Towards a “Synchronous Reactive” UML subprofile ?

Robert de Simone*
INRIA
Sophia-Antipolis

Charles André
I3S laboratory
University of Nice Sophia-Antipolis

August 31, 2003

Abstract

The domain of *Real-Time Embedded (RTE)* systems was ackowledged as being largely influencial on many feature additions to the upcoming UML2.0 standard [1]. Work on UML1.4 *Scheduling, Performance & Time(SPT)* profile also goes in that direction. Still, the generic paradigms underlying these modeling efforts is that of software components, running on a real-time OSs with physical time constraints and middleware (e.g., RT-Corba) concerns. In other areas of Embedded System Design other paradigms are at work, owing to codesign techniques at the border between software and hardware, or discrete time mathematical engineering (MATLAB/Simulink) and DSP algorithms, etc. The paradigm of *Synchronous Reactive (S/R)* systems [2, 3], with discrete logical time and behavior decomposition into instantaneous reactions, proved quite natural in such areas to model mixed HW/SW *System-Level Design (SLD)*. We describe here some of the modeling paradigms needed for a true S/R model framework, and corresponding diagrammatic interpretations.

1 Introduction

The field of electronic Real-Time Embedded systems is currently gaining even more attention from researchers, due to the emergence of numerous applications pervading to the largest audience, as shown in modern cars, cellular phones, and other handheld appliances. The design of such RTE systems borrows from a variety of distinct engineering activities, and includes different aspects of modeling, simulation and prototyping, code generation and synthesis or reprogramming or component reuse, early user requirement capture and later heavy testing. RTE systems are even becoming a domain of choice for the definition of formal methods and verification, owing to the fact that *a posteriori* debug is often unfeasible once systems are delivered to customers.

The fact that RTE systems are intrinsically heterogeneous, and that their design owes to modeling and programming activities pertaining to many different enginering fields (microelectronics, applied mathematics and control theory, computer science and networking, and even mechanics) make them a challenging application domain to the UML representation techniques and associated design methodologies. This influence should be apparent as part of the forthcoming UML2.0 standard update, or the “*Schedulability, Performance & Time*” (SPT) [4] profile definition. Still, in both cases the historical background of middleware software engineering can strongly be felt, with paradigms drawn straight from the world of software components and asynchronous communicating agents or Real-Time OS underlying many modeling notions.

Another aspect worth mentioning here is that the modeling of *physical* timing considerations as introduced in the SPT profile primarily aimed at allowing performance analysis, and relies on a descriptive style of behaviors, rather than a generative/programming style. In other words, real-time analysis considerations are applied to individual execution scenarios (which may still exhibit partial independence between concurrent activities, but no alternative choice as in a full program). One main aim of synchronous reactive formalisms is to provide, inside their range of application, for a true programming formalism based on *logical* time paradigm, in which multiple executions can be generated according to various contextual inputs and *logical* instants. Timing analysis can be reintroduced at later stages [5]. The relevance of synchronous formalisms as specification languages leading to real code through synthesis methods can be seen through the ESTEREL STUDIO and SCADE SUITE products marketed by ESTEREL TECHNOLOGIES[6]. They use SYNCCHARTS [7], a synchronous state-based model.

* Research partly supported by the Project ITEA “Prompt2Implementation”.

Benefits of the Synchronous Reactive paradigm are numerous (when dealing with classes of applications for which they make sense, of course). First, the logical division of time between discrete instants allows proper mathematical models and operational semantics to be defined, while confining most subsequent analysis *inside* a given reaction. In particular clear notions of *simultaneity* and *absence* can be defined. Second, inside a given atomic instant, the partial ordering of activities is nonambiguously defined from high-level abstract criteria (a value must be computed or received for that precise instant before it can be used), leading through confluence properties to deterministic and predictable behaviors. The logical and functional correctness of the application at specification level does not result from mere physical time chronology but relies on clear causality relations instead. This should not be confused with the fact that timing and scheduling issues of activities could not be tackled any longer. Indeed, these issues are even favored at later stages because of the established boundedness of individual reactions, and the fact that inside a reaction a sufficient partial order between activities is already achieved. As a matter of fact a huge number of applications allow a *uniform* partial order to be defined, which may be projected down onto active parts at each instant (here "uniform" means: valid uniformly for all reactions). Then full static scheduling, accurate WCET estimations, and other similar analyses can be performed at compile time on program models. Also, the timing aspects of individual functions need not be fixed or even known (while those timing figures, provided at specification time, are often utterly suspect anyway). They can be attached later on, as an additional information that can be modified and updated, leading to different scheduling results. The *platform-based design* approach, as advocated for instance in the AAA methodology and the SynDEx tool [8], even allows to attach different timing characteristics to the mappings of the same elementary function onto various potential physical implementation resources. Then sheduling and placement decisions (composing the global *mapping*) are algorithmically synthesized (together with necessary communications), in a compilation-like phase.

To be fair, there appears to be still some drawbacks to the synchronous approach, which have mostly to do with the rigidity of the time division, and the fact that the designer must apparently be well aware of this cycle division so as not to miss important signal notifications (hardware people would talk of "cycle-accurate" models, and would then face the issues of "retiming"). Still, this is not such a real problem, in our views, since the underlying mathematical models do allow precise analysis of the related phenomena, and proper programming primitives (such as `await S` in Esterel) allow to consider easily the next instant when a signal/event occurs, whatever the number of logical instants elapsed before.

