Improving Efficiency of Hybrid System Simulation in Modelica

Victorino Sanz
Dpto. Informática y Automática, ETSI Informática, UNED
Juan del Rosal, 16
28040, Madrid, Spain
vsanz@dia.uned.es

Alfonso Urquia
Dpto. de Informática y Automática, ETSI Informática, UNED
Juan del Rosal, 16
28040, Madrid, Spain
aurquia@dia.uned.es

Francesco Casella
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano
Piazza Leonardo da Vinci, 32
20133, Milano, Italy
francesco.casella@polimi.it

ABSTRACT

Modelica supports the modeling of hybrid systems by means of differential and algebraic equations, and discrete-events. The simulation of Modelica models is carried out by solving the continuous-time equations, detecting the occurrence of events and managing the triggered events. After managing an event, the event iteration algorithm is used to find consistent restart conditions before resuming the numerical integration. The objective of the work presented in this manuscript is to describe a procedure to optimize the event iteration algorithm. The structure of the model is analyzed in order to only re-evaluate the equations involved in the management of events. The use of this procedure is described by means of an application example.

Categories and Subject Descriptors

I.6 [**Computing Methodologies**]: Simulation and Modeling; I.6.2 [**Simulation and Modeling**]: Simulation Languages—*Modelica*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Combined, Hybrid*; G.4 [**Mathematics and Computing**]: Mathematical Software—*Algorithm Design and Analysis*; I.1.2 [**Computing Methodologies**]: Algorithms—*Analysis of algorithms*

General Terms

Algorithms, Languages, Performance

Keywords

Hybrid system modeling, event iteration, simulation performance, Modelica

1. INTRODUCTION

Modelica is a general-purpose object-oriented modeling language [16]. It is mainly designed to describe mathematical models of physical systems using differential, algebraic and discrete equations. It also supports the management of discrete-events, thus facilitating the description of hybrid-systems (i.e., systems with combined continuous and discrete behaviors) [15, 5, 8, 17].

In order to simulate a Modelica model its equations have to be translated into executable code. As a first step, the model equations are expanded into a flat set of statements, named the flat hybrid differential and algebraic equation (DAE) model. These manipulations are described with detail in [6] and [11].

Event conditions are checked during the simulation and if any event is triggered, the solution of the DAE system is halted and the event is managed. After each event, the simulator (Dymola in our case [7]) has to find consistent restart values for the variables of the hybrid DAE model before resuming the integrator for the continuous-time part. This procedure is called event iteration. A similar approach is used in the OpenModelica compiler [3].

During the event iteration, after handling each event, the entire model is re-evaluated. However, the management of the event may not affect the whole model. For example, in hybrid systems, an event may propagate a chain of events along the discrete-event part of the model without involving any change in the continuous-time part. A chain of events takes place when the management of an event immediately triggers another event, and so on. During the management of these chains of events, an event iteration is performed after each event and the whole model is re-evaluated during the iteration, even if the events are local to a sub-system of the discrete-event part (e.g., see case studies presented in [20] and [21]). This situation is demonstrated with an example in Section 5.

Following the approach described above, unnecessary evaluations of some model equations may be performed, which may significantly degrade simulation performance. On the one hand, due to performing unnecessary event iterations when the values of some variables are changed during the event but no further events are triggered. On the other hand, when the event iteration is needed, due to the evaluation of equations not involved in the event. Similarly, an improvement for the detection of the events, in order to avoid unnecessary evaluations of equations during root-finding, is presented in [12].

The objective of the work presented in this manuscript is to describe a procedure to optimize the execution of event iterations. The idea is to analyze the dependencies between

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
EOOLT 20014, October 10 2014, Berlin, Germany
Copyright 2014 ACM 978-1-4503-2953-8/14/10 ...$15.00
http://dx.doi.org/10.1145/2666202.2666205.

the modification of the values of some variables during an event and their effect in the conditions that trigger additional events. In some cases some parts of the model will not be affected by the events triggered, and so they should not be involved in their management. In general, the simulation of complex hybrid systems such as communication networks [9], statecharts [10], DEVS [20, 2], Cellular Automata [22], Petri Nets [18] or logistic systems [19, 14, 13] will be improved with this procedure. The proposed procedure does not modify the usual manipulation performed to the Modelica model, but provides additional information based on the description of the hybrid DAE model.

The relationship between the discrete-event and the continuous-time parts of the system is considered by other simulators in order to improve performance. For example, Saber, by Synopsys [24, 23], uses the patented Calaveras algorithm to independently simulate the discrete-event part and the continuous-time part of mixed-signal systems. The simulation of both parts is efficiently synchronized by the algorithm in order to ensure the correctness of the overall simulation.

