NISTIR 7846

An Analysis of Solver-Based Simulation Tools

Ion Matei
Conrad Bock

National Institute of Standards and Technology
U.S. Department of Commerce

NISTIR 7846

An Analysis of Solver-Based Simulation Tools

Ion Matei
Conrad Bock
Systems Integration Division
Engineering Laboratory

http://dx.doi.org/10.6028/NISTIR.7846

March 2012

U.S. Department of Commerce
John E. Bryson, Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Under Secretary of Commerce for Standards and Technology and Director

An Analysis of Solver-Based Simulation Tools

Ion Matei and Conrad Bock

Abstract

Computer-interpretable representations of systems' structure and behavior are at the center of designing today's complex systems. Engineers create and review such representations using (graphical) modeling languages that support specification, analysis, design, verification and validation of systems that include hardware, software, data, personnel, procedures, and facilities (such as the Systems Modeling Language, an extension of the Unified Modeling Language). However, these languages are usually not enough in the analysis and design steps of the engineering process and they must be enhanced with domain specific tools for simulation and performance analysis. These tools are often used separately and sequentially, which reduces the efficiency of the design process. As a result, there is an increasing need for integrating different simulating tools under one common framework. In this report, we analyze a set of general purpose simulation and performance analysis tools for dynamical systems. We study their common constructs and their semantics in order to build an abstract model of these tools. This abstract representation is aimed to facilitate the integration of multiple simulation tools into a common framework.

I. Introduction

Simulation tools for dynamical systems are software applications for modeling, simulation and analysis of electrical, mechanical or thermodynamic systems. Within these tools systems are modeled using block diagrams formed by blocks and connections between blocks. On a block diagram, physical elements (such as mechanical or electrical components) are represented using rectangles or sometimes icons. Connection lines between blocks represent the actual physical connections such as electrical line, mechanical connection, heat flow, etc. Blocks have associated a physical behavior described by differential equations.

In this report we present the main characteristics of a set of general purpose simulation tools. We cover both free and commercial simulation tools, considering also their popularity in industry and academia. The chosen simulation tools are described in what follows.

Scicos (Scilab Connected Object Simulator) [10], [9] is a Scilab [2] package for modeling and simulation of explicit and implicit dynamical systems including both continuous and discrete subsystems. Scilab is a scientific software package for numerical computations providing a powerful open computing environment for engineering and scientific applications. Developed since 1990 by researchers from INRIA (French National Institute for Research in Computer Science and Control), and ENPC (National School of Bridges and Roads), it is now maintained and developed by Scilab Consortium since its creation in May 2003.

Simulink [8], developed by MathWorks, is a commercial tool for modeling, simulating and analyzing multidomain dynamic systems, widely used in control theory and digital signal processing for multidomain simulation and Model-Based Design. It offers tight integration with MATLAB [7] environment and can either drive MATLAB or be scripted from it. MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation.

Dymola (Dynamic Modeling Laboratory) [1] is a tool for modeling and simulation the dynamic behavior and complex interactions between systems of many engineering fields, such as mechanical, electrical, thermodynamic, hydraulic, pneumatic, thermal and control systems. The Dymola environment uses the open Modelica [11] modeling language, an object-oriented, equation based language to model complex physical systems.

SystemBuild [6], a product of National Instruments, provides graphical framework for modeling and simulating complex dynamic systems, as well as specifying and testing control and software algorithms. SystemBuild has a hierarchical block-diagram modeling paradigm designed for development of complex models based on a library of primitive blocks and based on XMath [5], a mathematical analysis, visualization and scripting software environment.

Our goal is to emphasize the common constructs and their semantics. It is not intended to provide a detailed analysis of each of these tools, but rather to be to be used as a starting point for creating an abstract representation of these tools that can be used for integration with graphical modeling languages such as the Systems Modeling Language, an extension of the Unified Modeling Language (SysML/UML).

The rest of the report is organized as follows. In Section II we briefly discuss the graphical user interface of the simulation tools. Section III presents the main constructs of used to build models of systems, while Section IV analysis the simulation tools from the perspective of numerical simulation.

II. Graphical user interface

Commonly, the graphical user interface (GUI) for the simulation tools examined includes at least two windows: a main window where the models are constructed (sometimes called the block diagram window) and a window containing *libraries* of predefined, dedicated blocks, organized based on their functionality or domain application. Some simulation tools call their libraries *palettes* (in the case of Scicos and SystemBuild) or *package* (in the case of Dymola). Common libraries include: *Sources* (for signal generation), *Sinks* (for signal plotting), *Signal Routing*, *Continuous* (for representing continuous systems), *Discrete* (for representing discrete systems), *Mathematical Operation* and *Ports and Subsystems* (to provide support for conditionally executed systems). Some tools include domain specific libraries. A unique characteristic of Dymola is that its libraries are organized in terms of domains (mechanics, thermodynamics, etc.). Figures 1 through 4 show parts of the GUI of the four simulation tools.

