in the queue are handled. The notion of instant and simultaneity are not defined. Relying on such a non-deterministic feature would clobber all

the advantages of the synchronous model (simplicity, determinism, suitability to formal analysis).

The second drawback is that UML state machines do not enforce a strict hierarchical organization: the macro-states are not encapsulated since transitions may freely cross their borders. Of course this feature has a strong power of expression, but it makes the behavior harder to understand (like goto's in a program!), harder to compose, and harder to prove. Indeed the strict compositionality is the key to handle proofs on complex systems.

Thus we use the SyncCharts [1], a graphical companion of synchronous languages. It proposes a purely synchronous vision of the behavior with a strictly encapsulated hierarchy of states. SyncCharts translate directly into Esterel code and thus may take advantage of the simulation and model checking tools that were developed for this language (SyncCharts has been chosen as the main way of describing behavior in the EsterelStudio commercial tool [15]).

However, the SyncCharts model cannot be obtained as a simple specialization of the UML state machines (mostly because of state encapsulation). The corresponding meta-model, which is given on figure 2, corresponds to an heavyweight extension. *Actions* associated with states and transitions, and *triggering events and guards* associated with transitions are not detailed. They have been adapted to support «S_signal». *Stg* stands for *State Transition Graph*, a connected graph made of states and transitions.

Figure 7 in the next section shows an example of a syncChart.

Inter-objects: The communication between objects is usually expressed using UML interaction diagrams. We are mainly interested here in a time- and event-based representation (even though our time is only logical), thus our focus is more on the sequence diagrams than on collaboration ones.

There has been much discussion about sequence diagrams. Obviously the inventors of UML wanted to keep the model of interaction simple, so that it may be used by non computer specialists. This is certainly a point worth considering. However, the model is really too simple, even simplistic, when it comes to represent complex interactions (scenarios): no loops, weak representation of conditions, no structure...

Moreover a sequence diagram presents an ordering of events exchanged between the objects participating to a scenario. But is this order chronological or causal[7]? Are the flows of events universal (a mandatory behavior) or existential (an optional behavior)?

Message Sequence Charts (MSC [21]) have existed before UML sequence diagrams and have a greater power of expression. The recent Life Sequence Charts (LSC) [13] adds to MSC several features, especially the notion of existential and universal behavior.

Our own model, Synchronous Interface Behavior (SIB [3]) is much inspired by MSC. It is even closer to LSC: the two models have been developed independently, and yet they introduce similar notions (structure, existential and universal behavior, ...). SIB adds feature to represent simultaneity and the notion of instant to scenarios. Any SIB can be automatically translated into a semantically equivalent Esterel module. This module is then composed (synchronous parallel composition) with the controller to be

[7] Chronology is not causality. *Post hoc, ergo propter hoc* ("after this, hence because of it") is a sophism that Francis Bacon denounced a long time ago.

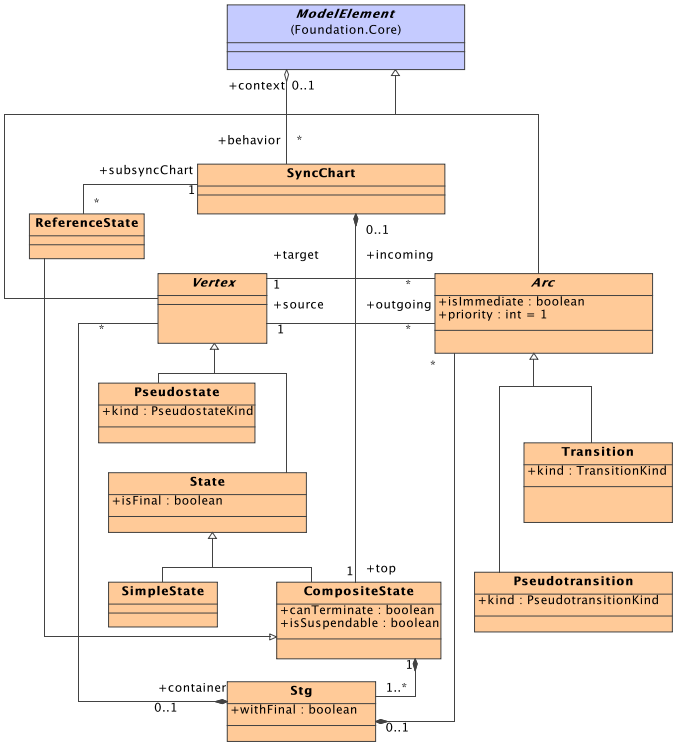


Fig. 2. A UML meta-model for SyncCharts

validated. This module is an observer [17], which can emit a failure or a success signal, according to the observed behavior. A SIB represents either a history the system should perform at least once (existential form) or a safety property (invariant).