The structure of this manuscript is as follows. A brief description of the hybrid DAE model, the simulation of Modelica models and the event iteration procedure is presented in Section 2. The proposed procedure to analyze the model equations and to optimize the execution of event iterations is presented in Section 3. The detailed description of the algorithm is included in Section 4. In order to demonstrate the use of the proposed procedure, an application example is described in Section 5. The manuscript is finished with some conclusions in Section 6.

2. SIMULATION OF MODELICA HYBRID MODELS

A Modelica model has to be translated into a flat hybrid DAE model in order to be simulated [11, 6]. According to the Modelica specification (version 3.2, revision 2) [16] the hybrid DAE model is defined as (the special cases of the `noEvent()` and the `reinit()` operators are not considered):

$$\mathbf{c} := \mathbf{f}_c(relation(\mathbf{v})) \quad (1)$$

$$\mathbf{m} := \mathbf{f}_m(\mathbf{v}, \mathbf{c}) \quad (2)$$

$$\mathbf{0} = \mathbf{f}_x(\mathbf{v}, \mathbf{c}) \quad (3)$$

where,

$\mathbf{v}$	is $\mathbf{v} := [\dot{\mathbf{x}}; \mathbf{x}; \mathbf{y}; t; \mathbf{m}; \mathbf{pre}(\mathbf{m}); \mathbf{p}]$.
$\mathbf{p}$	are the variables declared as *parameter* or *constant*.
t	is the independent variable, *time*.
$\mathbf{x}(t)$	are variables of *Real* type that appear differentiated.
$\mathbf{m}(t_e)$	are the unknown discrete variables, of *discrete Real*, *Boolean* and *Integer* types. Their values only change at event instants t_e. $pre(\mathbf{m})$ are the values of $\mathbf{m}$ immediately before the event.
$\mathbf{y}(t)$	are the algebraic variables of *Real* type.
$c(t_e)$	are conditions of if-expressions and converted when-clauses (used to trigger events).
$relation(\mathbf{v})$	are relations containing variables v_i.

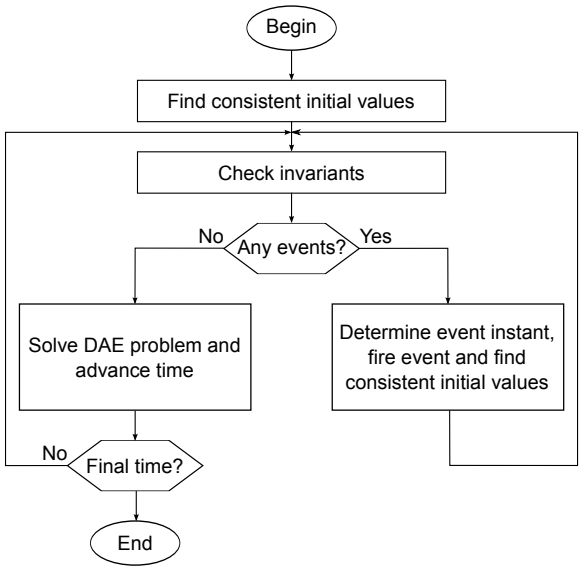

Figure 1: Hybrid DAE simulation algorithm [1].

The simulation of the model is summarized in the diagram shown in Fig. 1. The simulation starts by finding consistent initial values for the model variables. After that, the DAE equations (Eq. (3)) are solved using a numerical integrator. Conditions c and discrete variables $\mathbf{m}$ are kept constant. During integration, the relations from Eq. (1) (i.e., the invariants) are monitored in order to detect changes in their values (i.e., triggered events). When an event is triggered the integration is halted and an iteration is performed to find the interval in which the event has occurred, given a pre-defined error threshold [4]. Time events (i.e., relations that depend only on time) can be pre-scheduled (e.g., using a calendar of events). At the event instant, the model consist of a mixed set of algebraic equations that has to be solved for the Real, Boolean and Integer unknowns.

After the event is treated, consistent initial values have to be found before the integration is restarted. The procedure used to re-initialize the model equations after the event (named *event iteration*) is shown in Algorithm 1 [16]. Basically, this procedure re-evaluates the model until the value of the discrete variables remains unchanged. If the value of any discrete variable changes, additional events may be triggered during the same time instant.