III. Constructs for systems modeling

In this section we describe the main constructs used for modeling dynamical systems, which are present in all studied simulation tools. The simulation tools are based on four constructs, organized in a hierarchical structure, namely: *ports*, *links*, *blocks*, *subsystems* and *models*. At the top of the hierarchy we have the model, followed by subsystems and blocks. Subsystems are blocks composed of other blocks. Ports act as interfaces for blocks and subsystems, while links are used to connect blocks and subsystems in a model. In the following we present in more detail the aforementioned structural constructs, together with their associated semantics. Figure 5 shows a typical model structure.

A. Ports

Ports are graphical constructs of a simulation tool that act as interfaces for blocks or subsystems, to allow them to interact with other blocks and subsystems. Depending on the simulation

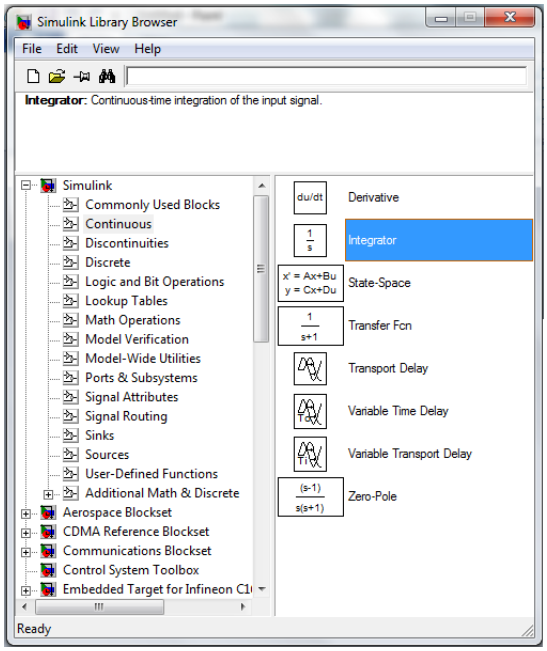


Fig. 1: Simulink library of blocks

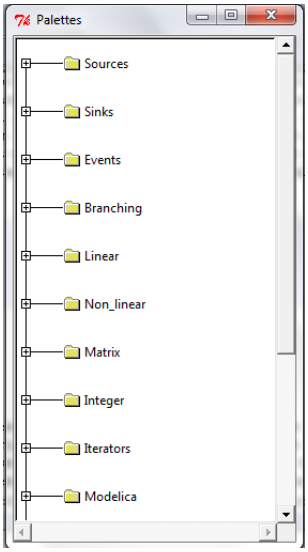


Fig. 2: Scicos library of blocks

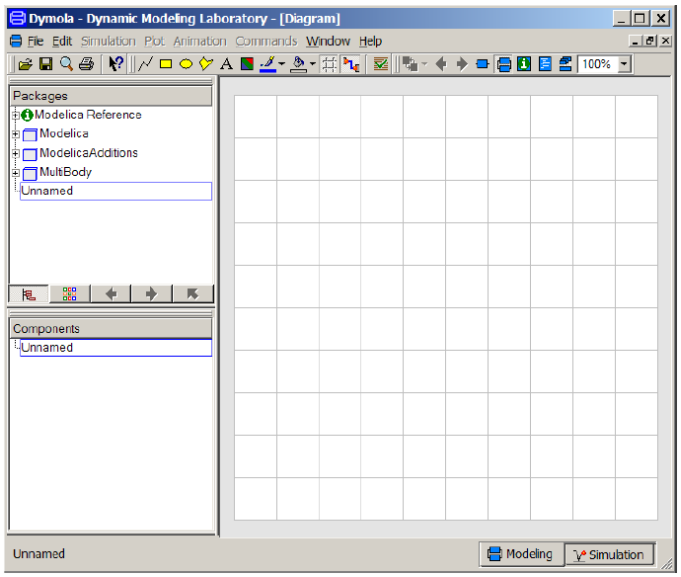


Fig. 3: Dymola library of blocks and main window

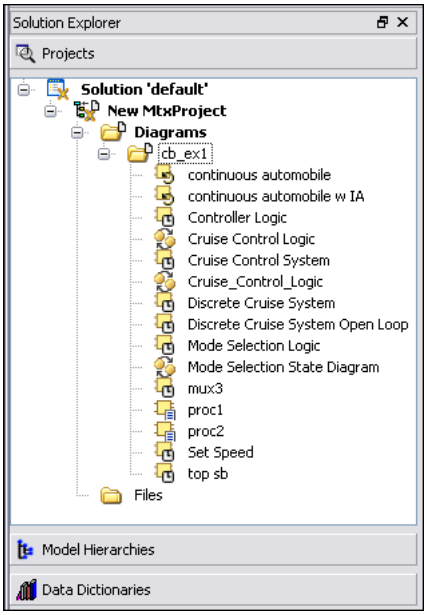


Fig. 4: SystemBuild projects window

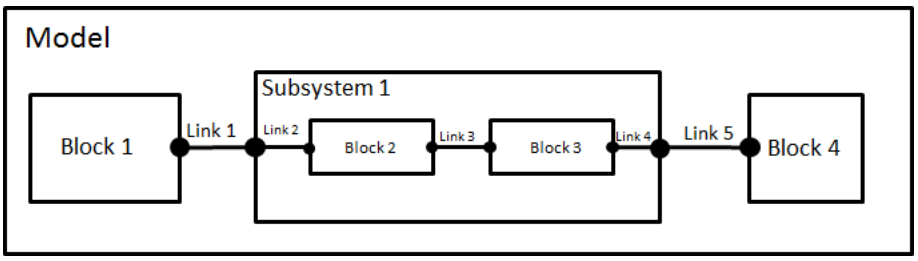


Fig. 5: Model hierarchical structure

tool, they can be input, output, bidirectional (in simulation tools based on Modelica), or control ports. Control ports are block interface through which signals that activate or deactivate a block are received. They can exist as stand-alone constructs, and can be explicitly included in blocks or subsystems to specify interfaces.

Ports have associated variables for communication with other blocks. *Input* and *output variables* are associated with input and output ports, respectively. Variables that satisfy the flow conservation law (Kirchhoff law) (also called *flow variables*) are associated with bidirectional ports. A semantic constraint of Modelica language is that if a port has flow variables, it can not be an input/output port nor any of the other non-flow variables of the port can be defined as inputs/outputs. We will elaborate more on the semantics of bidirectional ports in next subsection, which describes the connections between blocks. Control ports have associated *control variables*, which affects the behavior of a block or subsystem (more on this in the subsections dedicated to blocks and subsystems). Figure 6 shows the graphical representation of different types of ports in Simulink.

