IMPLEMENTATION OF CONTINUOUS-TIME DYNAMICS IN SCICOS

Masoud Najafi Azzedine Azil Ramine Nikoukhah

INRIA, Rocquencourt,
BP 105, 78153 Le Chesnay cedex, France

KEYWORDS: Hybrid systems, Synchronous language, DAE, Simulation software, Numerical solver.

Abstract

Scicos is a software environment for modeling and simulation of dynamical systems. The underlying formalism in Scicos allows for modeling very general dynamical systems: systems including continuous, discrete and event based behaviors. This paper presents some aspects of the simulation software, especially the features which have to do with the hybrid nature of the models. We study in particular the implementation issues concerning the efficient use of ODE (Ordinary Differential Equations) and DAE (Differential/Algebraic Equations) solvers in the simulator.

INTRODUCTION

Scicos is a toolbox of the scientific software package scilab [Bunks et al. 1999, Chancelier et al. 2002]. Both Scilab and Scicos are free open-source softwares (`www.scilab.org`, `www.scicos.org`). Scicos includes a graphical editor for constructing models by interconnecting blocks, representing predefined or user defined functions, a compiler, a simulator, and some code generation facilities.

Scicos Formalism

The formalism used in Scicos is based on the formalism of synchronous languages, in particular *Signal* and its extension to continuous-time systems [Benveniste 1998]. We do not give a full presentation of the formalism in this paper (for more see [Nikoukhah and Steer 1996-a, Nikoukhah and Steer 1997, Nikoukhah et al,1999]). We simply review some aspects which specifically have to do with the continuous-time behavior to lay the ground for presenting the specific issues related to continuous-time simulation.

Even though there exists an inheritance mechanism in Scicos formalism which provides a data-flow type of behavior, Scicos is fundamentally event-triggered. This has made the continuous-time extension non-trivial. The basic idea consisted in treating the continuous-time events just like an ordinary event [Nikoukhah and Steer 2000].

Events are signals which activate system components in discrete event-driven environments. We extend the notion of event to obtain what we call an *activation signal* which consists of a union of isolated points in time and time intervals [Nikoukhah and Steer 1996-a, Djenidi et al. 2001]. Each Scicos signal is then associated with an activation signal specifying time instances at which the signal can evolve, see Fig.1.

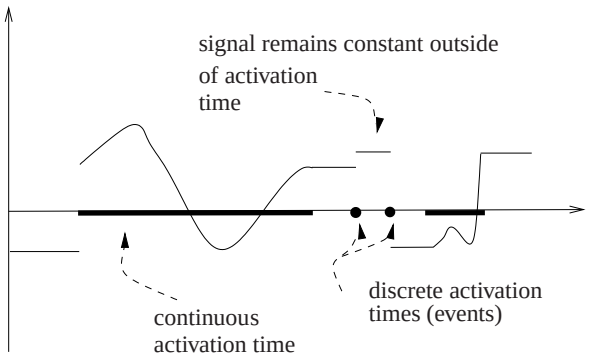

Figure 1: A typical Scicos signal. Thick line segments represent the activation times.

The fundamental assumption is that over an activation interval, in the absence of events, the signal is *smooth*. Scicos compiler propagates the activation signals through the model in order to obtain activation information about all the signals present in the system. This information is valuable for the simulator which has to properly parameterize and call the numerical solver.

It would be unrealistic to imagine that a formalism can be developed independent of the properties of the numerical solver. That is why it is important to study these properties and identify precisely what properties are important and must be taken into account in the development of the formalism and the implementation of the modeler and simulator.

Proceedings 15th European Simulation Symposium
Alexander Verbraeck, Vlatka Hlupic (Eds.)