Due to the synchronous data flow principle and the single

Algorithm 1: Modelica event iteration procedure.

known variables : $\mathbf{x}$, t, $\mathbf{p}$
unknown variables: d$\mathbf{x}$/dt, $\mathbf{y}$, $\mathbf{m}$, pre($\mathbf{m}$), c
`// pre(m) = value of m before event occurred`
repeat
  solve (1),(2) and (3) for the unknowns, with pre($\mathbf{m}$) fixed;
  if $\boldsymbol{m} == pre(\boldsymbol{m})$ **then**
    break;
  pre($\mathbf{m}$) := $\mathbf{m}$;
until;

Listing 1: Model `NoEventIter1`.

```
model NoEventIter1
  Real x (start = 0);
  Real y (start = 0);
  Real z (start = 0);
equation
  der(x) = 1;
  when y > 0.5 then
    z = pre (z) + 1;
  end when;
  when x > 1 then
    y = pre(y) + 1;
  end when;
end NoEventIter1;
```

assignment rule, a change in a discrete variable that is also used in another equation, or condition, without using the *pre()* operator does not require event iteration. The reason for this is because the BLT ordering of the equations provides the correct order for evaluating the equations. It is dependent on the implementation to use this characteristic or to find new consistent initial values by means of event iterations. The example model `NoEventIter1`, shown in Listing 1, can be evaluated as follows.

Equation `z = pre(z)+1` will be ordered after equation `y = pre(y)+1` since the former is contained within a condition that used the value of `y`, that is computed in the latter. When `x > 1` becomes true a new value for `y` is computed, and this change may trigger `y > 0.5` event, which will be computed later in the BLT. At the end of the evaluation of the model equations, all feasible events had been checked and computed if their conditions became true.

On the other hand, a change in a discrete variable that is involved in another equation, or condition, under the *pre()* operator (e.g., `when pre(x) > 0.5 then`) may need event iterations. A condition using `pre(v)` may be ordered before the computation of `v` in the BLT. As stated in the Modelica specification (Section 3.7.3.1):

"A new event is triggered if at least for one variable v "pre(v) <> v" after the active model equations are evaluated at an event instant. In this case the model is at once reevaluated..."

Consider the example shown in Listing 2.

Listing 2: Model `NoEventIter2`.

```
model NoEventIter2
  Real x (start = 0);
  Integer c (start = 0);
equation
  der(x) = 1;
  when x > 0.5 then
    c = pre(c)+1;
  end when;
end NoEventIter2
```

When the state event is triggered (`x > 0.5`), the value of `c` becomes different than `pre(c)` due to the assignment inside the `when` clause, and according to the specification an event iteration has to be performed. However, the state event will not be triggered again, and the event iteration could be avoided. In this case, the difference between values of `c` and `pre(c)` is irrelevant for the event iteration. The same situation can be observed in the previous example - i.e., `NoEventIter1` - where `y <> pre(y)` and `z <> pre(z)` at the end of the evaluation of the event. An *a-priori* analysis of the model equations and the event conditions could be used to avoid unnecessary event iterations. The procedure described next performs this analysis.

3. OPTIMIZING EVENT ITERATIONS

When an event is triggered, Algorithm 1 specifies how the consistent restart conditions for the model can be computed.

When solving Eqs. (1)-(3) for the first time, the values of some variables may change, possibly triggering further events. However, if BLT ordering is used to solve Eqs. (1)-(3), assuming a fixed value for $pre(\mathbf{m})$ and taking into account all dependencies, all conditions not containing $pre(\mathbf{m})$ will be properly ordered in the BLT, and the correct sequence of further events to be triggered will automatically stem from the ordered solution of the BLT blocks, as shown by the example in Listing 1. Since the last statement in the loop only changes the value of $pre(\mathbf{m})$, these conditions will not be triggered in the next iterations; therefore, all the equations depending on these conditions can be removed from the set of equations to be solved iteratively, and the values of their corresponding variables considered as fixed for the next iterations. The same argument applies to time-induced conditions, which are also not affected by the change of $pre(\mathbf{m})$.

On the other hand, conditions using *pre()*, for example `pre(x) > 1`, may be ordered in the BLT before the calculation of the variable under *pre()*, e.g., `x = pre(x)+1`. As a consequence, a change in the variable may trigger an additional event whose condition has to be checked during an additional evaluation (i.e., the event iteration). The goal of the algorithm is to identify the minimal set of equations affected by the change of the *pre()* variable, thus avoiding unnecessary computations during the event iteration.

In order to do so, one can interpret the termination condition of Algorithm 1, $m = pre(\mathbf{m})$, as an additional set of constraint equations that must be fulfilled by the computed consistent restart conditions. A suitable dependency graph can be built, to analyze the dependencies in the system that results from adding these constraint equations to the original set (1)-(3), and removing those equations that depend on conditions not containing *pre(m)*, which are irrelevant.