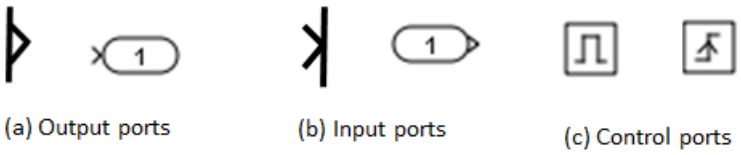


Fig. 6: Different types of ports

B. Links

Links are graphical constructs that connect blocks. Depending on the simulation tool, the links can be directed or undirected. Dymola is among the few simulation tools that support undirected links (since is based on Modelica), supporting differential algebraic equations (DAE)[1] and the concept of flow. The majority of simulation tools supports only directed links, which connect the output of a block to the input of another block. Given a block a with output y_a and a block b, with input u_b (see Figure 7), connecting the output of block a to the input of block b (mathematically) means that

$$u_b = y_a,$$

and that block a determines a value for the input of block b. When connecting outputs to inputs some constraints apply: a block input can only be connected to one output, an output can not be directly connected to the input of the same block; an output can be connected to several inputs.

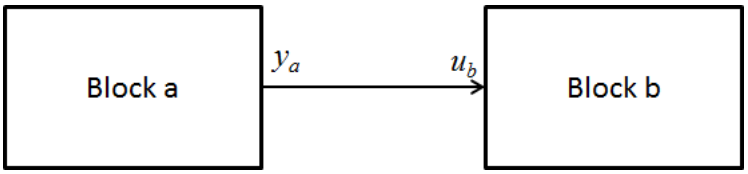


Fig. 7: Connecting inputs and outputs between blocks

As mentioned in the previous section in the case of Modelica language based tools (such as Dymola, OpenModelica) based on the Modelica language, blocks can have bidirectional ports, which can have special type of variables called *flow variables*. The semantics of connecting two ports with flow variables is expressed through the following example. Let a and b be two blocks with two bidirectional ports, with variables u_a, i_a, and u_b, i_b, respectively, where i_a and i_b are flow variables. Connecting the ports of the blocks a and b (Figure 8) has the following semantics:

$$u_a = u_b,$$

[1]Differential algebraic equations are a general form of differential equations for vector-valued functions x, in one independent variable t, $F(\dot{x}(t), x(t), t) = 0$.

$$i_a + i_b = 0.$$

In other words, the non-flow variable must be equal (but without the unidirectional characteristic as in the case of inputs and outputs) and the flow variables must satisfy the flow conservation law, namely the Kirchhoff law.