The paper is organised as follows: we first introduce informally on a running *Mutual Exclusion Arbiter* example some of the issues dealing with the relative lack of expressive power in the current UML models. We discuss the merits of a *synchronous* solution, which uses in fact synchronous version of state and activity diagrams, and we contrast it against UML standpoints on those issues (mostly logical discrete time division, with well-defined notions of simultaneity and absence, leading to clean priority and preemption modeling features offered to the user in readable syntax). We then recall in more details the foundational assumptions of *synchronous reactive modeling*, before presenting a UML model of objects in the semantic domain and relationships between them. We discuss briefly the connections between the two main description styles in synchronous reactive programming (*imperative* and *declarative*) with, respectively, *state* and *activity* diagrams, focusing on precise adaptations to the synchronous approach. We close, with more general comments linking (to our views) weak points in the current SPT profile to potential "synchronous" solution. Of course it should be understood that is not to be taken as an overall criticism of the actual profile in its state, but as a statement that other classes of problems might better benefit from the synchronous assumption to enjoy a more natural modeling, at the right level of behavioral abstraction demanded by the application.

2 Example: A Mutual Exclusion Arbiter

The purpose of this example is to illustrate natural benefits of the *synchronous modeling* while pointing out limitations of the current UML standard. These issues are expressivity and proper level of abstraction in description.

2.1 Informal system specification

N users compete with each other for the exclusive access to a critical resource. They send requests to an arbiter that grants or denies access to the resource. Each user has a different static priority. The resource is

always granted to the requesting user with the highest priority, possibly blocking a user that has previously got the resource.

The Users

Each user can work in a stand-alone (non-critical) mode, called the autonomous mode. In this mode the user may, for instance, collect information and process data. At times, a user needs the critical resource (i.e., a bus). He/she explicitely requests the resource by sending a request signal (Rq).

The user then waits for an answer from the Arbiter. This answer can be either a grant signal (G) or a deny signal (D) valued by the number of candidates for the resource with higher priority. When the resource is granted, the user enters a new mode (usingRsc). When the access is denied, according to the value conveyed by D, the user may choose either to stay in the autonomous mode by sending back a release signal (Rl) or to enter the usingRsc mode.

Once the user has entered the usingRsc mode, it has effective access to the resource only when G is present. Conversely, the presence of D suspends its activity until the resource is granted back. When leaving this mode, the user sends a release signal (Rl) to the arbiter.

The Arbiter

The Arbiter must ensure that the resource is granted to *at most one* user, and that this user has priority over the other candidates. It must also guarantee that the resource is *immediately* (not eventually) granted to a user when at least one user is requesting.

Additional requirement

The description of the control must be linear in the number of users (N). So, we adopt a modular specification of the arbiter, conceived as an array of ArbUnit (arbitration units) arranged according to a decreasing priority order.

In what follows, we describe a possible control for the arbitration. For simplicity, we consider that signal D is pure, that is it does not convey any value. A complete version of the solution is available in a technical report [9].

2.2 Limitations of UML State Machines

The State Machine package of the UML specifies a set of concepts used for modeling discrete behavior through finite state-transition systems. The Arbiter seems to be relevant to this approach. Nevertheless, this will not work if we strictly comply with UML assumptions. To illustrate this point, we will refer to the UML 1.4 specifications [10] (the recent UML 1.5 did not change state machine specifications). The forthcoming UML 2.0 will introduce only slight changes in the specifications of the UML State machines. The quotations below, written in italic font, are excerpts from the UML 1.4 specifications.

UML State Machines are an object-based variant of Harel statecharts [11]. The semantics of the UML state Machine "*are described in terms of the operations of a hypothetical machine that implements a state machine specification*". Events trigger changes of states of the State Machine. "*An event is a specification of a type of observable occurrence. The occurrence that generates an event instance is assumed to take place at an instant in time with no duration*". In our modeling, we use only *signals*, a special kind of events. Events, or more exactly event instances, are received in an *event queue* of the hypothetical machine. An *event dispatcher mechanism* selects and dequeues event instances from the event queue for processing. There is place for semantic variations, for instance in choosing an order for event dequeuing. However, some constraints are more stringent:

- *Events are dispatched and processed by the state machine, one at a time.*
- *The semantics of event processing is based on the run-to-completion assumption, interpreted as run-to-completion processing.*

Dispatching events *one at a time* is a sensible hypothesis for most object-based software, but this forbids actually dealing with *simultaneous occurrences*, and thus rules out large classes of behaviors, such as the ones encountered in control-dominated systems. The Arbiter, a control-dominated system, should consider *all* the incoming requests *at once*.

The run-to-completion assumption imposes that no other event is dequeued before the processing of the previous event is fully completed. This makes it impossible for the Arbiter to determine the exact number of users currently competing for the resource (some requests may be pending in the queue).

"*The run-to-completion assumption simplifies the transition function of the state machine, since concurrency conflicts are avoided during the processing of event, allowing the state machine to safely complete its run-to-completion step*". This rule is not easy to apply when concurrent evolutions are required. The run-to-completion rule often leads to a serialization of the concurrent evolutions: each "concurrent" evolution is itself processed in a run-to-completion step, one at a time. UML does not forbid to apply run-to-completion steps orthogonally to the orthogonal regions of a composite state, but it rather advises against doing so. A comment in the UML 2.0 is revealing in that respect: "*such semantics are quite subtle and difficult to implement*". Unfortunately such semantics can be useful for highly reactive systems.