ISBN 3-936150-28-1 (book) / 3-936150-29-X (CD)

Solver properties

Scicos uses two numerical solvers: `Lsodar` [Hindmarsh 1980, Petzold 1983-a] and `Daskr` [Petzold 1983-b, Brown et al. 1998]. `Lsodar` is an ODE solver which is used when the Scicos diagram does not contain *implicit* blocks. Implicit blocks are blocks which contain implicit dynamics of the form

$$\begin{aligned} 0 &= f(\dot{x}, x, t, u) \\ y &= g(\dot{x}, x, t, u) \end{aligned}$$

where u, x, y, and t represent respectively block's input, state, output, and the time; $\dot{x}$ represents the time derivative of the state x. Note that the implicit nature of the block has to do with the absence of an explicit expression for $\dot{x}$, the input and output are still explicitly defined. If a diagram contains an implicit block, the overall continuous-time system to be solved becomes implicit and `Daskr` is used by Scicos simulator.

The solver properties discussed here are those of the two solvers mentioned above, however these properties are common to most modern solvers. The most important of these properties is that these solvers require that the system be sufficiently smooth over an integration period. This means that Scicos simulator must make sure to stop and reinitialize the solver at each potential point of non-smoothness (discontinuity, discontinuity in the derivative, etc...).

The other important property is that these solvers are variable step, meaning the discretization is not regular in time and more importantly, the solver can take a step forward in time and then later step back so that the evaluation calls do not constitute an increasing time sequence.

Another important property is the possibility to set constraints on solver. For example constraints on the relative and absolute error tolerances can be imposed globally or on each state variable separately. Time constraint can be imposed to forbid the solver to advance time beyond a given time called the *stopping time*. Normally the solver is allowed to step beyond the final integration time and returns the value at final time by interpolation. This increases the efficiency when the integration is restarted.

Finally there are complex issues related to re-initialization as far as the implicit blocks are concerned and the use of `Daskr`. We do not discuss these issues here.

Parameterization of the solver

Besides obvious parameterizations such as setting various error tolerances, maximum step size, etc..., the simulator must automatically start and stop the simulation when necessary.

When an event occurs, the continuous-time simulation must stop so that the event-triggered components of the system can be activated. Once this is done, the continuous-time simulation can proceed. So events times are the times of stop and restart of the continuous-time simulation. Although they are similar in this way, the events originating from different sources, perform different tasks. Hence they have different importances and properties. In continue, two important properties of events will be introduced.

Event criticality

Although all events force the simulator to stop ad restart the integration, they are however not equal in importance. Consider Scicos diagrams in Fig. 2 and Fig. 3. In the first diagram, the event generated by the *Event Clock* drives the *Scope*. The integrator (`1/s`) receives on its regular input port the sine function which is generated by the *ever-active*, *sinusoid generator* block (ever-active means the block is always active[1]). The output of the integrator is sampled by the *Scope* at its activation times (times at which it receives an activation on its activation input port, (i.e., the times of the events generated by the *Event Clock*). In this case, the solver must be stopped at each of these times, however, since the corresponding events do not produce any non-smoothness on the input of the integrator, there is no reason to re-initialize the solver (do a *cold restart*). We say this event is *not critical*.

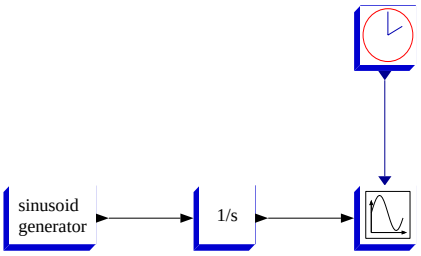

Figure 2: Non-critical event.

In the second diagram, the event activates the *random generator* block which outputs a random variable. This output remains constant until the next event reactivates the block. The integrator then receives a piecewise constant signal to integrate. Thus the event in this case creates a discontinuity at the input of the integrator. Clearly in this case the solver must be re-initialized. This event is *critical*.

Furthermore, an event can be critical in another way: an event can cause a jump in the internal state of a block. One such example is the integrator with reset on event. Clearly if a jump is produced in the state, the solver must be restarted cold.

[1] The activations in Scicos, in general, come through activation signals via activation ports, however, for the sake of simplifying diagram construction, ever-active activations are not explicitly drawn.