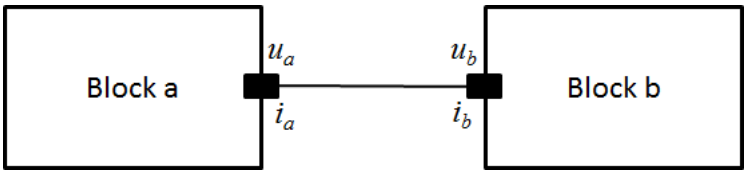


Fig. 8: Connecting inputs and outputs between blocks

C. Blocks

Blocks are graphical constructs with an associated dynamical behavior, that can be viewed as atomic elements on which models are build upon. A block has a vector of *inputs* u, a vector of *outputs* y and a vector of *states* x, which depend on time. The inputs, outputs and states do not have to all be present simultaneously (i.e., there can be blocks with outputs, but no inputs, with inputs, but no outputs, with inputs and outputs, but no states, or with no inputs and outputs, but with states). Often, the inputs and outputs of a simulation tools are called *signals*. Blocks with states x, are also called *blocks with memory*. A block can also have a set of *parameters* θ, that affect the behavior of the block. For example in the case of a state-space representation of a linear, time-invariant system given by

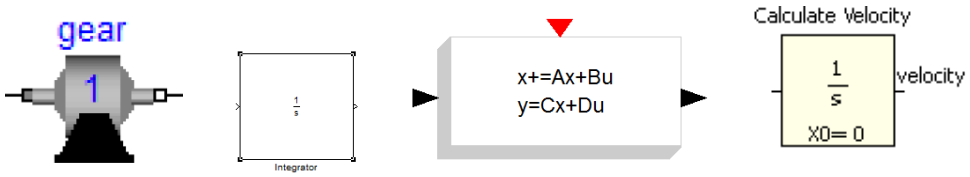
$$\dot{x}(t) = Ax(t) + Bu(t), \; x(0) = x_0, \tag{1}$$

$$y(t) = Cx(t) + Du(t), \tag{2}$$

the parameter θ refers to the matrices A, B, C and D and to the initial condition x_0.

Blocks have *ports* (also called *pins* or *connectors* by some applications), through which signals are exchanged with other blocks; signals which can be continuous or discrete, vector valued or scalar.

The state vector can contain continuous (x_c) and discrete (x_d) states, $x' = [x_c', x_d']$, where by x' we denote the transpose of the column vector x. The behavior of a block can be mathematically



(a) (b) (c) (d)

Fig. 9: (a) Dymola block; (b) Simulink block; (c) Scicos block; (d) SystemBuild block.

described as a hybrid system, that is a combination of continuous and discrete ordinary differential equations (ODE):

$$y(t) = h(t, x_c(t), x_{d_{t_k}}, u(t), \theta), \;\; x(0) = x_0, \tag{3}$$

$$\dot{x}_c(t) = f(t, x_c(t), x_{d_{t_k}}, u(t), \theta), \tag{4}$$

$$x_{d_{t_{k+1}}} = g(t, x_c(t), x_{d_{t_k}}, u(t), \theta), \tag{5}$$

where $x' = [x'_c, x'_d]$ is the state vector containing both the continuous and discrete components, and x_0 is the initial condition of the state vector. Note that t represents continuous time, while t_k represents discrete time. From equations (3) and (4) we note that $y(t)$ and $x_c(t)$ change continuously with time t, while the discrete component of the state $x_{d_{t_k}}$ changes only at discrete times instants (Figure 10 shows an example of the state evolution in the case of a hybrid system, where the continuous and discrete components of the state are scalars). However, no simulation tools can perform continuous updates, and therefore $x(t)$ and consequently $y(t)$ must be approximated through numerical integration. This involves computing the state at a set of discrete time instants, depending on the integration method. Also note that x_d, the discrete component of the state, is assumed **constant** between t_k and t_{k+1}, for any k. The mathematical model previously described is supported by all simulation tools. Some simulation tools (such as Dymola or SystemBuild) support not only ODEs, but also DAEs. In this case, (3) and (4) can be replaced by

$$y(t) = h(t, \dot{x}_c(t), x_c(t), x_{d_{t_k}}, u(t), \theta), \;\; x(0) = x_0, \tag{6}$$