Any loop in the graph that connects an equation used to compute a variable x, the equation added to represent the exit condition $x == pre(x)$, the condition including that previous value $pre(x)$, and the equation(s) restricted by that condition, will then identify a set of equations that needs event iterations to be solved, due to cyclic dependencies. Equations and variables which are outside such loops and which are not influenced by the variables computed in such loops will not be affected by the change $pre(m) := m$ at the end of each iteration; therefore, they can be excluded from the event iteration loop, avoiding unnecessary computations.

The following concepts are used in the description of the algorithm:

- *pre-node*: is a node in the graph that represents the equation used to update the previous value of a variable (e.g., `pre(x) = x`).
- *condition-node*: is a node in the graph that represents a discrete-time expression of the model that contains at least a variable under the *pre()* operator, and is used as a condition for an event.

- *loop-node*: is a node in the graph that belongs to a loop. A loop is composed at least by a *pre-node* and a *condition-node*.
- *update-node*: is a node that does not belong to a loop but receives an edge from a *loop-node*, or from another *update-node*.

4. ALGORITHM DESCRIPTION

The procedure starts considering the BLT ordered equations of the flat model. Some equations may be restricted under certain conditions (i.e., defined inside when- or if-expressions). Some of the variables involved in these conditions may be affected by the *pre()* operator. Each set of equations inside the same when clause is divided into individual *when* clauses, each containing one equation. This procedure is based on the analysis of the relationship between the changes in the discrete variables and their appearance in the conditions using the *pre()* operator.

The procedure is described next, with the help of the example shown in Listing 3:

Listing 3: Example model.

```
model example
  parameter Real a = 1;
  parameter Real b = 1;
  parameter Real c = 1;
  Real x( start = 1, fixed = true);
  Real y( start = 0, fixed = true);
equation
  when y > 3 and pre(x) > 0 then
    x = a + b;
  end when;
  der(y) = x + c;
end example;
```

1. Remove all equations declared inside when clauses whose conditions may trigger time events. Also, remove the non-active branch of if-expressions whose condition may also trigger time events. Time events will not be triggered during event iteration (i.e., time is not advanced) and so, it is not necessary to evaluate those equations. Eq. (1), of the hybrid DAE model, is then reduced to conditions that trigger state events. In the example there are no time event conditions.

2. Remove all equations declared inside when clauses whose conditions do not include the *pre()* operator. Similarly, remove the non-active branch of if-expressions whose condition does not include the *pre()* operator. In the example the condition of the `when` clause includes `pre(x)`, so equation `x = a + b` is not removed.

3. If all equations have been removed, event iteration can be avoided and the procedure ends. In this case, the modification of the discrete variables of the model during an event is not related with the conditions that may trigger additional state events.

4. If not, add the equations corresponding to the updates of *pre()* variables described in Algorithm 1, that is, $pre(x) = x$.

5. Using the current system of equations, construct a directed graph of equation-variable relations. The nodes of the graph represent each equation of the system and the variable that is computed using that equation, including the *pre-nodes* corresponding to the equations added in the previous step. The edges of the graph go from the node in which a variable is computed to the nodes that represent the equations that use the previously computed variable. In the case of an algebraic loop in a sub-system of equations, the whole sub-system of equations may be represented using a single node with multiple edges representing all the variables computed in the loop. The directed graph corresponding to the example presented in Listing 4 is shown in Fig. 2.

Listing 4: Modelica flat model including *pre()* update equations.

```
1  when y > 3 and pre(x) > 0 then x := a+b; end when;
2  der(y) := x+c;
3  pre(x) := x;
```

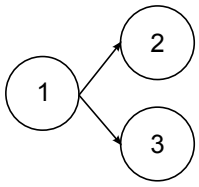

Figure 2: Directed graph for example code (node numbers correspond to line numbers in Listing 4).

6. Add the *condition-nodes* to the graph. These conditions will include variables using the *pre()* operator. The equations of the example including the conditions are presented in Listing 5.

Listing 5: Modelica flat model including conditions.

```
1  when y > 3 and pre(x) > 0 then x := a+b; end when;
2  der(y) := x+c;
3  pre(x) := x;
4  (y > 3 and pre(x) > 0)
```

7. Add edges to the graph that correspond to the relationships between the variables of the conditions and their corresponding *condition-nodes*. The new nodes and edges corresponding to the *condition-nodes* are shown in Fig. 3 (i.e., node 4).