Insofar as simultaneous event occurrences are to be considered, the absence of some occurrence may be significant as well. *Reacting to the absence* of an event instance is definitely beyond the capability of the UML State Machine. In our modular design of the Arbiter, some transitions are explicitely triggered by the absence of a given signal.

To summarize, the UML State Machines have semantic limitations that make them inadequate to express highly reactive system behavior. The first hypothesis to relax is the processing of one event at a time: simultaneous event occurrences must be considered. To deal with simultaneous events, we need a clear notion of *instant*. With this notion of instant, reaction to the absence of an event occurrence can be soundly defined. Finally, concurrency should be treated as a first class concept, and the semantics of concurrent evolutions not reduced to a form of serialized evolutions. The synchronous paradigm, presented in Sec. 3 fulfils these requirements in a sound and effective way.

2.3 A synchronous solution

In this subsection, we propose a synchronous state machine description of the behavior of an arbitration unit. The model in use is SyncCharts, akin to UML State Machines and the Harel Statecharts, but the semantics of which is *strictly synchronous*. SyncCharts were introduced in 1996 [7]. A comprehensive presentation of their behavior is proposed in a tutorial [12] that describes the semantics of SSM (Safe State Machine)[1].

SyncCharts

The syncChart shown in Fig.1 specifies the behavior of the arbitration unit (ArbUnit). Besides signals Rq, Rl, G, and D that are used for communications between a user controller and the associated arbitration unit, new signals have been introduced for synchronization. Signals Fi (Free input) and Fo (Free output) are used to propagate the resource availability. The idea is to forward the Fi signal like a token, from an arbitration unit to the next arbitration unit, the units being arranged according to a decreasing priority order. The arbitration unit that grants the resource, absorbs this signal (i.e., does not forward it).

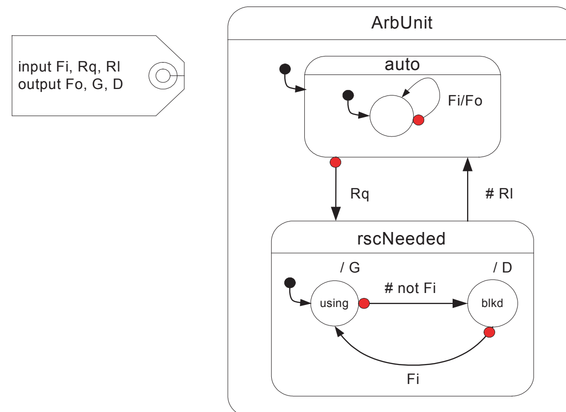

Figure 1: Behavior of an arbitration unit.

[1]SSM is the name given to SyncCharts in Esterel Studio and SCADE, products from Esterel Technologies.

The Arbiter is made of N arbitration units. Its behavior is defined by the parallel composition of N instances of the syncChart ArbUnit, and the introduction of local signals. Fig.2 is a structure diagram for the full controller (arbitrations units and user controllers) when N=3.

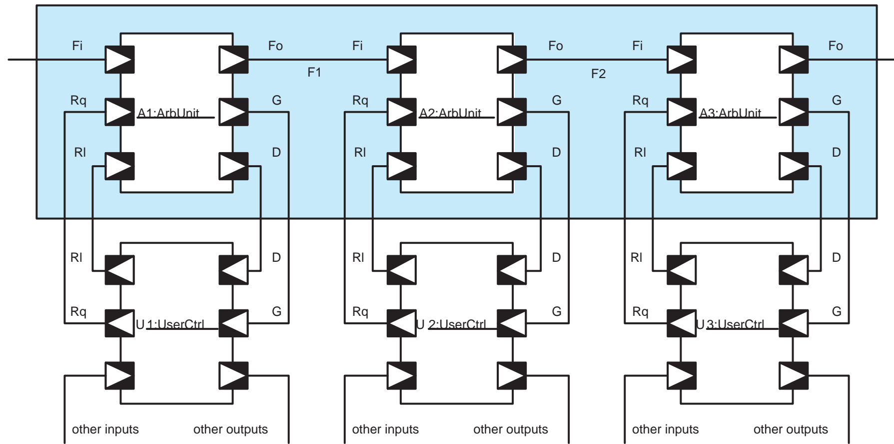

Figure 2: Structure diagram for the controller.