$$0 = f(t, \dot{x}_c(t), x(t), x_{d_{t_k}}, u(t), \theta). \tag{7}$$

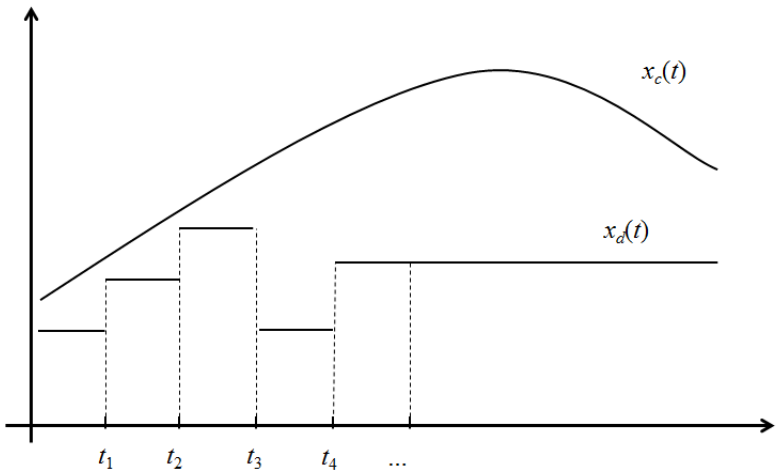


Fig. 10: Example of the state evolution in the case of a hybrid system

Some simulation tools go even further, by accommodating over-determined differential algebraic equations (ODAE). In such case, in addition to (6) and (7), a new equation is added

$$0 = f_c(t, \dot{x}_c(t), x_c(t), x_{d_{t_k}}, u(t), \theta), \tag{8}$$

where f_c is a function reflecting additional constraints on the state and input of the system.

It is important to note that the behavior of a block can be viewed from two perspectives: the *ideal* (mathematical) behavior and *simulated* behavior. Each perspective has associated a time, a mathematical time and a simulation time, respectively which are often different. From the mathematical perspective, the state and output changes are assumed to occur **instantly**. However, from the perspective of the simulator (solver), the changes in the outputs and state happen as a result of numerical calculations that will **not** take place instantly, since they may involve complex numerical computations.

Many simulation tools support hybrid systems by combining continuous and discrete blocks. In the case of a continuous block, its (mathematical) behavior is given by

$$y(t) = h(t, x(t), u(t), \theta), \;\; x(0) = x_0, \tag{9}$$

$$\dot{x}(t) = f(t, x(t), u(t), \theta), \tag{10}$$

while in the case of a discrete block, the mathematical behavior is given by

$$y_{t_k} = h(t, x_{t_k}, u_{t_k}, \theta), \;\; x(0) = x_0, \tag{11}$$

$$x_{t_{k+1}} = f(t, x_{t_k}, u_{t_k}, \theta), \tag{12}$$

where t_k is the discrete time. The discrete time instants are determined by the *sample time* T and the *offset* Δ, such that $t_k = k \cdot T + \Delta$, where k is a positive integer. As mentioned earlier, the output of a discrete block is assumed to be constant between two discrete time instants. This suggest that a zero-order hold (ZOH)[2] is implicitly assumed as part of the discrete block and therefore, the output of a discrete block is a piecewise continuous function.

Although the input of a discrete block can be continuous, the variations of the input between two sample-times are ignored and only the value of the input $u(t)$ at the time instant t_k is used to compute $x_{t_{k+1}}$.

Each block has a set of graphical interface parameters such as: name, description, port locations, orientation, foreground/background color, position of the block in the block diagram window, position of the block name, font and font size. In addition, all simulation tools support operations on blocks, such as creating a new block, copying a block, moving a block, resizing a block or deleting a block.

Besides blocks with memory (internal state), simulation tools have a variety of blocks without memory. These blocks perform mathematical operations (such as summation, multiplications, dot-products, etc.), generate, plot or route signals or perform logic operations. Additional information about memoryless blocks can be found in [6], [8], [1].

D. Subsystem

A subsystem (also called a superblock by some simulation tools) is a graphical construct used to simplify the representation of a group of blocks. Subsystems have ports, which are internally connected through links to the variables (inputs, outputs, flow variables) of some of their constituent blocks. They can be composed of both continuous and discrete blocks, and therefore can represent hybrid systems. The behavior of the subsystem is dictated by the behavior of the constituent blocks.