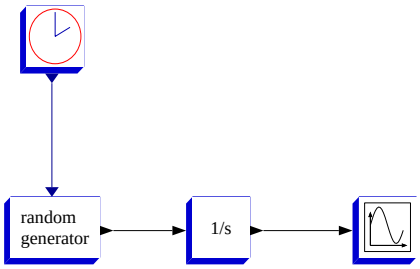

Figure 3: Critical event.

Event predictability

We have seen that events can be put into two categories: critical and non-critical. But they can also be put into two other categories: *predictable* and *non-predictable*. Consider the following system:

$$\dot{x} = \begin{cases} x\sqrt{1-t} & \text{if } t \leq 1 \\ 0 & \text{if } t > 1 \end{cases}$$

To model this system in Scicos, we need to generate an event at time $t = 1$ in order to change the dynamics of the system. Clearly this event is critical. See Scicos diagram illustrated in Fig. 4.

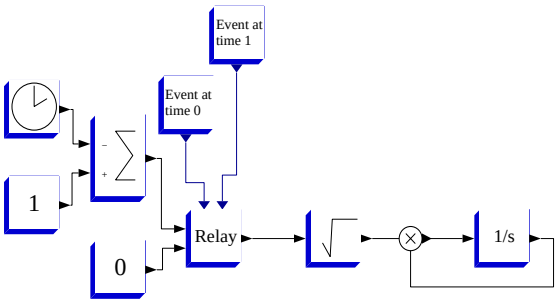

Figure 4: Predictable event.

But this event is also predictable. We know ahead of time its occurrence time ($t = 1$). In general, most events in a Scicos diagram are generated by *Event Clocks* and their times are predictable.

Non-predictable events are *zero-crossing* events. These events are generated when a signal crosses zero and their activation times are not known in advance. A predictable event can be considered and modeled as a non-predictable event. For example in the above example, the event at time 1 can be obtained by using a zero-crossing test on the function $1 - t$. But that would be inefficient for two reasons: first, the zero-crossing tests are additional work for the solver. Second, to detect a zero-crossing, the solver has to go beyond the crossing and perform iterations to pin-point exactly the location of the crossing. In the above example, this results in an error since for $t > 1$, $\sqrt{1-t}$ is not defined. See Scicos diagrams illustrated in Fig. 5 and Fig. 6. In the Scicos diagram illustrated in Fig. 5, the Zcross block defines a zero-crossing surface. The solver has to go beyond the surface in order to find the exact time of crossing. That is why it attempts to evaluate the value of $\sqrt{1-t}$ for t larger than 1. The same system is implemented in a slightly different way as illustrated in Fig. 6. Here, the *if-then-else* block redirects its activation signal to one of its output activation ports depending on the value of its regular input. If this latter is positive, the activation goes through the *then* port, if not, it goes through *else*. The *if-then-else* and the *event-select* blocks are the only blocks which redirect activation signals without creating delays. These blocks are referred to as *Synchro blocks*. They are the counterparts of conditional statements *if-then-else* and *switch* in programming languages such as C. *Synchro* blocks have built-in zero-crossing surfaces that generate an event when the activated output port changes. Because such a change may produce discontinuity and thus the solver must be informed. The presence of this zero-crossing forces the solver to step beyond the $t = 1$, and as a result, the simulation fails again.

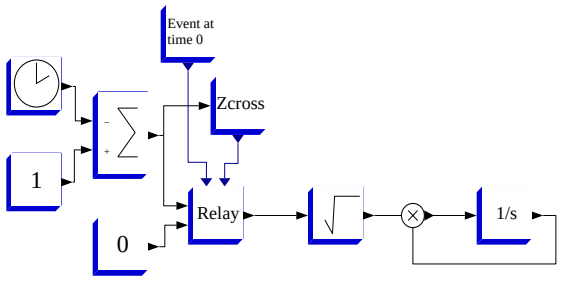

Figure 5: The simulation fails in this case.

If the event is treated as predictable, the simulator, using the fact that the upcoming event at time 1 is critical, sets the critical time to 1 preventing the solver to step beyond $t = 1$.

Critical event classification

In the previous sections the importance of Critical events has been shown. Here the definition, the classification algorithm, and its applications in simulation will be discussed.

