# Draft for a simulation runtime in OpenModelica

Willi Braun

March 16, 2016

This draft describes the current state of the OpenModelica run-time and the requirements of improvements for the current c-run-time.
*Keywords: Simulation, c-run-time, solver, OpenModelica*

## 1. Introduction

This draft summarizes all tasks which are needed to be done in the simulation run-time for the modeling and simulation environment OpenModelica. The simulation run-time is a library that is used with the generated model code to be able to execute a simulation of the library. For the OMC exist several run-time libraries and therefore the generated model code is different each library.

- c-runtime
- cpp-runtime
- adevs-runtime
- csharp-runtime

This draft is concerning the c-run-time. This library has grown over the last few years. The historical growth is one reason for a weak structure. Therefor the c-run-time will be reimplemented during the OpenModelica Developer week in November 2011.

Further, this draft wants to setup a clear structure for the current tasks, also with respect to future tasks.

The c-run-time is used for the following jobs:

- Simulation of a generated model.
- FMI Export uses functions of the simulation run-time[**openmodelica.org:doc-extra:fmi1.**
- Interactive Simulation

- ...

Therefore, section 2 will state and describe all requirements on a simulation run-time. In the next section 3 the goals for the redesign will be summarized.

# 2. Requirements

The main purpose of the simulation run-time is to calculate the simulation of a translated model. In addition, there are some further tasks that are solved by using functions of the simulation run-time (e.g. FMI, interactive simulation, etc.). The calculations that are needed to be done during the simulation process are specified in A.

to be continued ...

# 3. Goals for Redesign

- Redesigning the usage of variables in the run-time. Separating static and dynamic variables (see 3.1).
- The reduction of the size of the generated model code and the size of every single function (see 3.2).
- Increasing the possibilities of debugging the solving process and provide better information for the users (see 3.3).
- Fixing some issues in the current simulation run-time (see 3.5).
- Preparing the run-time for future tasks (see 3.4).

to be continued ...

## 3.1. Separate static and dynamic variables

In the current c-run-time we have a `struct sim_DATA` that contains almost all data of a simulation model(see 1). This should be separated in at least two parts. One part for dynamic variables and one part for static variables. Further, we should create a new structure for all simulation variables, that contain all information about it (e.g. current value, start value, fixed-value, pre-value, max-min-values, etc.).

Listing 1: struct of the current sim_DATA from the current c_runtim

```
sim_DATA { /* this is the data structure for saving
  important data for this simulation.
  Each generated function has a DATA
  parameter which contains the data.
  An object for the data can be created using
  initializeDataStruc() function*/
```

```
double* states;
double* statesDerivatives;
double* algebraics;
double* parameters;
double* inputVars;
double* outputVars;
double* helpVars, *helpVars_saved;
double* initialResiduals;
double* jacobianVars;
/* True if the variable should be filtered */
modelica_boolean* statesFilterOutput;
modelica_boolean* statesDerivativesFilterOutput;
modelica_boolean* algebraicsFilterOutput;
modelica_boolean* aliasFilterOutput;
/* Old values used for extrapolation */
double* states_old, *states_old2, *states_saved,
   *states_start;
double* statesDerivatives_old, *
   statesDerivatives_old2,
*statesDerivatives_saved, *
   statesDerivatives_start;
 double* algebraics_old, *algebraics_old2,
 *algebraics_saved,*algebraics_start;
 double oldTime,oldTime2; double current_stepsize
    ;
/* Backup derivative for dassl */
double* statesDerivativesBackup;
double* statesBackup;
char* initFixed; /* Fixed attribute for all
   variables and parameters */
char* var_attr; /* Type attribute for all
   variables and parameters */
int init; /* =1 during initialization, 0
   otherwise. */
int terminal; /* =1 at the end of the simulation,
    0 otherwise. */
void** extObjs; /* External objects */
/* nStatesDerivatives == states */
fortran_integer nStates,nAlgebraic,nParameters;
long nInputVars,nOutputVars,nFunctions,nEquations
   ,nProfileBlocks;
fortran_integer nZeroCrossing/*NG*/;
long nJacobianvars;
long nRelations/*NREL*/;
```

```
long nInitialResiduals/*NR*/;
long nHelpVars/* NHELP */;
/* extern char init_fixed[]; */
DATA_STRING stringVariables;
DATA_INT intVariables;
DATA_BOOL boolVariables;
DATA_REAL_ALIAS* realAlias;
long nAlias;
const char* modelName; /* For error messages */
const char* modelFilePrefix; /* For filenames,
   input/output */
/* to check if the model_init.xml match the model
    */
const char* modelGUID;
const struct omc_varInfo* statesNames;
const struct omc_varInfo* stateDerivativesNames;
const struct omc_varInfo* algebraicsNames;
const struct omc_varInfo* parametersNames;
const struct omc_varInfo* alias_names;
const struct omc_varInfo* int_alg_names;
const struct omc_varInfo* int_param_names;
const struct omc_varInfo* int_alias_names;
const struct omc_varInfo* bool_alg_names;
const struct omc_varInfo* bool_param_names;
const struct omc_varInfo* bool_alias_names;
const struct omc_varInfo* string_alg_names;
const struct omc_varInfo* string_param_names;
const struct omc_varInfo* string_alias_names;
const struct omc_varInfo* inputNames;
const struct omc_varInfo* outputNames;
const struct omc_varInfo* jacobian_names;
const struct omc_functionInfo* functionNames;
const struct omc_equationInfo* equationInfo;
const int* equationInfo_reverse_prof_index;
double startTime; /* the start time of the
   simulation */
double timeValue; /* the time for the simulation
   */
/* used in some generated function */
/* this is not changed by initializeDataStruc */
/* The last time value that has been emitted. */
double lastEmittedTime;
/* when != 0 force emit, set e.g.
by newTime for equidistant output signal. */
```

```
  int forceEmit;
  /* An array containing the initial data of
  samples used in the sim */
  sample_raw_time* rawSampleExps;
  long nRawSamples;
  // The queue of sample time events to be
     processed.
  sample_time* sampleTimes; /* Warning: Not
     implemented yet!? */
  long curSampleTimeIx;
  long nSampleTimes;
} DATA;
```

