Synchronous Reactive Programming in Ptolemy

Frédéric Boulanger
Frederic.Boulanger@supelec.fr
tel: (33 ˜ 1) 69 85 14 84
fax: (33 ˜ 1) 69 85 12 34

and

Guy Vidal-Naquet
Guy.Vidal-Naquet@supelec.fr
tel: (33 ˜ 1) 69 85 14 75
fax: (33 ˜ 1) 69 85 12 34

École Supérieure d'Électricité — Service Informatique
Plateau de Moulon
91192 Gif-sur-Yvette Cedex
France

and

Laboratoire de Recherche en Informatique
Université de Paris-Sud Orsay
Bâtiment 490
91405 Orsay
France

ABSTRACT

Synchronous reactive languages allow a high level deterministic description of reactive systems such as control-command systems. Their well defined mathematical semantics makes it possible to check formal properties on the control of a system.

In previous work, we developed an object-oriented execution model for synchronous reactive modules. This model is implemented as a set of tools and a C++ class library, and allows us to use object-oriented methodologies and tools for the design of complex applications with both transformational and reactive parts.

Among these design tools, the Ptolemy system stands as an object-oriented framework that supports various execution models, or "domains". We are currently working on a translator from the output format of the Lustre and Esterel compilers to the Ptolemy language. Since no existing domain matches the reactive synchronous execution model, we also plan to develop a SEC (Synchronous Execution and Communication) domain. Such a domain will provide support for the execution of synchronous modules in Ptolemy.

One of the most interesting features of Ptolemy is the communication between domains. Therefore we discuss the interface of the SEC domain to other domains to determine the meaning of communications between them. The main goal is to allow the use of synchronous reactive modules for the control of the behavior of data-flow or discrete event processes.

SYNCHRONOUS AND OBJECT-ORIENTED APPROACHES

A reactive system, as defined by Harel and Pnuelli (1985), is a system that interacts continuously with its environment, at the pace of this environment. This defines reactive systems as opposed to transformational systems (they get all their data at the beginning and stop when they have produced their result), and interactive systems (they interact with their environment, but at their own pace).

In the category of the reactive systems, we find most of the real-time systems (control/command, signal processing), but also user interface systems or network protocols.

These systems are generally deterministic, must obey time and reliability constraints, and are often described as several concurrent processes. The synchronous approach, which is based on the hypothesis that a system reacts instantaneously to its inputs by producing its outputs at the same time, makes it possible to describe reactive systems in high level languages that combine parallelism and determinism. A real system cannot react instantaneously, but provided that the reaction to an event is finished when the next event occurs, the system can be considered to follow this model.

Lustre (Halbwachs et al., 1991) and Signal (Benveniste and Le Guernic, 1990) are two data-flow synchronous reactive languages. With these languages, a system is described as a network of operators, each operator being itself either primitive or described by another network of operators. The operators communicate through signals. A signal is a series of values associated with a clock that specifies when the values are present. The network of operators is equivalent to a system of equations between the signals. The compiler solves this system of equations for the clock part of the signals, and yields a sequential deterministic ordering of elementary operations on the value of the signals. The result of the compilation can easily be translated into any sequential language.

Esterel (Berry and Gonthier, 1992) is an imperative parallel language in which processes communicate through signals by instantaneous broadcast. A signal has a value which is remnant, and it can be emitted or not at a given instant. Control takes no time in Esterel, and only three primitives — the emission of a signal, the test for the pres-

ence of a signal — and the execution of an instruction until the occurrence of signal, permit the expression of all behaviors. A fourth "exec" primitive has been added (Berry et al., 1992) to handle the interaction between synchronous and asynchronous processes. The exec primitive allows the execution of an asynchronous task to be integrated in the semantics of the other primitives of Esterel.

The mathematical semantics of the synchronous languages makes it possible to check formal properties on programs. For instance, if one writes a synchronous program to drive an elevator, it is possible to check that the elevator never moves with the door open. These proofs only deal with control properties (a signal is emitted or not), they do not concern properties of data values. The possibility to make formal proofs on the behavior of a system is very important for some applications where a design flaw may have catastrophic consequences.