Definition 1 *A discrete event is critical if upon activation, either an input of a block, containing zero-crossing*

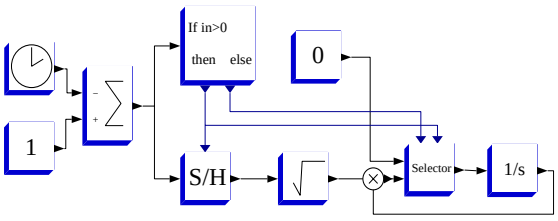

Figure 6: The simulation fails also in this case.

surfaces or continuous-time states, is updated, Or the continuous-time states of a block jump.

Before explaining critical event classification algorithm, the following items should be noted.

- The critical property concerns only discrete events. Therefore to avoid considering a continuous-time event as critical, all pure continuous-time activation links must be excluded from the classification procedure.
- Block having direct input-output relationships (direct feed-through) and ever-active blocks can pass on the discontinuities arriving at their inputs to the following blocks. These blocks are called DTB (Discontinuity transmitting block).
- Synchro blocks that do not have an input event port and inherit the continuous-time activation, are referred to as Continuous Synchro (CS) blocks.

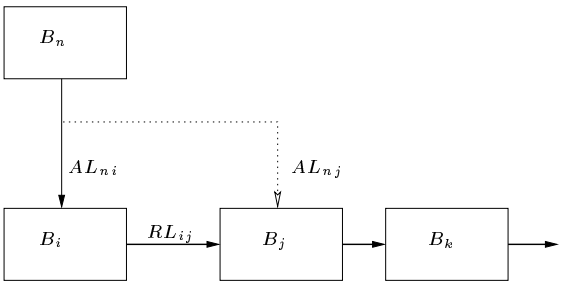

Figure 7: Event inheritance

Algorithm:

1. Propagate the events through DTB's to determine the blocks that potentially subject to discontinuity at their inputs. For example in Fig. 7, IF B_n block activates B_i block (here i.e. AL_{ni}), AND IF a regular link between B_i and B_j blocks exists (i.e. RL_{ij}), AND IF the B_j block is a DTB, THEN an explicit activation link between B_n Block and B_j block (i.e. AL_{nj}) is established[1]. Repeat this process until no more activation link can be established.

2. Find all CS blocks (Synchro blocks not activated explicitly) and store them in CS-list.

3. Identify all the blocks activated by CS blocks and add them to the list of DTB's. For example, in Fig. 8, the B_j block is activated by CS_n. so it should be added it to the list of DTB's. The reason is that the block B_j receives a continuous-time activation through CS_n therefore it can pass the eventual discontinuities at its input port into the B_k block.

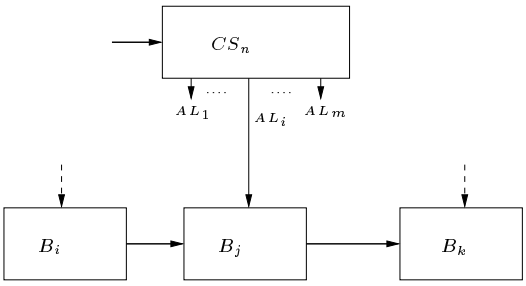

Figure 8: Continuous Synchro Block

4. Remove all activation links originating from CSs (e.g. in Fig. 7 all AL_i links) and then remove CSs from the diagram.

5. Go to step 1 and repeat until in step 2 CS-list becomes empty.

6. For each event source, search through all the blocks it activates. If any of them contains continuous-time states or zero-cross surfaces, then flag the event as critical.

At later stages of compilation, Synchro blocks (i.e., *if-then-else* and *event-select*) are duplicated if they have multiple sources of activation. So each Synchro, at the end, is activated just by one activation source [Nikoukhah and Steer 1996-a, Nikoukhah and Steer 1997]. This process is done after critical event classification and the newly generated events inherit the property of the original events.