All the variables in the listing 1 and all other global variables in the `simulation_*.cpp` files should categorised and separated in of the categories in the figure 1. Therefore could be create a new file `simulation_data.h` that contains all types.

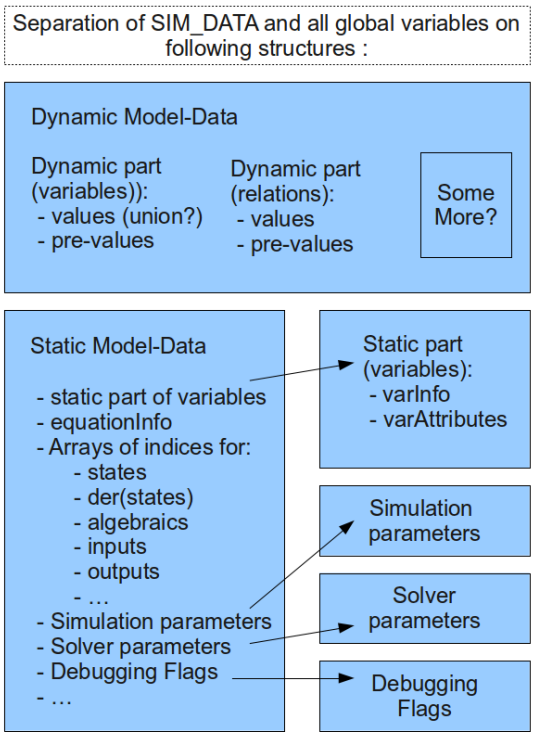


Figure 1: Schematic separation of variables used in c-run-time.

**Dynamic part**

**dynamicVar** contains all values of the variables that might be changed during the simulation. Hence we need to interpolate some values during the simulation, we need also some old values stored here. For handling events proper we need also save for every variable a pre-value.

**dynamicHelp** in the current implementation we save relations, ZeroCrossings in `helpvars`. So we should collect them all in central structure that is organized similar to all other variables.

**More?**

**Static part:**

**staicVar** contains all static information about a variable of the simulated model. The static information consists of varInfo and varAttributes.

**equationInfo** a structure defined in `simulation_varinfo.h`.

**lookupVars** contains the indices for the categorised variables (states, der(states), algebraics, etc.) of the dynamicVar array.

**SimData** contains all parameters to control the simulation process.

**SolverData** contains all parameters to setup the used solver.

**debugFlags** contains the flags for debugging the simulation.

**More?**

### 3.1.1. implementation issues

unions, ring buffer ??? to be continued in more detail ...