The lack of support for the joint use of synchronous modules and transformational code in the same application led us to develop an object-oriented execution model for synchronous modules (Boulanger, 1993). Once compiled, a synchronous module is a black box whose behavior can only be observed through its input and output signals. This is very close to the notion of an object with encapsulated data, whose behavior can only be observed through the activation of its methods. The occ++ tool[1] uses this similarity to translate the intermediate code produced by the Lustre and Esterel compilers into a C++ class. The libSync class library implements the execution machine for synchronous objects and allows them to communicate synchronously. A set of interface classes implements the communication between synchronous objects and their environment, which includes transformational and interactive code or even hardware. For example, we developed an interface between libSync, the VxWorks kernel and data-acquisition VME cards.

SYNCHRONOUS REACTIVE EXECUTION MODEL

The synchronous reactive execution model of libSync and occ++ makes interconnected synchronous objects behave just like the corresponding synchronous modules in a synchronous language. At a given instant, all signals have a value and are emitted or not. Since the system is composed of several synchronous objects, we must make them react in turn, but in an order that preserves the synchronous semantics.

For example, figure 1 shows two interconnected modules A and B. The value of the α and β signals must be the same for both modules, so B must react after A because the value of β is computed by A. But the value of α must be preserved after the reaction of A so that B sees the same value of α as A.

[1]The software presented in this paper is available at ftp://ftp.supelec.fr/pub/cs/distrib/

Preserving the synchronous semantics obviously prohibits loops in the dependency graph because it would make one input of a module dependent on one of its outputs, and cause a signal to have several values in the same instant.

So, one limitation of our execution model is that instantaneous loops are forbidden. However, loops are allowed if they contain a connection (for instance through a delay) that does not imply an immediate dependency. With this limitation, finding an order of reaction for the synchronous modules consists in a topological sort of the dependency graph. Several orders may be possible, but will be strictly equivalent from the synchronous point of view. However, scheduling directives are available to influence the choice among the possible orders since it may have an influence on side effects of the modules (for instance, the order of activation is visible if some modules write on a file when they react because the reader is not synchronous and reads the file line after line, even if these lines were written at the same instant).

To sum up, in our synchronous execution model, instantaneous loops are not allowed, the order in which objects react is determined statically by a topological sort of the dependency graph, and each object reacts exactly once in an instant.

PTOLEMY

The Ptolemy system (Buck et al., 1991), developed at the University of California at Berkeley, is an object-oriented framework within which diverse models of computation (or domains) can coexist and interact. It supports heterogeneous system specification, simulation and design and is extensible: users can create new components in a domain or even create entirely new domains.

Since our goal is to combine the use of the synchronous paradigm with other models of computation, Ptolemy is a very good foundation upon which to build a synchronous reactive domain. First, this new domain will benefit from currently existing domains, and second, if an application needs to use synchronous modules with another paradigm, its designer will benefit from the Ptolemy kernel to make them interact.

Let us introduce some terminology: in Ptolemy, basic building blocks are called "stars". Stars communicate by exchanging data tokens named "particles" between their "portholes". Stars can be composed into new building blocks called "galaxies". A complete application is called a "Universe". "Wormholes" allow the use of building blocks of a domain in another domain. Particles exchanged between two domains go through "Event horizons".

Before studying the new synchronous domain, we must take a look at the existing domains to determine if a new domain is really necessary. The most promising domain to host synchronous reactive modules is the Discrete Events (DE) domain, but we will also discuss why the Synchro-

nous Data Flow (SDF) domain, although its name contains "Synchronous", is not appropriate for synchronous reactive modules.

THE SDF DOMAIN

In this domain, a star consumes a fixed number of particles on each of its inputs and produces a fixed number of particles on each of its outputs when it is fired. The firing order of the stars is determined statically before the run so that the production and the consumption of particles balance on each connection. It is possible to build a network of stars for which there is no possible scheduling, in which case the scheduler reports an error.