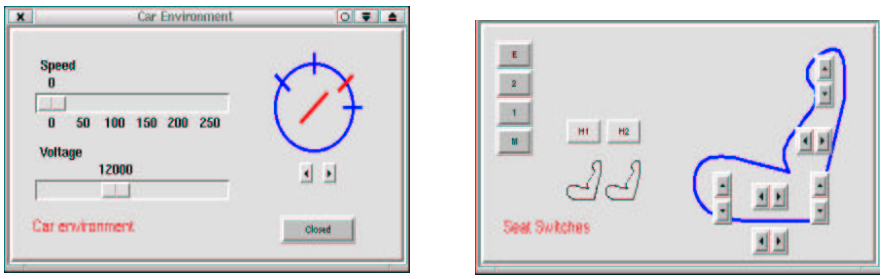
We do not insist on SIB here, since it is a work underway. If we can easily represent the trace of the events at the interface of an individual object (and perform formal analysis on it), we still have to decide (among several solutions) how to represent (graphically) the notion of an instant, when several objects communicate (in a scenario).

4 Example

4.1 Presentation of the Seat Controller

We illustrate our approach with some diagrams extracted from the example of a (luxury) car seat[8] The seat is composed of 6 motors which allow to modify the position of its different parts (distance to the steering wheel, inclination pf the back, heigth of the bottom...). Each motor has three states: stopped, forward, and backward. When running a motor emits a “tick” signal at each revolution. The seat adjustments are limited

[8] The example is inspired by a competiton initiated by Daimler Chrysler.



(a) Environment panel (b) Control panel

Fig. 3. The seat simulation environment

in their magnitude and characterized by a number of ticks. The motors are gathered into 2 groups of 3 motors with activation priority within a group (that is a motor has to be stopped if a motor with an higher priority in the same group gets activated).

The motor activation may be manual, through a control panel. It may also be automatic with several modes: calibration phases, courtesy (e.g., putting the seat backwards when the door is opened), and memorization (e.g., reaching a previously memorized seat position). Finally there are some constraints that some seat functions (heating, displacement) have to obey and that depend on the environment of the car.

Figure 3 describes the simulation environment of the car that we have designed using Tcl/Tk.

4.2 Class and Structure Diagrams for the Seat Controller

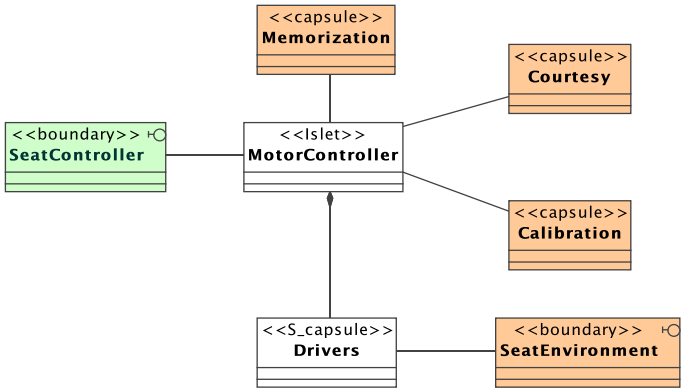
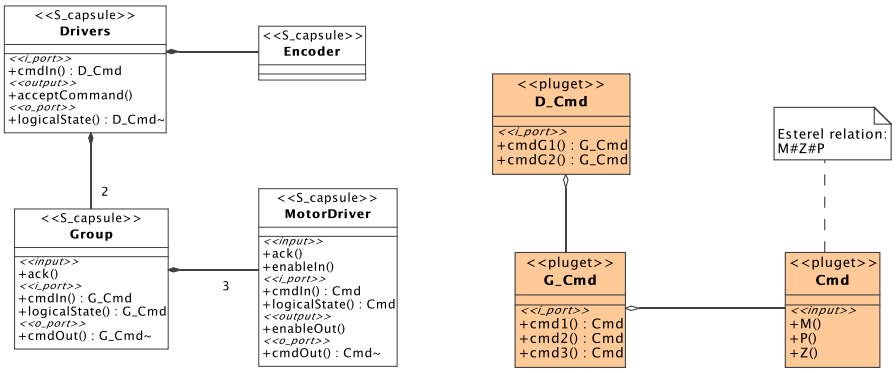