## 3.2. Reducing the size of the generated model code and generated functions

Currently, we generate almost every equation one time for "functionODE" and "functionAlgebraics" and a second time for "functionDAE". This could be avoided by generating every equation or equation-block just one time and creating one function with a parameter for the context in which this function is going to be executed. There are two different context where an equation can be executed: (1) during a continuous integration, where changing of relations and discrete variables is forbidden. (2) during an event, where the relations and discrete variables can be changed. Thus the size of the generated model will be reduced all in all and also the size of generated functions. Therefore this will result in a huge speed up of the compile time of the "gcc" for bigger models.

to be continued in more detail ...

## 3.3. Increasing the possibilities of debugging

This subsection addresses the following two issues:

- Providing better information about the solving process to the users.
- Creating possibilities for debugging the solving process from the point of view of a developer [**openmodelica.org:pop:sims:2007**].

to be continued ...

## 3.4. The Requirements of the run-time in the future

As some requirements in the future we could address the following issues:

- Handling appropriate dynamic state selection.
- Handling varying structure problems.
- Preparing the run-time for parallelization of the simulation process.
- ...

This topics should be discussed further at the developer week.
to be continued ...

## 3.5. Current unsolved issues in the run-time

In the current run-time there are also some issues, which aren't handled appropriate right now or missing at all. They should also be addressed during this redesign.

- Algorithm sections should be initialized correctly. Therefore the start values of all variables are needed.
- Some Modelica statements are not supported appropriately. Following a first small list.

  **parameters** There is no separation of primary and secondary parameters. So that one could think to change secondary parameters in the init-file.

  **input varibales** There is no possibility to provide input variables by files.

  **Special Statements** `assert()` and `terminate()` are not handled appropriately.

  **initial()** In the current run-time the expression `when not initial() then ...` is not handled appropriately.

  ...
- ...

to be continued ...

# 4. Overview of current c-runtime

to be continued ...

# A. Calculations during the simulation

Flat hybrid DAEs could represent continuous-time behavior and discrete-time behavior. This is done mathematically by the equation (1).

$$F(\underline{\dot{x}}(t), \underline{x}(t), \underline{u}(t), \underline{y}(t), \underline{q}(t_e), \underline{q_{pre}}(t_e), \underline{c}(t_e), \underline{p_P}, \underline{p_S}, \underline{s_0}, t) = 0 \tag{1}$$

This implicit equation (1) is transformed to the explicit representation of equation (2) by block-lower-triangular transformation.

$$\begin{pmatrix} \underline{\dot{x}}(t) \\ \underline{y}(t) \\ \underline{q}(t_e) \end{pmatrix} = \begin{pmatrix} \underline{f_s}(\underline{x}(t), \underline{u}(t), \underline{q_{pre}}(t_e), \underline{c}(t_e), \underline{p_P}, \underline{p_S}, \underline{s_0}, t) \\ \underline{f_a}(\underline{x}(t), \underline{u}(t), \underline{q_{pre}}(t_e), \underline{c}(t_e), \underline{p_P}, \underline{p_S}, \underline{s_0}, t) \\ \underline{f_q}(\underline{x}(t), \underline{u}(t), \underline{p}, \underline{q_{pre}}(t_e), \underline{c}(t_e), \underline{p_P}, \underline{p_S}, \underline{s_0}, t) \end{pmatrix} \tag{2}$$

From this explicit form all necessary calculations can be educed for the simulation of the model. This is done by formulating the continuous-time part, followed by the discrete-time part. Below are summarized the notation used in the following equations:

- $\underline{\dot{x}}(t)$, the differentiated vector of state variables of the model.
- $\underline{x}(t)$, the vector of state variables of the model, i.e., variables of type `Real` that also appear differentiated, meaning that `der()` is applied to them somewhere in the model.
- $\underline{u}(t)$, a vector of input variables, i.e., not dependent on other variables, of type `Real`. They also belong to the set of algebraic variables since they do not appear differentiated.
- $\underline{y}(t)$, a vector of Modelica variables of type `Real` which do not fall into any other category.
- $\underline{q}(t_e)$, a vector of discrete-time Modelica variables of type discrete `Real`, `Boolean`, `Integer` or `String`. These variables change their value only at event instants, i.e., at points $t_e$ in time.
- $\underline{q_{pre}}(t_e)$, the values of $q$ immediately before the current event occurred, i.e., at time $t_e$.
- $\underline{c}(t_e)$, a vector containing all `Boolean` condition expressions evaluated at the most recent event at time $t_e$. This includes conditions from all `if`-equations and `if`-statements and `if`-expressions from the original model as well as those generated during the conversion of `when`-equations and `when`-statements.