If we suppose that each synchronous star consumes exactly one particle on each of its inputs and produces exactly one particle on each of its outputs when it is fired, the scheduling of SDF matches the scheduling of our synchronous execution model.

The problem with SDF is the hypothesis that each synchronous star will consume or produce exactly one particle on each of its ports. A synchronous module produces events only when specified by its behavior, and consumes events only when they occur. So all we can say is that a synchronous star produces or consumes at most one particle on each of its ports when it is fired.

A simple solution to this issue is to store an attribute of presence in the value of the particles: a particle is produced at each firing, and the attribute of presence says whether it corresponds to a real event or not. But this forbids the use of all the built-in data types of Ptolemy and is not practical.

THE DE DOMAIN

In the DE domain, stars interact through dated events. The scheduler maintains a global event queue and fetches the event at the head of the queue, which is the earliest event. This event is sent to the input port of its destination star, as well as all other simultaneous events for input ports of the same star. The star is then fired, producing events on its output ports.

If several stars can be fired because they have simultaneous events on their inputs and may produce events with a zero delay, the order of their firings is determined according to a topological sort, so that a star which depends on the output of another star is fired after it. The effective scheduling algorithm is more subtle, but this will be enough to show what may go wrong for synchronous modules in DE.

Except for the dynamic scheduling, which is not necessary for synchronous modules as we saw in the description of the execution model, everything seems fine in DE to make synchronous stars behave correctly. We have developed a translator from the oc code produced by Esterel and Lustre to pl, the Ptolemy Language. This translator, ocpl, produces a DE star from a synchronous module. This star can be dynamically linked into Ptolemy so that we are able to experiment with it. Examples of use of synchronous DE stars are available too.

One example shows what differs between the DE semantics and the synchronous reactive semantics: the "Four Turtles". Four turtles are initially placed at the corners of a square and move on a grid. The behavior of each turtle is to take a step toward the next one if the distance between the two turtles is greater or equal to 2 grid units (a step is of length 1 or $\sqrt{2}$ grid units). "Taking a step" must be understood as choosing a new position and producing a particle that represents this position.

Since the interconnection of the turtles makes a loop, we must introduce a delay. We put a delay between every pair of turtles to conserve the symmetry of the system which is represented on figure 2.

The system evolves until the four turtles are grouped at the center of the square, as shown on figure 3.

There are two kinds of possible delays in DE: Delay stars and arc delays. The first kind is a star that produces a particle which is equal to the one it receives, but with a timestamp incremented by the value of the delay. Such delays make the time progress. The second kind of delay is a property of the connection: it just informs the scheduler that what seems to be a delay-free loop is perfectly safe.

The first issue we met in DE is that a star is fired only if it receives an event on any of its inputs. In the synchronous execution model, a star is fired each time there is a new instant, that is to say, each time the previous instant has been processed. To make the system start, we must make our turtles derive from the `RepeatStar` class so that they are initially scheduled. After this initial phase, the events they produce while moving keep the simulation alive until they meet at the center of the square.

In the arc delay version of the example, the initial firing of the four turtles occurs because they have been scheduled during the setup of the simulation. They emit their current position with timestamp zero, so they all have an event on their input. They are triggered again, consume the event on their input (which contains the position of their target), compute their new position and emit this new position with the same timestamp as the input event, which is zero. In this system, time does not progress and all the activations take place at instant zero. Since the display star does not produce events, it is never scheduled and particles accumulate on the arcs between it and the turtles. When the four turtles meet at the center of the square, they stop producing events since their position does not change any more. Then, the display star is the only one to have events on its input and it is fired until the particles that have accumulated are all consumed.

With this example, we can see that in DE, *all simultaneous events for a star may not be processed in the same firing*, and a star may be fired several times in the same instant, which breaks the main rule of our synchronous ex-

ecution model.

So, even if synchronous reactive modules may be used in DE, this domain does not provide a truly synchronous reactive execution model (but that was not the purpose of its designers).