Fig. 4. Top level class diagram for the seat controller



(a) Details of the MotorController «Islet»

(b) The «plugets» used by the MotorController

Fig. 5. The MotorController «Islet» and its «pluget»s

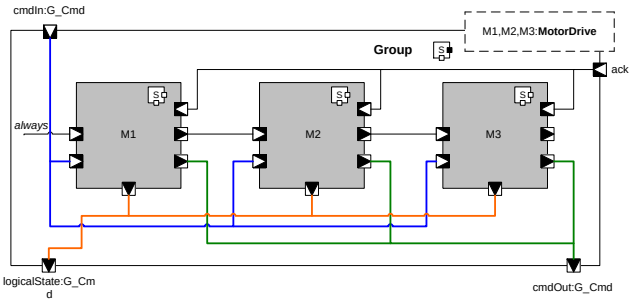


Fig. 6. Structure diagram for a group of three motors

Figure 4 presents a snapshot of the main classes of the application. The details of the MotorController «Islet» and the «plugets» that it uses are described on figure 5. The stereotypes «input» and «output» correspond to simple ports, composed of a single signal whereas «i_port» and «o_port» denote compound ports with several signals. The ports are typed with «pluget»s: for instance, this figure shows that Cmd stands for a combination of three signals: M, P, and Z; the Esterel relation (in the note) expresses that these signal are mutually exclusive (in the same instant). G_Cmd itself is composed of 3 «i_port»s of type Cmd. The ˜ character corresponds to the “conjugated” pluget (obtained by reversing inputs and outputs).

The structure diagram of figure 6 describes the internal organization and connections of the components of a (motor) Group, composed of 3 MotorDrivers. Note the

bus-like notation to represent the broadcast/combine of signals, and also the template-like notation which expresses that it is more a *role* diagram than an instance one.

4.3 Behavioral Aspects of the Seat Controller

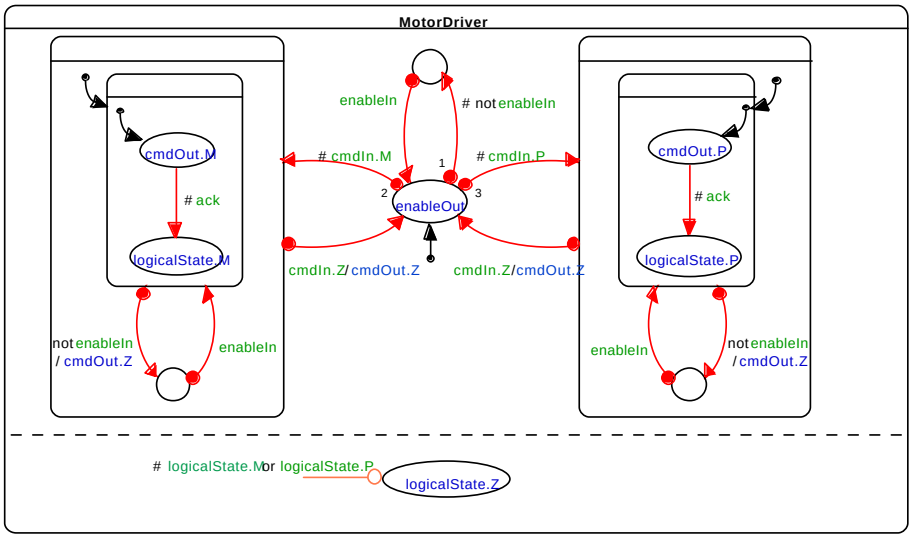


Fig. 7. SyncChart for «S_capsule» Motordriver

As already mentionned we restrict to the state representation (inter-objects). Each «S_capsule» is associated with a syncChart to express its internal state model. As a matter of example, figure 7 presents the syncChart of the MotorDriver capsule (see figure 4). Note the use of the dot notation to refer to the constituents of a port (e.g., logicalState.M).

5 Perspectives

In this paper we described some tracks to integrate the Synchronous Model as smoothly as possible into UML. We try to use, whenever applicable, lightweight extensions. However, the representation of synchronous behavior is still on the heavyweight side. We strive to “alleviate” them, and there might be hope to succeed (at least partially) with some forthcoming UML 2.0 proposals.

The work presented here is still in progress and will follow three main axes. The first is to provide tools to manipulate the synchronous islets, the structure diagrams, and especially the Synchronous Interface Behavior notation. There is still work to do so that SIB may cope with the synchronous hypotheses, in a graphically convincing way. It is mostly a matter of notation and layout, but this is an important issue in modeling.