- $\underline{p_P} = p1, p2, \ldots$, a vector containing the Modelica variables declared as **primary parameter** i.e., variables without any time dependency and without a dependence on other parameters.
- $\underline{p_S} = p1, p2, \ldots$, a vector containing the Modelica variables declared as **secondary parameter** i.e., variables without any time dependency, but with a dependence on other parameters.
- $\underline{s_0} = s1, s2, \ldots$ all start values in the model.
- $t$, the Modelica variable time, the independent variable of type `Real` implicitly occurring in all Modelica models.

## A.1. Continuous Behavior

The continuous behavior of hybrid DAEs can be formulated with the following equations (3).

$$\begin{pmatrix} \underline{\dot{x}}(t) \\ \underline{y}(t) \end{pmatrix} = \begin{pmatrix} \underline{f_s}(\underline{x}(t), \underline{u}(t), \underline{q_{pre}}(t_e), \underline{c}(t_e), \underline{p_P}, \underline{p_S}, \underline{s_0}, t) \\ \underline{f_a}(\underline{x}(t), \underline{u}(t), \underline{q_{pre}}(t_e), \underline{c}(t_e), \underline{p_P}, \underline{p_S}, \underline{s_0}, t) \end{pmatrix} \tag{3}$$

The states $\underline{x}(t)$ are determined by an integration method, so that they are assumed to be known as the vectors $\underline{u}(t)$ and $\underline{p_{[P|S]}}$. For discrete variables and the condition expressions $t_e$ is used instead of $t$ to indicate that such variables may only change values at event points of time and are kept constant in the continuous parts of the simulation.

Imported is also that all conditions $\underline{c}(t_e)$ are kept on there current value for the whole continuous step. If the continuous step cause

## A.2. Discrete Behavior

The discrete behavior is controlled by events. Events are triggered by the event conditions $\underline{c}(t_e)$ and can appear at any time as well as influence the system several times.

An event occurs when a condition of $\underline{c}(t_e)$ changes it's value at time $t_e$ from `false` to `true` or the other way around. This occurs if and only if for a sufficient small $\epsilon$, one condition in $\underline{c}(t_e)$ is changed, for e.g. $\underline{c}(t_e - \epsilon)$ is `false` and for $\underline{c}(t_e + \epsilon)$ is `true`. When an event occurs all caused changes in the system can be carried out. In addition, the entire system must be determined by the function (2) to guarantee the synchronism of all equations. However, it is not enough to determine only the discrete variables by the function $\underline{f_q}$ at this point.

The problem to be solved here is the most accurate determination of the event time $t_e$. For this conditions $\underline{c}(t_e)$ can be divided into three groups.

1. Conditions $\underline{c_k}(t_e)$, which also depend on continuous variables.
2. Conditions $\underline{c_d}(t_e)$, that only depend on discrete variables.

3. Conditions $\underline{c_{\text{noEvent}}}(t)$, are evaluated without resulting in an event.

If the `smooth` operator applies to a condition in $\underline{c}(t_e)$ this condition can be categorized depending on the order of the integration method by 1. or 3., respectively.

The second and third group of conditions are easy to handle, because if a condition in $\underline{c}(t_e)$ only depends on discrete variables, then they could only change at events and the conditions $\underline{c_d}(t_e)$ must be tested only at events. The conditions $\underline{c_{\text{noEvent}}}(t)$ result logically in no events. Thus, the equations which depend on conditions $\underline{c_{\text{noEvent}}}(t)$, will be determined during the continuous integration at the output points. Hence the variables that are determined by the function (4) should be treated appropriately, like algebraic variables.

$$\underline{q_{\text{noEvent}}}(t) := g(\underline{x}(t_e), \underline{u}(t_e), \underline{q_{pre}}(t_e), \underline{c_{\text{noEvent}}}(t_e), \underline{p}, t) \tag{4}$$

What remains is the group of conditions, that lead to state events. For this group of conditions a time-consuming search has to be performed. These conditions have to be checked during the continuous solution as described in the next section.