THE SR DOMAIN

At the University of California at Berkeley, Stephen Edwards is working on a SR (Synchronous Reactive) domain. This domain is targeted toward the heterogeneous design of synchronous reactive systems. Whatever the original paradigm used to specify the building blocks, their behavior must be defined by a monotonous function. The behavior of the whole system at each instant is the least fixed point of the composition of the module functions. This fixed point is computed by convergent iteration.

In the SR domain, instantaneous loops are allowed because the behavior of the system is computed at each instant. If an instantaneous loop is not causal, this will only be detected at the instant when the causality fault appears. The counterpart of this lack of compile time causality checking is the small time needed to determine the behavior of the system, what makes it possible to compute it at each instant.

So SR can be seen as a synchronous reactive emulator, allowing the rapid prototyping of synchronous reactive systems from heterogeneous building blocks with great flexibility. SR systems may be used as stars in our Synchronous Execution and Communication domain.

THE NEW SEC DOMAIN

While SR deals with the heterogeneous design of synchronous systems, our project concerns the integration of already compiled synchronous modules into a heterogeneous design: we want to provide a synchronous reactive execution model to host synchronous modules and allow them to communicate with other domains. We do not require the behavior of the modules to be expressed by monotonous functions: the reaction of a module is atomic, producing all the output events at once.

The building blocks in our SEC (Synchronous Execution and Communication) domain are already compiled, and we do not deal with causality checking. These blocks are either obtained through ocpl from a description in a synchronous language (more exactly, it is the state machine in the oc format which is translated, so only Esterel and Lustre are currently usable with ocpl), or written from scratch in C++. They are considered as black boxes so, unless otherwise specified, all their outputs are supposed to depend on all their inputs.

The "unless otherwise specified" means that it is possible to make the box "less black" by saying that such output does dot depend instantaneously on the inputs, or that such input does not influence instantaneously the outputs. This makes it possible to specify and use delays to break instantaneous loops since we can express the fact that the output of a delay does not depend on its input instantaneously.

The most important problem to solve is the meaning of the communications between SEC and other domains. We have already considered the problem of the communication between synchronous and asynchronous worlds in the libSync library. When asynchronous events arrive in a synchronous system, they must be "synchronized", i.e. put on a particular instant of the synchronous system. The simplest way is to put an asynchronous event on the next occurring instant of the system, but this may not always be what we want. And what should we do when two or more asynchronous events occur between two instants of the synchronous system? Should the earlier event be discarded or should we queue the events to absorb a temporary burst?

This choice is very dependent on the desired behavior of the synchronous system, and it must be done by the designer since it is related to the application domain. In libSync, interface classes are in charge of the synchronization of asynchronous events, and the designer of a system chooses the appropriate synchronization policy for asynchronous events. He may write a new interface class to implement a particular policy. We think that the same solution applies for the SEC domain: their will be interface stars to communicate with other domains, just as there are interface stars to communicate with the user or the operating system in other domains.

Basic interface stars may be provided with SEC to implement the most common strategies of synchronization (event queuing, discarding all but the latest event) as well as the enforcement of the implications and exclusions that can be specified in a synchronous language. Implications and exclusions are assumptions on the behavior of the asynchronous world that allow the compiler to optimize the object code. An implication states that the presence of a signal implies the presence of another signal. An exclusion states that a signal is never present when another is. Since this information is used by the compiler, it is very important to make sure that these rules will be respected at run-time, otherwise the system will have to react to an event that is considered impossible.

We have only dealt with the issue of the inputs to a synchronous system. There is no difficulty for the outputs since a synchronous star produces at most one event on each of its outputs: all we have to do is to make sure that a particle is produced on each output when we communicate with a domain such as SDF. This can be done with a "Sampler" star, like in DE.