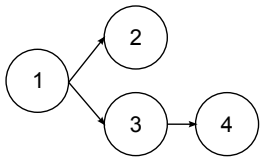

Figure 3: Directed graph including *condition-nodes*.

8. Add an edge from each *condition-node* to the nodes of equations whose computation is restricted by the

condition (i.e., equations that will be active when the condition is satisfied). These new edges are shown in Fig. 4 for the example graph. Some loops will be formed in the graph, due to the dependencies between *pre-nodes*, *condition-nodes* and nodes restricted by *condition-nodes*.

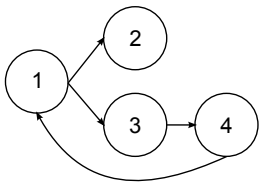

Figure 4: Directed graph including edges from *condition-nodes*.

9. If the graph contains any loop composed of a pair of nodes, merge these nodes into one single node. These pairs correspond to the use of the *pre()* value of a variable to calculate its new value (e.g., `x = pre(x) + 1`).

10. Identify the nodes of the graph that belong to a loop, named *loop-nodes* (e.g., nodes 1, 3 and 4 in the example). These nodes will have to be evaluated in the event iteration, because the loop indicates that the modification of a discrete variable and the update of its *pre()* value may modify the value of a condition and trigger a new event. In the case of nodes merged in the previous step, both nodes will become *loop-nodes*.

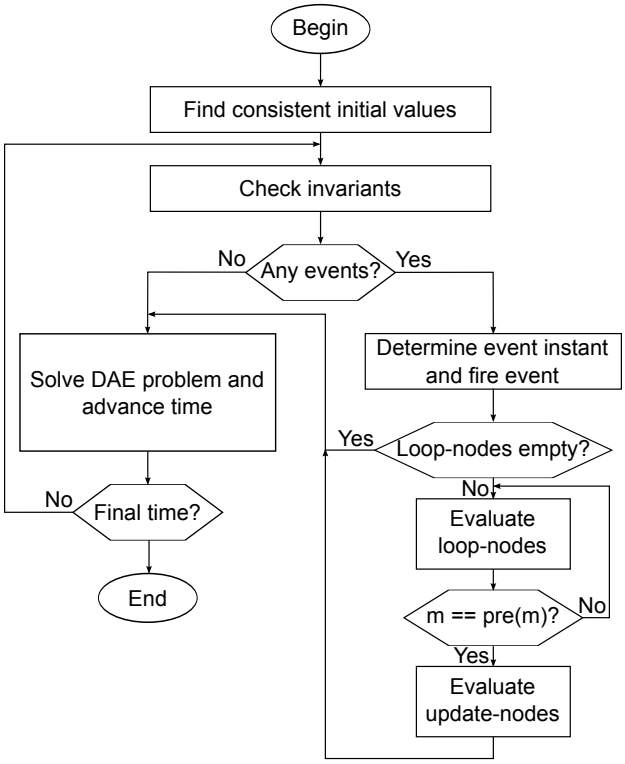

Figure 5: Adapted hybrid DAE simulation algorithm.

11. *Update-nodes* will have to be evaluated once before restarting the integration, in order to correctly update the value of their variables with the final value of the variables involved in the event iteration. This corresponds to node 2 in the example.

12. The rest of the nodes (i.e., nodes without edges or nodes with edges pointing to any *loop-* or *update-node*) does not have to be evaluated in the event iteration. In such a simple example this set of nodes is empty, but it will contain some nodes in the application example discussed in Section 5.

This procedure can be computed in advance, previously to the simulation, during the translation of the model. The simulation algorithm can be adapted to evaluate *loop-nodes* and *update-nodes* equations. The diagram of the adapted simulation algorithm is shown in Fig. 5.

5. APPLICATION EXAMPLE

In order to demonstrate the described procedure, an application example is presented. This model, named `hybridSys`, describes a hybrid system where the discrete and the continuous parts are clearly separated. The only interaction is by using the values of `x_End` and `x1` to compute `x2`, using the function named `func2`. The parts, phases and behavior of the model are shown in Fig. 6. The Modelica source code for this model is shown in Listing 6.

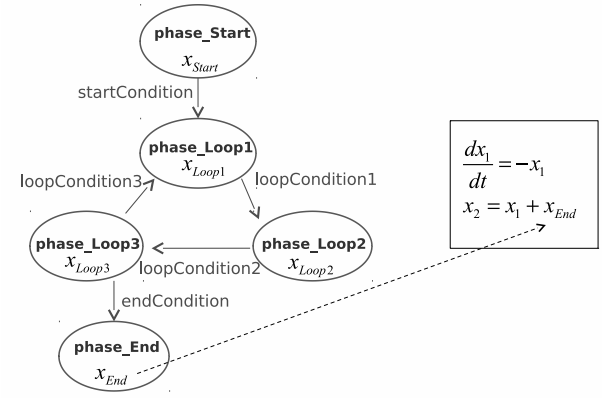

Figure 6: Diagram describing relations between phases and equations for the `hybridSys` model.