Activities

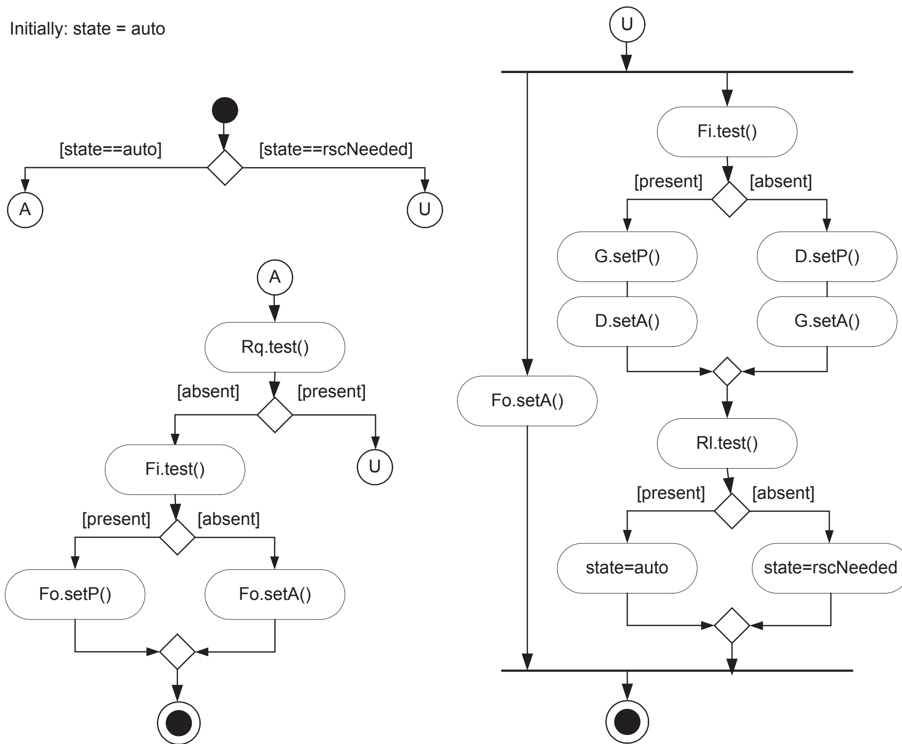

Figure 3: Activity diagram for the ArbUnit.

A syncChart is an imperative description of the expected behavior in terms of events (signals) and changes of states. This representation is concise and precise. An alternative representation of the behavior may be activity-oriented, instead of state-oriented. The *Activity Diagram* shown in Fig 3 represents the behavior of an arbitration unit. Note that the arbitration unit behavior is especially simple: each signal can be emitted at one place only. This feature greatly facilitates the translation from the state-based representation

to the activity-based representation The general case is much more complex, analogous to a distributed decision procedure.

For the arbitration unit there are only simple actions:

- test that tests whether a signal is present or absent,
- setP that forces the status of a signal to present,
- setA that forces the status of a signal to absent.

This description is at lower specification level than the corresponding syncChart. States are now implicit and replaced by variables (the state variable in the ArbUnit example). The activity diagram of the arbitration unit shows what this unit may potentially do during an instant and in which order. The InitialNode (solid circle) and the ActivityFinalNode (bull's-eye) clearly mark the begin and the end of the activity during one instant. This diagram also reflects the partial-ordering of activities.

Note that the translation from the control-flow model (the syncChart) into the data-flow model (the activity diagram) is usually done by compilers. We will come back to this double representation of synchronous evolutions in Sec. 3.3.

Reaction

Both above models (the syncChart and the activity diagram) represent the behavior of only a part of the system. These models have to be composed to represent the *emergent behavior* resulting from the cooperation of all the parts of the system. For simplicity we choose a small number of users ($N = 3$) and we illustrate the behavior for a particular evolution.

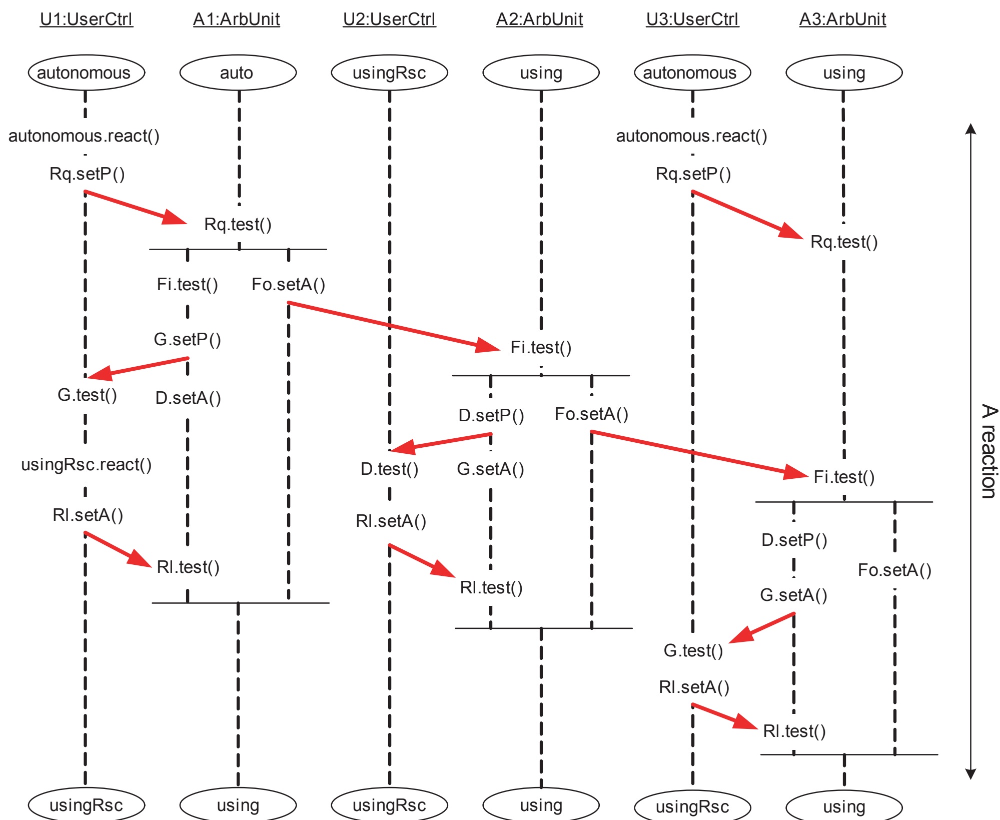

Figure 4: Partial order associated with a reaction.