HOF: HIGHER ORDER FUNCTIONS

Once a system is described as a graph of black boxes, transforming its graph transforms the system. We have

developed the bdl (Block Description) language to define generic transformations of a graph and to apply them to the graph of an application. Such transformations are useful to handle multiple configurations of the same system. For example, we used transformations to add safety features to an application, allowing separate development of its operational and safety parts. In this context, a transformation may consist in replacing a module with N copies of this module and to add a voter on the outputs. Even if such a transformation seems simple, combinations of transformations may involve such a large number of connections that it would be error prone or even impossible to do them by hand.

Using transformations facilitates the maintenance and the management of several configurations because it loosens the coupling between the application and its operating environment: from the same initial graph, several transformations will produce the different configurations or versions.

The HOF domain in Ptolemy allows the design of stars that apply other stars to their inputs. The behavior of these stars is to modify the interconnection graph of the system and to disappear before the simulation or the code generation begins. HOF stars even allow the use of conditions and recursion in the design of a system without runtime overhead since these stars do not appear in the simulation or in the generated code.

So, by making HOF a subdomain of SEC, we will be able to implement a graphical version of bdl, and this will allow the design of complex systems that would be painful to build step by step.

CONCLUSION

Our first experiments with ocpl reveal the need for a new domain that will provide a suitable execution semantics to the modules produced by synchronous languages compilers. This new domain will come with interface stars that implement various communication strategies with the other domains, and will allow the use of Higher Order Functions stars to build complex systems or handle several configurations of the same system.

These experiments also show that using synchronous reactive modules in Ptolemy is promising for the design of heterogeneous systems that mix data-flow signal processing and event driven behaviors. This comes from the possibility to rapidly go back and forth between edition and simulation, and from the richness of the existing domains.

REFERENCES

Harel, D. and Pnuelli, A., 1985, "On the Development of Reactive Systems", Weizmann Institute of Science, Rehovot, Israel.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D., 1991, "The Synchronous Data Flow Programming Language LUSTRE", *Proceedings of the IEEE*, 79(9), September 1991.

Benveniste, A., and Le Guernic, P., 1990, "Hybrid Dynamical Systems Theory and the SIGNAL Language", *IEEE Transactions on Automatic Control*, 35(5), May 1990.

Berry, G., and Gonthier, G., 1992, "The ESTEREL synchronous programming language: Design, semantics, implementation", *Science of Computer Programming*, 19(2):87-152, November 1992.

Berry, G., Ramesh, S., and Shyamasundar, R.K., 1993, "Communicating Reactive Processes", *Proceedings of the 20th ACM Conference on Principles of Programming Languages*, Charleston, Virginia.

Boulanger, F., 1993, "Intégration de Modules Synchrones dans la Programmation par Objets", PhD thesis 2977, University of Paris-XI–Orsay.

Buck, J., Ha, S., Lee, E.A., and Messerschmitt, D.G., 1991, "Ptolemy: A mixed-paradigm simulation/prototyping platform in C++", in *Proceedings of the C++ At Work Conference*, Santa Clara, CA.

FIGURES

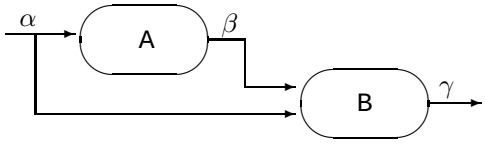

Fig. 1: Synchronous communication

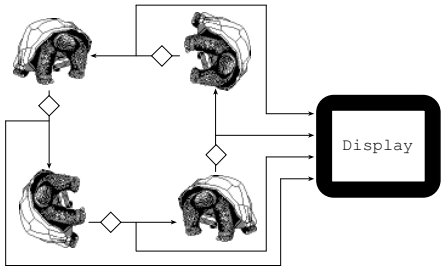

Fig. 2: The "Four turtles" example

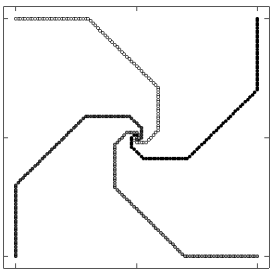

Fig. 3: Evolution of the "Four turtles" example