Use of critical event classification in simulation

The classification of the events allows us to avoid unnecessary cold restarts. Without it, at every event time, the solver should be cold restarted to make sure that the numerical solver uses consistent information.

[1] All the changes made in the diagram are discarded at the end when the critical events are identified so that other phases of compilation can be carried out normally

The lack of consistency is avoided in two ways. If the critical event is fired immediately, a cold restart is performed. Because in such a case, a discontinuity may occur in a signal (or in its derivative) fed to a block containing continuous-time states or zero crossing surfaces. This discontinuity may cause the solver to fail during restart. This is particularly a problem in Scicos because Scicos uses a BDF (backward differentiating Formula) method to integrate the continuous-time systems.

Another situation where inconsistency can occur is when a critical event is programmed for a future time. If the solver had been allowed to look beyond this time, the new critical event may affect values which are already used by the solver, i.e., values beyond this time. This change of model can result in a failure of the solver which expects to receive consistent informations. To see more precisely what happens in this case, note that after each event at time $t(i)$, when the integration resumes, the start-time $t(i)$, the end-time $t(i+1)$, and the time beyond which the solver is not allowed to explore $t_{\text{stop}}(i)$ are passed to the solver to integrate the system from $t(i)$ to $t(i+1)$. Note that the the solver may advance the time beyond $t(i+1)$ and compute the solution at $t(i+1)$ by interpolation. Let t_{max} be the largest value of t for which the solver has requested an evaluation of the function; clearly $t(i+1) \leq t_{\text{max}} \leq t_{\text{stop}}(i)$. After the processing of the event(s) at time $t(i+1)$, the integration resumes from $t(i+1)$. If the events at $t(i+1)$ program a critical future event at a time t_c earlier than t_{max} then the function evaluations done in the previous integration period are no longer valid. One way to make sure that such a situation does not occur is to force a cold-restart if $t_c < t_{\text{stop}}(i)$. Note that in this case we set $t_{\text{stop}}(i+1) = t_c$ if not $t_{\text{stop}}(i+1) = t_{\text{stop}}(i)$.

Simulation optimization

To each event, we associate a list specifying the blocks which are to be activated when the event fires and the order in which they should be activated.

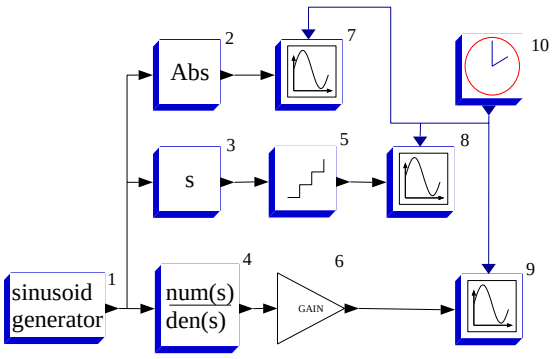

Figure 9: Execution order.

The same holds for continuous-time activations (as it was said before, Scicos treats the continuous-time activation similarly to the way it handles discrete events). A continuous-time activation is a call to the system to update its continuous-time components. For that, there is an activation list and an order in which the activations must be realized. We call it *CORD* list.

For example, in Fig. 9, to update the system at the requested times, the blocks $\{(1), (3, 4), (2, 5, 6)\}$ should be called to update their outputs. During the continuous-time simulation (when the solver is at work), the 3, 4 blocks are called to deliver their updated residuals ($f(\dot{x}, x, t, u)$). Note that these blocks are the blocks in *CORD* which contain continuous-time states.

Note that the *CORD* can be used both for updating the continuous-time parts of the system at specific times and also during the integration by the solver. So unlike discrete events, continuous-time events are evoked in three distinct situations:

1. In the simulator, to update the continuous-time parts at a requested time.
2. By the differential equation solver, to update the continuous-time parts so that system residuals can be correctly evaluated.
3. By the differential equation solver, to update the values of the zero-crossing surfaces.