Assume that User1 and User3 are in the autonomous mode, and User2 is in the usingRsc mode. The system is then in the *stable configuration* shown in the upper part of the Fig. 4. For this informal presentation, a configuration can be interpreted as the set of active concurrent states. What is the behavior of the system when both User1 and User3 simultaneously request the resource? The answer is given in Fig. 4 as

a partial ordering of actions. The vertical dashed lines express the partial order deduced from the activity diagram associated with each part (Fig. 3 for the arbitration units, and an activity diagram not given in this paper for the user controllers). Oblique arrows stand for additional ordering constraints imposed by a precedence relationship: any test of a signal test must be preceded by an action fixing its status (present or absent) at the current instant. This is a natural and sound rule imposed to all signals in the synchronous approach. Since each vertical line has a finite extension, this diagram is also finite, and it characterizes the *reaction* of the reactive system at the current instant. The bottom of the figure indicates the reached stable configuration.

A reaction in the synchronous approach can thus be assimilated to a complex activity that expresses the emergent behavior of the system at a given instant. Contrary to usual (asynchronous) interactions among objects, a reaction has always a well-defined extension from a stable configuration to another one.

Of course, Fig. 4 describes only one possible evolution. Compilers of synchronous models compute (symbolically) all possible reactions from a given stable configuration. They can also derive a partial order compatible with all the partial orders underlying each possible reaction. From this partial order a static scheduling for all the actions is then proposed.

Note that cyclic dependencies may preclude the construction of any valid reaction. These cases are detected by the compiler that rejects the model as a non constructive one.

3 The Synchronous Reactive Paradigm

3.1 Generalities

One calls "*reactive*" those systems that are primarily meant to react in time to input stimuli of some nature (in our case *signals*) by sending out some computed output information in return. Reactive systems are called *synchronous* when such a reaction cycle takes place inside a given logical instant, which is shared by all components of the system. So the main characteristics of *reactive synchronous* (S/R) systems is the assumption of discrete time division into *instants*. Actually an instant can be seen as an interval between two *ticks* of a (discrete) global clock, real or virtual. All components will complete their behavior for the current instant (or "stabilize") *before* the next global instant starts (at the next clock tick). The behavior of a single reaction is embodied in the following three steps:

1. Acquire input values (including signal presence status)

2. Compute internally from these values and the current state

3. Produce output values and update state for the next reaction

Each reaction is prompted by an initiating clock tick, which should be mentioned as "step 0" to allow proper chaining of reactions.

Important Notice: this definition, where the word "synchronous" applies to *components* behaving so-to-speak with identical speeds, should not be confused with another acceptation of the word "synchronous", more traditional to UML prose, where it is used to qualify *types of communications*. In a synchronous communication the calling process awaits upon a returned result to its invocation message before resuming its own activity, as in a remote procedure call. **We shall never use this second meaning in the current paper**. In synchronous reactive models signal exchanges are supposed to be instantaneous broadcasts anyway, occuring entirely *inside* a single reaction, *without* logical delays between emissions and (thus simultaneous) receptions.

The reader is defered to [2, 13] for a more philosophical debate about synchronous reactive assumptions. We want to assert here that it is the actual logical time model that underlies many formal design models, most important being the RTL level models in hardware circuit description languages (*netlists, logical gate schematics, Mealy machines*) or in discrete-time scientific engineering (block/diagram networks of operators). On the other hand, the description languages used to specify these systems for CAD simulation purposes often fail to fully recognize the importance of this assumption, as in for instance VHDL and Verilog, or Simulink simulation schemes. Time-driven reactive simulation is then replaced by event-driven simulation, under the influencial belief that the latter might lead to more efficient simulation speed. But as a result the semantics gets often unclear, informal and ambiguous, if not nondeterministic. Evidences that S/R semantics and event-driven efficient simulation are generally not contradictory were brought in [14, 15, 16].

The crux of the S/R paradigm is thus to be able to consider, through the higher atomic level of instants, *sets of* simultaneous input stimuli, which trigger internal and ultimately ouput behaviors, with a simultaneous change of state in a number of parallel components. All these computations take place in a single instant (time atomicity), but through a collection of causal behavior steps including possibly local signaling. In contrast to traditional UML approach an *event* shall thus now consist of *a set of occurring signals* (with possibly values), and the "run-to-completion" type of semantics will require the simultaneous consistent evolution of all components in the system, at the same clock pace, while propagating signal activities until a global stable state is found (and the instant is terminated, before the clock produces its next tick).

Importantly, the Synchronous Reactive Hypothesis allows to deal consistently and deterministically with the issue of *reaction to absence*. In asynchronous systems, where components may evolve at different paces or where signal arrivals may be delayed, presence or absence of a signal at a given reaction may vary non-deterministically. This is not so in S/R systems, where the absence/presence status of all signals can be deterministically determined in each reaction (at each instant). Actually, deciding on absence may yet involve insighful algorithms, akin to distributed knowledge perception: a signal can only be declared absent when the previous behavior decisions performed so far in the course of the reaction ban all potential emissions on that signal for the instant being.