Listing 6: Modelica code of `hybridSys` model.

```
package Example2

function func1
  input Real x;
  output Real func1_out;
algorithm
  func1_out := x;
end func1;

function func2
  input Real x1;
  input Real x2;
  output Real func2_out;
algorithm
  func2_out := x1 + x2;
end func2;
```

```
model hybridSys
  parameter Integer Niter = 4;
  // Variables of the discrete event model
  Boolean phase_Start(start=true);
  Boolean phase_Loop1(start=false);
  Boolean phase_Loop2(start=false);
  Boolean phase_Loop3(start=false);
  Boolean phase_End(start=false);
  Real x_Start(start=0);
  Real x_Loop1(start=0);
  Real x_Loop2(start=0);
  Real x_Loop3(start=0);
  Real x_End(start=0);
  Boolean startCondition(start=false);
  Boolean loopCondition1(start=false);
  Boolean loopCondition2(start=false);
  Boolean loopCondition3(start=false);
  Boolean endCondition(start=false);
  // Variables of the continuous-time model
  Real x1(start=10),x2;
equation
  // ---------------------
  // Continuous-time model
  der(x1) = -func1(x1);
  // No discrete-to-continuous interaction
  //x2 = func1(x1);
  // Discrete-to-continuous interaction
  x2 = func2(x1,x_End);
  // --------------------
  // Discrete-event model
  startCondition = time > 1;
  loopCondition1  = pre(x_Loop1) < Niter+1;
  loopCondition2  = pre(x_Loop2) < Niter+1;
  loopCondition3  = pre(x_Loop3) < Niter;
  endCondition    = not loopCondition3;
  phase_Start =
    pre(phase_Start) and not startCondition;
  phase_Loop1 =
    pre(phase_Start) and startCondition or
    pre(phase_Loop3) and loopCondition3 or
    pre(phase_Loop1) and not loopCondition1;
  phase_Loop2 =
    pre(phase_Loop1) and loopCondition1 or
    pre(phase_Loop2) and not loopCondition2;
  phase_Loop3 =
    pre(phase_Loop2) and loopCondition2 or
    pre(phase_Loop3) and not (loopCondition3
    or endCondition);
  phase_End =
    pre(phase_Loop3) and endCondition;
  when phase_Start then
    x_Start = pre(x_Start) + 1;
  end when;
  when phase_Loop1 then
    x_Loop1 = pre(x_Loop1) + 1;
  end when;
  when phase_Loop2 then
    x_Loop2 = pre(x_Loop2) + 1;
  end when;
  when phase_Loop3 then
    x_Loop3 = pre(x_Loop3) + 1;
  end when;
  when phase_End then
     x_End = pre(x_End) + 1;
  end when;
end hybridSys;

end Example2;
```

The continuous-time part is computed until the `startCondition` for the discrete event part is met. At that time, the discrete-event part enters a loop of computations until the `endCondition` becomes active, which happens after each phase in the loop is active `Niter` times. The activation of each phase in the loop triggers a new event (i.e., `loopConditionX`), so event iteration is required after the start condition is satisfied. However, during the event iteration the continuous-time equation is also evaluated, even when it will not change until the end condition is active and the loop ends. This situation is illustrated in the extract from the simulation log obtained using Dymola 2013, and shown in Listing 7.

Listing 7: Extract from `hybridSys` simulation log.

```
...
Expression time > 1 became true
  ( (time)-(1) = 5.73541e-13 )
W_[6](1) = 1
func2x_0out(1) = 3.678483823267749
    Variable phase_Loop1 = 1 at time 1
    Variable phase_Start = 0 at time 1
    Variable x_Loop1 = 1 at time 1
Iterating to find consistent restart
  conditions.
func1x_0out(1) = 3.678483823267749
W_[7](1) = 1
func2x_0out(1) = 3.678483823267749
    Variable phase_Loop1 = 0 at time 1
    Variable phase_Loop2 = 1 at time 1
    Variable x_Loop2 = 1 at time 1
Iterating to find consistent restart
  conditions.
...
```

The flat Modelica code for the example is shown in Listing 8. The graph constructed following the described procedure is shown in Fig. 7. This graph also includes the nodes corresponding to the conditions of the five when clauses included in the code (i.e., nodes C1, C2, C3, C4 and C5).