[2]The zero-order hold is a mathematical model that describes the effect of converting a discrete-time signal to a continuous-time signal by holding each sample value for one sample interval

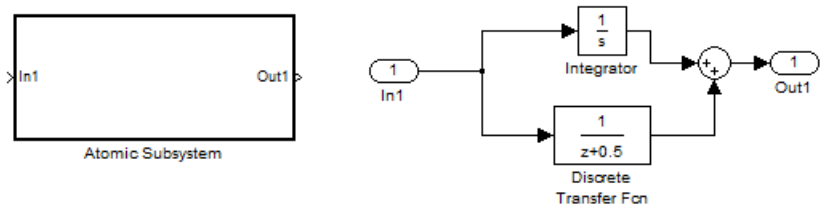
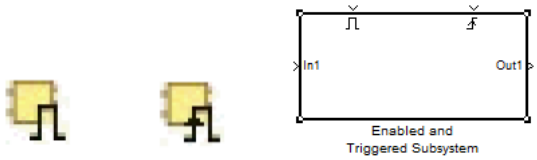


Fig. 11: Example of a subsystem in Simulink

1) Special type of subsystems: In the previous sections, the behavior of hybrid blocks was described, where the blocks were assumed to process data without interruption (at every time sample) from the beginning to the end of the simulation. There exist however subsystems that are not constantly active throughout the simulation, but rather become active as a result of an event. There are three types of such subsystems: *triggered* subsystems, *enabled* subsystems, and *triggered and enabled* subsystems. These subsystems, also called *conditionally executed subsystems*, receive control signals (through control ports) that determine when they execute. Simulink, Scicos and System Build simulation tools provide "ready to use" triggered and enabled subsystems. Dymola supports user-defined triggered and enabled blocks/subsystems.



(a) (b)

Fig. 12: (a) Enabled and Triggered symbols for systems in SystemBuild; (b) Triggered and Enabled subsystem in Simulink.

Triggered subsystems execute when an event is detected. The event consists of the control signal crossing zero in a positive direction, negative direction or both. Both output and states are updated when an event is detected, while in between events they are held constant. Due to the event based nature of the execution, the triggered subsystems can only be discrete (or with

sample time inherited from the output of another system).

Enabled subsystems execute (the states and outputs are updated) as long the control signal has a positive value. Note that although a systems may be disabled, its output can be made available. When a subsystem is enabled, it may be set to re-initialize its state or the keep its state at the previous value. Enabled subsystems can be both discrete and continuous in nature.

Subsystems can be both enabled and triggers by having two control signals. In such case triggering events determine the execution of the subsystem only if the subsystem is enabled, that is, the enabling signal is positive at the moment of the triggering event.

E. Model

A model of a system is represented as a block diagram. It is created using using subsystem/block references, (representing instances/usages of existing subsystems/blocks) connected by links. Semantically speaking, block diagrams define time-based relationships between variables of composing blocks (such as inputs, outputs, states). It has no inputs or outputs and it has two important parameters related to simulation: the *simulation time* and the *solver type*. The simulation time sets the simulation duration, and has an initial time and a final time (that can be infinity). The solver type determines the algorithm used for numerical integration. Further discussion about this is provided in the next section.

IV. Model simulation

As mentioned earlier, a model has an ideal, mathematical behavior and a simulated behavior. In this section we discuss the simulated behavior, i.e. what actually happens during simulation. The simulation is based on a numerical solver that performs numerical integration. Before a solver can be used for numerical simulations, the graphical model is translated into an appropriate representation, suitable for the numerical simulation, Figure 13 shows the two stages of the model simulation.

A. Model translation

The translation stage is composed of several steps. First the parameters of the blocks are evaluated and the resulting numerical values are used in the actual blocks. In the second step the hierarchial model is flatted. In this step all subsystems with the exception of the conditionally

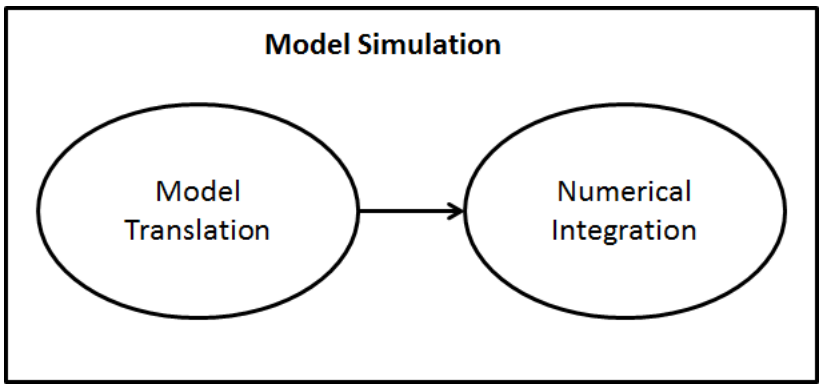


Fig. 13: Model simulation

executed ones, such as triggered or enabled subsystems are replaced by the blocks they represent. The conditionally executed blocks can not be replaced with their content, since their behavior is determined at the time of execution.