To resume: In S/R systems, *events* are sets of signals, complex behaviors can be composed "inside an instant", but in a way related to *abstract causality* rather than *concrete timeliness*.
The ability of dealing with simultaneous behaviors and events, partially ordered in a well-founded way is a strength of synchronous reactive formalisms. The precise underlying semantics allows all sorts of model analyses such as constructive causality and model checking, as well as optimization techniques based on static analysis methods. The previously mentioned models (Mealy machines and netlists) provide mathematical interpretation to formally justify such transformations.

3.2 A UML conceptual model for S/R systems

We shall try to sketch a UML *domain model* of S/R systems matching the previous concepts, through Figs. 5 and 6. (Warning: this is "work-in-progress", to be further elaborated on). In places we shall use words such as "model", "signal", or "unit" with our own specific meaning, possibly departing from their use in other domains. To avoid confusions we encapsulate our names with a "SR" prefix (short for "synchronous reactive").

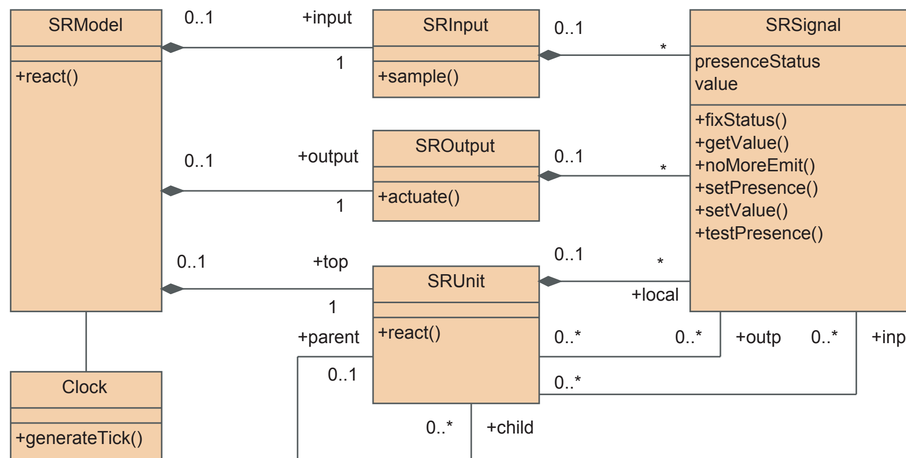

Figure 5: The synchronous reactive model: main concepts (1).

Fig.5 shows that a *S/R model* is linked to a *clock*. A S/R model consists of a tree hierarchy of *S/R units* and two sets of interface signals (named *input* and *output*). The input signals of the S/R model (input) are *sampled* from the environment at each reaction (sample operation of the SRInput class). The output signals of the S/R model (output) are *actuated* to the environment in the course of the reaction (actuate operation of the SROutput class).

The S/R unit that is root of the hierarchy is called the top S/R unit. Each S/R unit may own a set of S/R signals (local) and access to S/R unit's interface signals either as input or as output signal (inp and outp

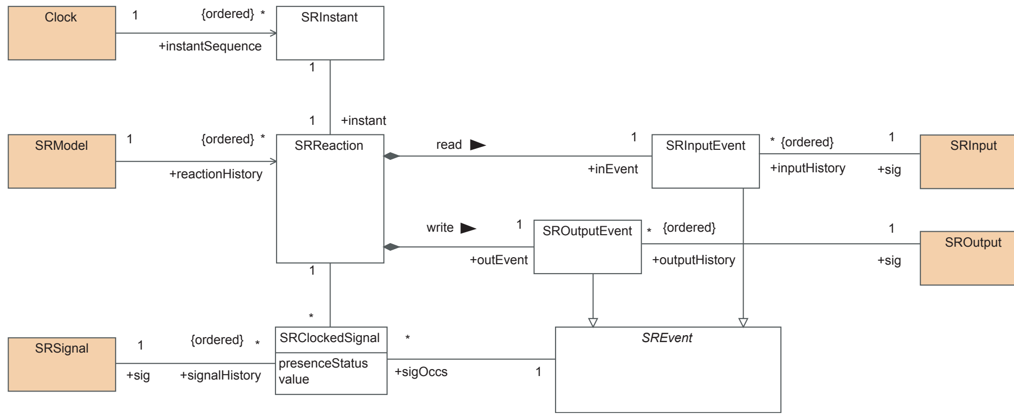

Figure 6: The synchronous reactive model: main concepts (2).

sets). The signal scoping must consistently follow the S/R unit hierarchy: for each S/R unit, except top, an interface signal (in inp or outp) must be either local to its parent or in the interface of its parent. For top, a signal in inp (outp, resp.) must be in the input (output, resp.) set of the S/R model. Access to the presence status (testPresence operation), and access to the value, if any (getValue operation) are allowed for any S/R signal visible to a S/R unit (i.e., local or interface signals). Setting the presence status of a visible signal (setPresence operation) or, when present, setting its value (setValue) is allowed only for local and output interface signals (local and outp). Given that a signal can be emitted several times at several locations in the same reaction, and that deciding upon signal absence requires to discover that no more such emissions are feasible, two extra operations are needed (fixStatus and noMoreEmit).