Additional, discontinuous changes can caused by the `reinit()` operator to the continuous states $\underline{x}(t)$. As for purely discrete conditions $\underline{c_d}(t_e)$ the `reinit()` operator can only be activated at event times $t_e$ . This new allocation to the states could use the function (5).

$$\underline{x(t_e)} := \underline{f_x}(\underline{x}(t), \underline{\dot{x}}(t), \underline{u}(t), \underline{q}(t_e), \underline{q_{pre}}(t_e), \underline{c}(t_e), \underline{p}, t) \tag{5}$$

## A.3. Run-time Algorithm

A general approach for the simulation of hybrid systems has been developed by Cellier ( cf. [**openmodelica.org:doc-extra:cellier1979**]). In the following a schematic Flowchart (see fig. 2) for the simulation is shown and each step is described.

First of all the simulation must be initialized consistently. For that the initial values are found with a simplex optimization method in OpenModelica (cf. [**openmodelica.org:bachmann:modelica:2006**]). By use of the initial conditions the initial values for the entire system can be determined with the function (2). This will also execute all initial events at time $t_0$.

After the initialization the main simulation loop starts with the continuous integration step that calculates the states $\underline{x}(t_{i+1})$. With the new values of $\underline{x}(t_{i+1})$, the functions $\underline{f_s}$ and $\underline{f_a}$ can be evaluated. Thus, the entire continuous system is determined.

The continuous integration step is accepted if none of the Zero-Crossing functions has a zero-crossing, i.e in $\underline{c}(t_{i+1})$ no value has changed compared to $\underline{c}(t_i)$. If no event has occurred the values can be saved and the next step can be performed.

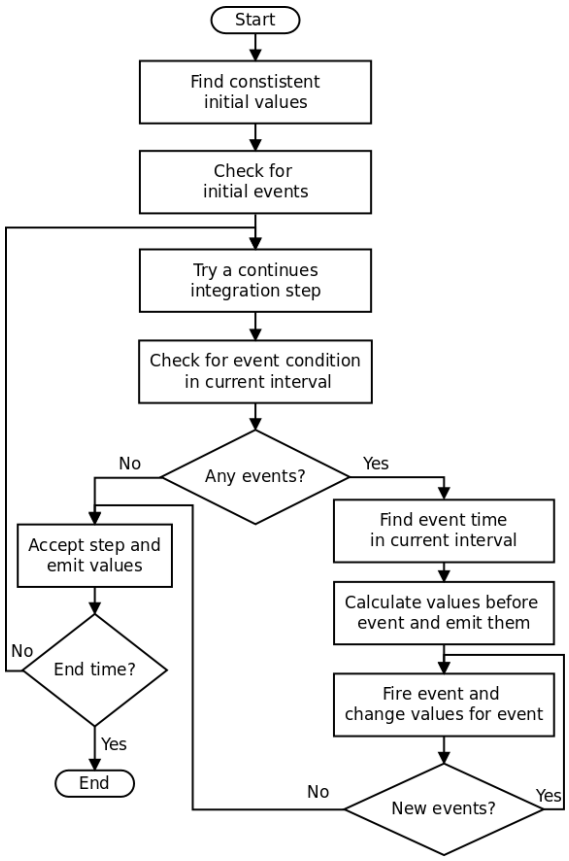


Figure 2: Schematic Flowchart for simulation hybrid models.

However, if a value of $\underline{c}(t_{i+1})$ changes, an event occurred within the interval $t_i$ and $t_{i+1}$. Then the exact time $t_e$ has to be detected. Therefore a root finding method is performed on the Zero-Crossing functions of the corresponding conditions. If several Zero-Crossing functions apply the first occurring root is chosen as the next event time $t_e$.

The next step is to prepare the treatment of an event by evaluating the system just before an event at time $t_e - \epsilon$, and shortly after the event at $t_e + \epsilon$. Current derivative-free root finding methods work under the principle that the root is approximated through limits at the two sides, so that the delivered root lies somewhere in the interval $[t_e - \epsilon; t_e + \epsilon]$. Here $\epsilon$ is the tolerance level of the root finding method. Thus the necessary information to treat the event are available after the root is found.

The treatment of an event looks like that: The continuous part is evaluated at the time just before the event $t_e - \epsilon$ and all values are saved to provide them to the `pre()` operator. Then the entire system is evaluated by the function (2) at time $t_e + \epsilon$. At this point the causing event is handled and now further caused events are processed with the so-called Event-Iteration.

Therefore the entire system constantly is re-evaluated, as long as there exist discrete variables $q_j$ that satisfy $\texttt{pre}(q_j) \neq q_j$. Only if for all discrete variables $\texttt{pre}(q_j) = q_j$ is fulfilled, the Event-Iteration has reached a stable state and the next integration step can be performed.