The next step is the scheduling of the block execution, where the order in which blocks are executed is determined. The scheduling algorithm uses computational attributes (such as the type of the block or the sample rate) to establish an execution priority and has procedures to detect and avoid *algebraic loops*[3]. Some simulation tools allow users to set the priority of blocks, taken into account during the execution process.

Finally, the model is checked for correctness, for example the vector length of the outputs of each block is checked to ensure that is of the same length as expected by the input of the blocks it drives, the expected data type of the inputs are met. The steps involved in model translation are summarized in Figure 14.

B. Numerical integration

After the previous steps, the transformed model is passed to the solver for numerical integration. In the numerical integration stage the key task performed is key task of computing (approximating) the state derivatives of the blocks (in the case where they have a continuous component). Several steps are involved in numerical integration.

[3]An algebraic loop in a model is a loop consisting of elements without ”memory like” functions. To calculate the variables in this loop, the variable values themselves are needed.

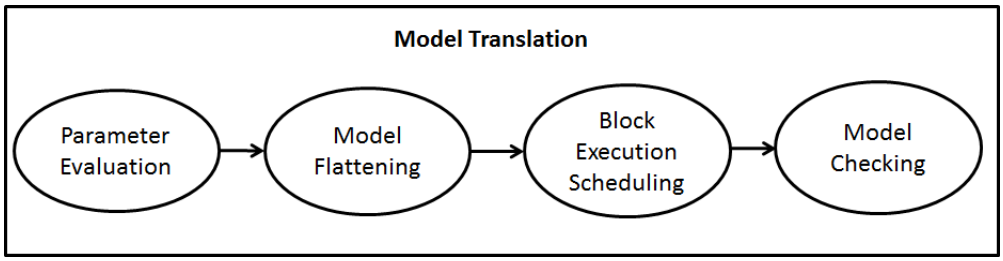


Fig. 14: Model translation

First, the outputs of each block are calculated according to the order determined by the scheduling algorithm, as functions of the current input and state values. Next, state derivatives are computed, based on the current time, state and input values. Finally, state derivatives are used to compute the new state vector at the next time point, and to update the sampled blocks. Figure 15 depicts the steps of the the numerical integration stage.

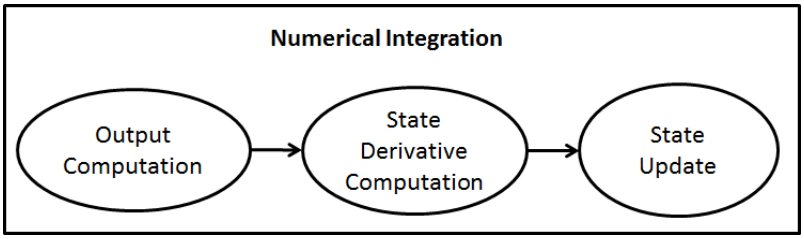


Fig. 15: Numerical integration

The simulation tools offer a variety of solvers to perform the numerical integration whose efficiency depends on the type of ODE they solve. The solvers must ensure that the outputs of discrete blocks are updated at time instants synchronized with the sample-period of the blocks, i.e. the numerical integration must be stopped for the update of the discrete states and then re-initialized. There are two main categories of solver: *variable-step* and *fixed-step* solvers. Variable-step solvers can dynamically change the time intervals over which numerical integration takes place, depending on the behavior of the block. The adjustment depends on the approximation error and on events that determine sudden changes in the behavior of the blocks. These types of algorithms are suitable for hybrid systems, where sudden changes in behavior are not unusual. Different variable-step solvers are based on different forms of the Runge-Kutta,

Adams-Bashforth-Moulton, or Rosenbrock formulas [3]. Fixed-step solvers use the same sample-period during simulation. They can not detect sudden changes in the behavior of the blocks. They are typically based on Dormand-Price, Runge-Kutta, Bogacki-Shampine, or Euler formulas [3].

It may be the case that the inputs of a block have different sample times. In such cases, if the smallest sample-time is a common divisor of the rest of the sample-times, it is used as sample time for the block. If not, the greatest common divisor is used, and if the computation of this quantity is not possible, the block is considered continuous.

To support numerical integration, blocks are endowed with a set of functions that are used at different steps in the numerical integration process. these functions can be summarized as follows:

1) *Initialization* is called once at the start of the simulation. It determines the initial conditions and sampling-times.
2) *State Derivative Computation* computes the state derivative $\dot{x}_c$.
3) *State Update* computes the current state of the block in the case of continuous blocks, and the next state in the case of of discrete block.
4) *Output update* computes the output vector y.
5) *Finalize* performs operations related to the termination of the simulation.

In some simulation tools, blocks have explicit functions for computing the Jacobian of the system (useful when solving implicit system) and for monitoring the occurrence of state events.