As already mentioned, the behavior of S/R systems is divided into a history sequence of reactions. So most of the objects will be endowed with a sequence of values, and behaviors can accurately be described by histories of *instantaneous snapshots*.

In Fig.6 classes with a clear background represent dynamic concepts relevant to our S/R semantics domain: the class SRInstant provides for a snapshot of the Clock, and similarly the classes SRClockedSignal and SRReaction provide behavioral snapshot versions of SRSignal and SRModel classes. Additionally at each instant the collection of SRClockedSignal instances (with their current presenceStatus and value) composes the current abstract SREvent snapshot, which is either a S/R input event (SRInputEvent class) or a S/R output event (SROutputEvent class). S/R event histories can in turn be projected to snapshots of individual S/R signal histories (signalHistory).

3.3 Behavioral models of description for S/R systems

The description of S/R systems calls for syntaxic constructs. Two prevalent (and complementary) styles currently exist: imperative syntax for control dominated components; declarative syntax for data-flow dominated parts.

The imperative family is represented by languages such as Esterel and SyncCharts[7]; it adheres to concepts originated in *Computer Science* language design community, generalizing traditional sequential flowgraphs with parallel constructs and signal exchange, and considering dynamic variable assignments for data handling. Global states are obtained as collections of active local states, themselves represented as particular *Program Counters* in the imperative code where evaluation need to rest until next instant. As seen in our previous example SyncCharts are very similar in structure to StateCharts or UML state machines, with the important difference that they adhere strictly to the synchronous hypothesis, making it possible to define formally clean notions of priority and preemption in case of simultaneous behaviors. Control flow in Esterel is intantaneous until an explicit `pause` statement is reached (or a derived language construct implying it), so that the time division is clear from syntax. Similarly in SyncCharts with explicit states with outgoing "immediate" transitions.

The declarative family is represented by languages such as Lustre/Scade and Signal[17, 18]. It adheres rather to concepts of *Control Theory* community, and describes computations as networks of operators (which also generalize sequential pipelines), much in the tradition of block diagrams and/or Kahn networks. Still, an important asumption is that here data flow is instantaneous and acyclic, so that basically only one

single reaction is described, with the intention that the program behavior is an infinitive iteration of this reaction. Outputs from fonctional blocks are produced at the same speed as they are consumed by further blocks, following the acyclic flow, so that the synchronous semantics here ensures that no values will be pending in buffers between blocks. The general case is slightly more complex as it introduces *clocks* which may refine the behavior in parts that are only active upon clock activation, but here a careful *clock calculus* ensures that the property remains that a value is produced only to be consumed at the same rate. The program has memory capability, in that one may refer to the value of a variable at the previous instant. Then the corresponding data values must be preserved in provision for the next instant, and these constitute the state. Here a state can thus be seen as a collection of data registers. Again, these assumptions (acyclic finite data flow graphs, and data register state elements to ensure consitency through successive reactions) correspond to the restrictions we were imposing on Activity diagrams in our previous example.

3.4 Merging the two styles of behavioral aspects

A complex system usually consists of both types of components, with the *imperative* part producing most control flow and modes to activate selectively some of the *declarative* parts, the latter performing intensive data flow computations. In a sense the imperative part will compute and provide the "clocks" which will trigger (or not) the actual data computation algorithmic parts. Here a very informal link to the combination of microcontrollers with DSP coprocessors in modern Systems on Chip could be illustrative.

The usual way of combining hierarchical FSMs with block/diagram data flow networks is in general to immerge individual state machines as further components of a general networks, with specific variables representing their current and next state being handled (tested or assigned) as part of the data flow computation. In practice the effect of the tests are to trigger the activity behaviors corresponding the actual current state, and inhibite others. So the FSMs can be seen as providing *activation conditions* of a *clock* nature. This we could call the "static" view, as it appears from the model that all block/diagrams could be concurrently activated, and the relations between clocks are not visible. Another, more dynamic way of assembling both styles would consist in "holding" subnetworks in various states, so that a block comes to be dynamically inserted as the current data computation when the state becomes active. This would be called the "dynamic" view, where control flow modes play the role of implicit activation clocks. This type of combination was advocated in mode-automata [19], and is analogous to other proposals outside the synchronous world [20, 21].

While there are informal statements in the UML litterature hinting that state and activity diagrams are essentially considered as fulfilling the same need of behavioral representation, with only distinct styles, there has never been any attempt at formally combining them. We shall not build further here on this issue, but live to the comment that behind the proper modeling styles there could be issues of efficient implementation or simulation, if ever models were to serve for code synthesis.

3.5 A lightweight comparison with the SPT profile concerns

We shall be very informal in this part. The purpose is mainly to distinguish and clarify matters between different modeling efforts, with very similar domain, terminology and concepts, but basically different aims. Needless to say, those aims can all be important and fully justified in their own rights.

As mentioned in the introduction, the overall aim of the SPT seems to be the performance and schedulability analysis of individual execution scenarios (possibly exhibiting concurrency). Scenarios as introduced in the SPT document do not actually refer to any existing UML model, and do not represent sets of alternative behaviors (as in a generative program). The synchronous reactive assumption, on the opposite, can be seen as a means to provide state diagrams (and to a certain extend a restrictive set of activity diagrams) with a clear and semantically sound way of generating valid execution sequences, thereby endowing then with a programming language quality.