Listing 8: Modelica flat model for model `hybridSys`.

```
1  der(x1) :=  -x1;

2  startCondition := time > 1;
3  phase_Start := pre(phase_Start) and  not
       startCondition;
4  loopCondition3 := pre(x_Loop3) < Niter;
5  loopCondition1 := pre(x_Loop1) < Niter+1;
6  phase_Loop1 := pre(phase_Start) and startCondition
       or pre(phase_Loop3) and loopCondition3 or pre(
       phase_Loop1) and  not loopCondition1;
7  loopCondition2 := pre(x_Loop2) < Niter+1;
8  phase_Loop2 := pre(phase_Loop1) and loopCondition1
       or pre(phase_Loop2) and not loopCondition2;
9  endCondition :=  not loopCondition3;
10 phase_Loop3 := pre(phase_Loop2) and loopCondition2
       or pre(phase_Loop3) and not (loopCondition3 or
       endCondition);
11 phase_End := pre(phase_Loop3) and endCondition;
12 when phase_Start then x_Start := pre(x_Start)+1;
       end when;
13 when phase_Loop1 then x_Loop1 := pre(x_Loop1)+1;
       end when;
14 when phase_Loop2 then x_Loop2 := pre(x_Loop2)+1;
       end when;
15 when phase_Loop3 then x_Loop3 := pre(x_Loop3)+1;
       end when;
16 when phase_End then x_End := pre(x_End)+1; end when
       ;
17 x2 := x1+x_End;
18 pre(phase_Start) := phase_Start;
19 pre(x_Start) := x_Start;
20 pre(x_Loop3) := x_Loop3;
21 pre(x_Loop1) := x_Loop1;
22 pre(phase_Loop3) := phase_Loop3;
23 pre(phase_Loop1) := phase_Loop1;
24 pre(x_Loop2) := x_Loop2;
25 pre(phase_Loop2) := phase_Loop2;
26 pre(x_End) := x_End;
```

The result of the applied procedure is summarized in Table 1. Note that since the continuous-time part of the model is simple, the number of equations not involved in the iteration is reduced. Also, note that nodes 18-26 have been

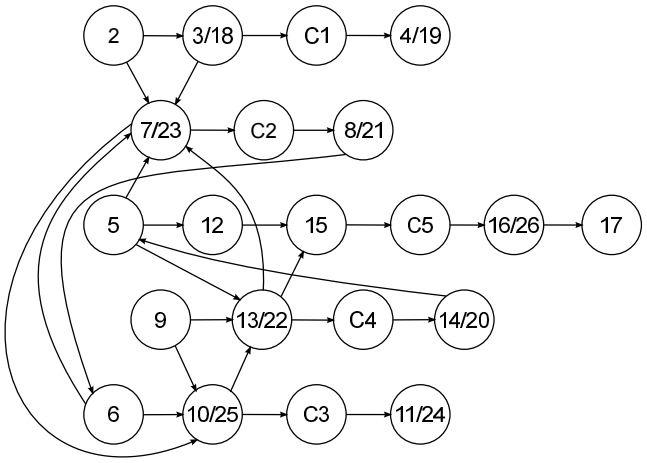

Figure 7: Re-structured graph for `hybridSys` model.

Table 1: Results of the procedure applied to `hybridSys` model

Loop-nodes	5, 6, 7, 8, 9, 10, 11, 13, 14, 20, 21, 22, 23, 24, 25
Update-nodes	12, 15, 16, 17, 26
Nodes not involved	1, 2, 3, 4, 18, 19

included during the procedure and does not represent real equations just the update of the *pre()* values.

6. CONCLUSIONS

A procedure to optimize the execution of event iterations during the simulation of hybrid systems has been proposed. This new procedure can be used to analyze the relationships between the update of *pre()* values of discrete variables in the model and their use within event conditions. The results of the procedure are two subsets of equations: a) *loop-nodes*, that contains the equations involved in the event iteration and; b) *update-nodes*, that are equations to be updated once the event iteration is finished. If the set of *loop-nodes* is empty, event iteration can be avoided. An adaptation of the simulation algorithm has been described in order to use the information provided by this procedure. The use of this procedure has been illustrated using an application example of a hybrid system. Future work will include the implementation of this algorithm in a simulation tool.

7. ACKNOWLEDGMENTS

This research was supported by 2013-026-UNED-PROY grant from UNED and DPI2013-42941-R grant from the Government of Spain.