C. Comments on the numerical integration of a particular set of systems

In this subsection we discuss stiff systems, systems with algebraic loops and systems with state events, from the numerical integration point of view.

1) Stiff systems: Although a default solver is assigned at the beginning of a simulation execution, users can select an integration algorithm, better suited to the dynamics of the system. Great care must be shown when simulating *stiff systems*, which present both slow and fast dynamics that "conventional algorithms" are not able to capture. These systems are prone to numerical instability unless step-size is carefully chosen. Solvers for stiff equations apply the iterative Newton-Raphson method at each time step, which requires the evaluation of the Jacobian of the system and consequently can be very expensive numerically. More sophisticated methods automatically switch between stiff and non-stiff methods to achieve good performance in both

cases. Common solvers for stiff-equations are typically based on Rosenbrock methods, Bulirsch-Stoer-Bader-Deufhard semi-implicit methods, implicit Runge-Kutta methods and backward differential formulas methods [4].

2) Systems with algebraic loops: As mentioned earlier, during the simulation process, great care must be taken when dealing with algebraic loops. Algebraic loops happens when an input port of a block with *direct feedthrough*[4] is driven by the output of the same block, either directly or through other blocks. They also appear when dealing with DAE, which model implicit systems[5] of the form

$$f(\dot{x}, x, t) = 0,$$

with initial conditions $\dot{x}(0) = \dot{x}_0, x(0) = x_0$.

There are two approaches in dealing with algebraic loops. If an explicit method is used (that is a method for solving ODE), then a delay is introduced in order to break the loop. Unfortunately, this approach does not always work, since the delay may make the system unstable. Another approach is to use a solver specifically designed for solving implicit systems. The solver for implicit systems works as follows. First, at each time instant, the differential $\dot{x}$ is approximated by a backward differentiation formula. Next, the approximation is substituted in $f(\dot{x}, x, t)$, resulting in a nonlinear algebraic equation. Finally, the nonlinear equation is solved by a Newton-Raphson iterative method.

Remark 4.1: Note that there are cases where the presence of algebraic loops does not harm the numerical simulation. One such example is loops involving triggered systems. At the time of the trigger, the inputs of the system are assumed sufficiently stable to be used to update the output.

3) State events: As described previously, a block can represent a hybrid system, where the state is determined by combinations of discrete and continuous dynamics. The evolution of the state of hybrid systems often suffers sudden changes that are represented as discontinuities in the state evolution (also called *state events*). From the point of view of numerical integration, when

[4]Blocks with input ports with direct feedthrough are blocks for which the outputs cannot be computed without knowing the values of the signals entering the blocks at the input ports.

[5]An implicit system is a system where the state differential can not be explicitly represented using conventional mathematical functions, but is a solution of some equation for the state derivative in terms of the state and input.

such discontinuities appear, the integration process has to be stopped, the state re-initialized (with the value of the state after the sudden change) and then integration is restarted. Therefore, it is very important to be able to detect the time instants when discontinuities occur. Simulation tools use zero-crossing to detect discontinuities in the signals.

V. Conclusions

In this report we present an overview of the main characteristics of simulation tools for dynamical systems. We focus on the main concepts of the hierarchical structure of the simulation tools and on the steps involved in a simulation of a model. The goal of this report is to lay the ground for creating a abstract representation of the simulation tools which can be used within a framework for system modeling, analysis and simulation, such as SysML/UML.

References

[1] Dynasim AB. DYMOLA, Dynamic Modeling Laboratory - User's manual. *Dynasim AB*, 2004.

[2] S. Annigeri. Scilab - A Hands on Introduction. 2004.

[3] E. Hairer, S.P. Norsett, and G. Wanner. *Solving ordinary differential equations I: Nonstiff problems, second edition*. Springer Verlag, Berlin, 1993.

[4] E. Hairer, S.P. Norsett, and G. Wanner. *Solving ordinary differential equations II: Stiff and differential-algebraic problems, second edition*. Springer Verlag, Berlin, 1996.

[5] National Instruments. Xmath. *www.ni.com/matrixx/xmath*.

[6] National Instruments. NI MATRIX, SystemBuild User Guide. *National Instruments*, 2007.

[7] MathWorks. MATLAB 7 - Getting Started Guide. *www.mathworks.com*, 2011.

[8] MathWorks. SIMULINK User's guide. *www.mathworks.com*, 2011.

[9] R. Nikoukhah. SCICOS: a dynamic systems modeler and simulator. *INRIA*.

[10] R. Nikoukhah and S. Steer. SCICOS - A Dynamic System Builder and Simulator Users Guide. *INRIA*.

[11] M. Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.