Second, we wish to extend the model of "synchronous" islets to cope with the relaxed synchronized models. This would open the way to a wider range of applications, including distributed ones.

Third, we need to define a methodology mixing the classical UML notation and the synchronous models (be they strict or relaxed). Indeed, the modeling entities we proposed here are not supposed to replace the classical UML classes, state machines, sequence diagrams, etc. nor are they to supersede well-known extensions such as capsules. We plan to use all of them together, having recurse to the synchronous model only for those parts of the system that are control-dominated, that have to face stringent constraints and formal analysis. Being able to model such a system composed of collaborating synchronous and asynchronous parts is still a challenge.

References

[1] C. André. Representation and Analysis of Reactive Behavior: a Synchronous Approach. In *Computational Engineering in Systems Applications (CESA)*. IEEE-SMC, 1996.

[2] C. André, F. Boulanger, M.-A. Peraldi, J.-P. Rigault, and G. Vidal-Naquet. Objects and Synchronous Programming. *RAIRO-APII-JESA*, 31(3), 1997.

[3] C. André, M.-A. Peraldi-Frati, and J.-P. Rigault. Scenario and Property Checking of Real-Time Systems Using a Synchronous Approach. In *4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Magdeburg, Germany, May 2001. IEEE.

[4] C. André, F. Boulanger, and A. Girault. Software Implementation of Synchronous Programs. In *Proceedings of the Second International Conference on Application of Concurrency to System Design, Newcastle upon Tyne, UK, June 25–29, 2001*, pages 133–142. IEEE Computer Society, 2001. IEEE Computer Society Press Order Number PR01071 Library of Congress Number 2001090878 ISBN 0-7695-1071-X.

[5] C. André and M.-A. Péraldi. Predictability of a RTX2000-based Implementation. In *Real-Time System*, volume 10, pages 223–244, May 1996.

[6] G. Berry. A Hardware Implementation of Pure ESTEREL. Miami, January 1991. ACM Workshop on Formal Methods in VLSI Design.

[7] G. Berry. *Essay in Honor of Robin Milner*, chapter The Foundations of Esterel. MIT Press, 2000.

[8] L. Blanc. *Répartition de systèmes temps réel à contrôle prédominant*. PhD thesis, Université de Nice-Sophia Antipolis, October 1999.

[9] F. Boniol. *Etude d'une sémantique de la réactivité : variations autour du modèle synchrone et application aux systèmes embarqués*. PhD thesis, ONERA-CERT (Toulouse), December 1997.

[10] F. Boulanger. *Intégration de modules synchrone dans la programmation par objets*. PhD thesis, Supelec, université de Paris Sud, December 1993.

[11] F. Boussinot, G. Doumenc, and J.-B. Stefani. Reactive Objects. *Ann. Telecommunication*, 51(9–10):459–473, 1996.

[12] P. Caspi and R. Salem. The Quasi-Synchronous Approach to Distributed Control Systems. In F.Cassez, C.Jard, B.Rozoy, and M.Ryan, editors, *Proceedings of the Summer School "Modelling and Verification of Parallel Processes" (MOVEP'2k)*, pages 253–262, 2000.

[13] W. Damm and D. Harel. LSCs: Breathing life into Message Sequence Charts. In *3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, pages 293–312. Kluwer Academic Publisher, 1999.

[14] B. P. Douglass. *Doing Hard Time: Developping Real Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.
[15] Esterel Technologies. http://www.esterel-technologies.com.
[16] S. Gérard. *Modélisation UML exécutable pour les systèmes embarqués de l'automobile*. PhD thesis, Université d'Évry, October 2001.
[17] N. Halbwachs. *Synchronous Programming of Real Time Systems*. Kluwer Academic Publisher, 1993.
[18] D. Harel. StateCharts: a Visual Formalism for Complex Systems. *Science of Computer programming*, 8, 1987.
[19] L. Kabous and W. Nebel. Modeling Hard Real-Time Systems with UML: the OOHARTS Approach. In *Proceedings «UML»'99*, number 1723 in LNCS, Fort Collins, CO, USA, October 1999. Springer Verlag.
[20] Y. le Biannic, E. Nassor, E. Ledinot, and S. Dissoubray. Spécifications objets UML de logiciels temps réel. In *RTS*, March 2000.
[21] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28, December 1996.
[22] B. Selic, G. Gullerkson, and P. T. Ward. *Real-time Object-Oriented modeling*. John Wiley and Sons, 1994.
[23] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real Time Systems. Technical report, ObjecTime, 1998. http://www.objectime.com.