8. REFERENCES

[1] M. Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1994.

[2] T. Beltrame and F. E. Cellier. Quantised state system simulation in Dymola/Modelica using the DEVS formalism. In *Proceedings of the* 5^{th} *International Modelica Conference*, pages 73–82, Vienna, Austria, 2006.

[3] W. Braun, B. Bachmann, and S. Pross. Synchronous events in the OpenModelica compiler with a Petri Net library application. In *Proceedings of the* 3^{rd} *International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 63–70, Oslo, Norway, 2010.

[4] F. E. Cellier. *Combined Continuous/Discrete System simulation by Use of Digital Computers: Techniques and Tools*. PhD thesis, ETH Zurich, Switzerland, 1979.

[5] F. E. Cellier, H. Elmqvist, M. Otter, and J. H. Taylor. Guidelines for modeling and simulation of hybrid systems. In *Proceedings of the IFAC World Congress*, Sydney, Australia, 1993.

[6] F. E. Cellier and E. Kofman. *Continuous System Simulation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] Dynasim AB. Dymola, Dynamic Modeling Laboratory. User's manual, 2006.

[8] H. Elmqvist, F. E. Cellier, and M. Otter. Object-oriented modeling of hybrid systems. In *Proceedings of the European Simulation Symposium*, Delft, The Netherlands, 1993.

[9] D. Färnqvist, K. Strandemar, K. H. Johansson, and J. P. Hespanha. Hybrid modeling of communication networks using Modelica. In *Proceedings of the* 2^{nd} *International Modelica Conference*, pages 209–213, Oberpfaffenhofen, Germany, 2002.

[10] J. Ferreira and J. E. de Oliveira. Modelling hybrid systems using StateCharts and Modelica. In *Proceedings of the* 7^{th} *IEEE International Conference on Emerging Technologies and Factory Automation*, pages 1063–1069, 1999.

[11] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Computer Society Pr, 2003.

[12] C. Höger. Sparse causalisation of differential algebraic equations for efficient event detection. In *Proceedings of the* 8^{th} *EUROSIM Congress on Modelling and Simulation*, pages 351–356, Washington, DC, USA, 2013.

[13] W. D. Kelton, R. P. Sadowski, and D. T. Sturrock. *Simulation with Arena*. McGraw-Hill, Inc., New York, NY, USA, 4^{th} edition, 2007.

[14] A. M. Law. *Simulation Modelling and Analysis*. McGraw-Hill, New York, NY, USA, 4^{th} edition, 2007.

[15] S. E. Mattsson, M. Otter, and H. Elmqvist. Modelica hybrid modeling and efficient simulation. In *Proceedings of the* 38^{th} *IEEE Conference on Decision and Control*, pages 3502–3507, 1999.

[16] Modelica Association. Modelica - An unified object-oriented language for physical systems modeling. Language specification version 3.2 rev. 2, 2013.

[17] M. Otter, H. Elmqvist, and S. E. Mattsson. Hybrid modeling in Modelica based on the synchronous data flow principle. In *Proceedings of the* 10^{th} *IEEE International Symposium on Computer Aided Control System Design*, pages 151–157, 1999.

[18] S. Pross and B. Bachmann. A Petri Net library for

modeling hybrid systems in OpenModelica. In *Proceedings of the* 7^{th} *International Modelica Conference*, pages 454–462, Como, Italy, 2009.

[19] V. Sanz. *Hybrid System Modeling Using the Parallel DEVS Formalism and the Modelica Language*. PhD thesis, ETSI Informática, UNED, Madrid, Spain, 2010.

[20] V. Sanz, A. Urquia, F. E. Cellier, and S. Dormido. System modeling using the Parallel DEVS formalism and the Modelica language. *Simulation Modeling Practice and Theory*, 18(7):998–1018, 2010.

[21] V. Sanz, A. Urquia, F. E. Cellier, and S. Dormido. Hybrid system modeling using the SIMANLib and ARENALib modelica libraries. *Simulation Modeling Practice and Theory*, 2013(37):1–17, 2013.

[22] V. Sanz, A. Urquia, and A. Leva. 1D/2D cellular automata modeling with Modelica. In *Proceedings of the* 10^{th} *International Modelica Conference*, pages 489–498, Lund, Sweden, 2014.

[23] Synopsis Inc. Saber platform for modeling and simulating physical systems (`http://www.synopsys.com/saber`), 2014.

[24] M. Vlach. Modeling and simulation with Saber. In *ASIC Seminar and Exhibit, 1990. Proceedings., Third Annual IEEE*, pages T/11.1–T/1111, Sep 1990.