It turns out that in the second and third cases, we don't necessarily need to activate all the blocks in the *CORD* list. In the second case, we only need to update the outputs of blocks which affect directly or indirectly the input of a state bearing block. Same for the third case except that blocks containing zero-crossing surfaces must be considered. For example in Fig. 9, the 2, 5, 6 blocks which are in *CORD* don't have any effect on the residuals. Thus, they can be excluded from *CORD* and be stored in a new list. The new list, *OORD* list, is comprised of all the blocks which their outputs may affect the input to a block containing a continuous state. The blocks affecting the zero-crossing surfaces (in this case, *Abs* and the *Quantizer* blocks) are 1,2,3,5. We call this list *ZORD*.

The use of *OORD* and *ZORD* optimizes the number of calls to each block in the integration phase. This optimization is very important for simulation efficiency because the solver requires many function evaluations (depending on the stiffness/nonlinearity of the system and the required precision). In a typical application, for every evaluation due to a discrete event, the solver requires hundreds or even thousands of continuous-time function evaluations.

Conclusion

In this paper, we have presented the mechanism by which properties of events (critical, predictable) are extracted

automatically and used to generate an appropriate parameterization for the numerical solver. We also discussed a technique for increasing simulation efficiency.

References

[Bunks et al. 1999] C. Bunks, J. P. Chancelier, F. Delebecque, C. Gomez (ed.), M. Goursat, R. Nikoukhah and S. Steer, *Engineering and Scientific Computing with Scilab*, Birkhauser, 1999.

[Chancelier et al. 2002] J. P. Chancelier and F. Delebecque and C. Gomez and M. Goursat and R. Nikoukhah and S. Steer, *An introduction to Scilab*, Springer-Verlag, 2002.

[Benveniste 1998] A. Benveniste, *Compositional and Uniform Modeling of Hybrid Systems*, IEEE Trans. Automat. Control, AC-43, 1998.

[Nikoukhah and Steer 2000] R. Nikoukhah and S. Steer. *Conditioning in hybrid system formalism*, ADPM, Dortmund, Germany, Sept. 2000.

[Hindmarsh 1980] A. C. Hindmarsh, *Lsode and Lsodi, two new initial value ordinary differential equation solvers*, ACM-Signum Newsletter, no. 4, 1980.

[Petzold 1983-a] L. R. Petzold, *Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations*, SIAM J. Sci. Stat. Comput., No. 4, 1983.

[Petzold 1983-b] L. R. Petzold, *A Description of DASSL: A Differential/Algebraic System Solver, in Scientific Computing*, R. S. Stepleman et al. (Eds.), North-Holland, Amsterdam, 1983.

[Brown et al. 1998] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold, *Consistent Initial Condition Calculation for Differential-Algebraic Systems*, SIAM J. SCI. COMP., NO. 19, 1998.

[NIKOUKHAH AND STEER 1996-A] R. NIKOUKHAH AND S. STEER, *Scicos a dynamic system builder and simulator*, IEEE INTERNATIONAL CONFERENCE ON CACSD, DEARBORN, MICHIGAN, 1996.

[NIKOUKHAH AND STEER 1996-B] R. NIKOUKHAH AND S. STEER *Hybrid systems: modeling and simulation*, COSY: MATHEMATICAL MODELLING OF COMPLEX SYSTEM, LUND, SWEDEN, SEPT. 1996.

[NIKOUKHAH AND STEER 1997] R. NIKOUKHAH AND S. STEER, *Scicos: A Dynamic System Builder and Simulator, User's Guide - Version 1.0*, INRIA TECHNICAL REPORT, RT-0207, JUNE 1997.

[NIKOUKHAH ET AL,1999] S. STEER, R. NIKOUKHAH, *Scicos: a hybrid system formalism*, ESS'99, ERLANGEN, GERMANY, OCT. 1999.

[NIKOUKHAH AND STEER 1999] R. NIKOUKHAH AND S. STEER, *Hybrid system modelling and simulation*, FEMSYS'99, MUNICH, GERMANY, MARCH 1999.

[DJENIDI ET AL. 2001] R. DJENIDI, R. NIKOUKHAH AND S. STEER, *A propos du formalisme Scicos*, MOSIM'01, TROYES, FRANCE, APRIL 2001.