As hinted in the section 2.3, there is a formal way to translate SyncCharts into the appropriate class of activity diagrams, with the target activity encoding the desired behavior as one of a single instant, where states are encoded through data variables, themselves tested at the start of the reactive activity to deduce really active behaviors, and reassigned at the end of the reaction to prepare for next instant. This is interesting because then this behavioral model, being acyclic and statically bounded in terms of activities, can be subject to performance analysis and schedulability techniques *"as a whole"*, while it represents a *collection* of behaviors as generated in a program. To be fully exact this requires an additional property (of uniform

acyclicity in the various communication schemes), which is nevertheless very often met in practice. On the other end the scheduling reported then is only based, and relevant, to the *intra* instant behaviors.

The last topic we wish to mention deals with the fact that the specification style promoted by the synchronous reactive approach is fully independent from any runtime system mechanism or execution platform. Thus, this model can be mapped afterwards in an optimized fashion onto any platform description that matches its level of formal description. The corresponding methodolgy then becomes: first describe the application as a synchronous "platform independent model", and independently provide a full-fledged platform model (including hardware and software components, communication media and so on); third, provide a cost for each elementary function on any possible resource. Then optimizing methods can compute potential solutions for the full mapping of the application onto the platform. This is now often called *platform-based design* in the codesign community, and is largely put at work in the AAA methodolgy and the SynDEx tool [8].

Interestingly the output format displaying the result of such a scheduling bears strong resemblance with the extended "timed" sequence diagrams introduced in SPT. Again this result is currently valid for one single reaction and instant.

4 Conclusion

In this paper we defended the case of synchronous/reactive formalisms for the modeling of certain classes of real-time embedded systems. We proposed preliminary temptative solutions to place our approach in the light of UML behavior diagrams and the SPT profile.

Between the "timeless" traditional UML modeling and the SPT profile, based on concrete physical time approach, there should be room for a modeling approach based on more abstract logical time and causality. The synchronous paradigm partially answers this need. This is obtained at the price of somehow strong assumptions, which are nevertheless often met, but in a range of applications certainly more familiar to control theoretists and hardware synthesis engineers than classical computer science ones.

When such assumptions are practically satisfied, it would be interesting to investigate further how the synchronous reactive modeling can indeed form a descriptive support on which efficient timing analysis can be globally performed and uplifted to full programs (allowing, for instance, accurate WCET computations). But one should not be misled by these conclusions in a very precise context: the purpose of the SPT timing modeling is to be able to analyze systems with much larger dynamical aspects on one hand, and the relevance of the scheduling aspects in synchronous systems are often limited to individual reactions. More work is certainly needed in the future to try and relax some of these limitations.

References

[1] M. Björkander. System Level Modeling and UML. In *FDL'03 Keynote address*, 2003.

[2] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, January 2003.

[4] OMG. *UML Profile for Schedulability, Performance, and Time Specification (draft specification)*. Object Management Group, Inc., 492 Old Connecticut Path, Framing-ham, MA 01701., April 2003. OMG document number: ptc/2003-03-02.

[5] D. Weil, E. Closse, M. Poize, P. Venier, J. Pulou, S. Yovine, and J. Sifakis. TAXYS : a Tool for Developing and Verifying Real-Time Properties of Embedded Systems. In *CAV'01*, LNCS 2102, 2001.

[6] `http://www.esterel-technologies.com`.

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

[8] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The syndex software environment for real-time distributed systems design and implementation. In *European Control Conference (ECC'91)*, 1991.

[9] C. André. Resource Allocation Management using SYNCCHARTS: a Case Study. Technical Report ISRN I3S/RR–2003–20–FR, I3S, Sophia-Antipolis, France, August 2003.

[10] OMG. *OMG Unified Modeling Language Specification*. Object Management Group, Inc., 492 Old Connecticut Path, Framing-ham, MA 01701., Sept 2001. OMG document number: 01-09-67.

[11] David Harel. StateCharts: a Visual Formalism for Complex Systems. *Science of Computer programming*, 8, 1987.

[12] C. André. *Semantics of SSM (Safe State Machine)*. Esterel Technologies, electronic version available at http://www.esterel-technologies.com, April 2003.

[13] G. Berry. *The Constructive Semantics of Pure Esterel*. electronic version only, 1999.

[14] Stephen Edwards. Compiling Esterel into sequential code. In *Proceedings CODES'99*, Rome, Italy, May 1999.

[15] Etienne Closse, Michel Poize, Jacques Pulou, Patrick Vernier, and Daniel Weil. Saxo-rt: Interpreting Esterel semantic on a sequential execution structure. *Electronic Notes in Theoretical Computer Science*, 65, 2002.

[16] D. Potop and R. de Simone. Optimizations for Faster Execution of Esterel Programs. In *MEMOCODE'03*, 2003.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[18] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[19] F. Maraninchi and Y. Rémond. Mode-Automata: a new domain-specific construct for the development of safe critical systems. 2002.

[20] J. Liu and E. A. Lee. A component-based approach to modeling and simulating mixed-signal and hybrid systems . *ACM TOMACS*, 2002.

[21] Radu Grosu, Thomas Stauner, and Manfred Broy. A modular visual model for hybrid systems. *Lecture Notes in Computer Science*, 1486:75–91, 1998.