IDA Simulation Environment
a tool for Modelica based end-user application deployment

Per Sahlin Pavel Grozman

Equa Simulation AB
Box 1376, 172 27 Sundbyberg, Sweden
http://www.equa.se/

Abstract

A new Modelica implementation based on IDA Simulation Environment (IDA SE) is presented. IDA SE is primarily used for development of equation based simulators for end-users with limited modeling skills but provides interesting features also for the advanced user. A recently developed Modelica application for simulation of tunnel ventilation for commuter rail networks illustrates IDA usage. Excerpts of models from this application are presented in some detail as well as a list of present limitations of the IDA based Modelica implementation.

1 Introduction

Modelica has proven to be of excellent service to advanced modelers in several domains. However, presently, there is usually close contact between model developers and end-users. In fact, they frequently coincide in a single person. As Modelica uptake evolves, the need to deploy Modelica based simulators among less experienced users is likely to increase. IDA Simulation Environment (IDA SE) has been developed to facilitate this process. Originally based on a Modelica predecessor, NMF [1], IDA SE has been used for equation based end-user application development since the early nineties. Several real-scale simulation applications have been developed, some of which have earned leading roles in their respective markets.

IDA SE is based on the concept of pre-compiled component models, i.e. most IDA application end-users work only with fixed[1] component models that may be combined into arbitrary (input-output free) configurations without need for compilation. Simulators do not require a working compiler installation. Encryption is not needed to preserve component model secrecy. The new Modelica implementation which has been included in the IDA SE package retains this structure, separating the typical roles of the model developer and end-user.

A large majority of potential simulation users have little appreciation of the beauty and generality of an advanced modeling language. They have a design problem to solve and want quality answers with minimum effort. Quite often the full mathematical formulation of the problem is of less interest. A good simulation application must communicate in terms natural to the user and in most situations this does not involve any modeling language but rather physical concepts from the target application. Pipes, pumps and valves may well be the optimal elements of communication rather than differential-algebraic equations.

The structure and main features of IDA Simulation Environment are presented in the next section. In Section 3, a sample IDA application is presented, followed by a discussion about the current state of the Modelica implementation. Some code details from train traffic modeling are discussed in an Appendix.

2 IDA Simulation Environment

Figure 1 shows the three main software modules of IDA SE:

IDA Modeler:	the interactive front-end
IDA Solver:	the numerical DAE solver
IDA Translator:	the model source code editor and processor

A development version contains all three, while a runtime installation lacks IDA Translator. The developer uses IDA Translator to generate a set of C

[1] array sizes, including connector arrays, can be modified after compilation

or F77 routines for each component[2], for equation evaluation, analytical Jacobian evaluation and general information about the model. The code is compiled from the translator into a Windows DLL which is then linked to IDA Solver. The Modelica (or NMF) source may or may not be shipped with the application, depending on the desired level of confidentiality. Also generated are native class descriptions for IDA Modeler, containing structural information about the model library. This code may then be complemented by application specific extensions.

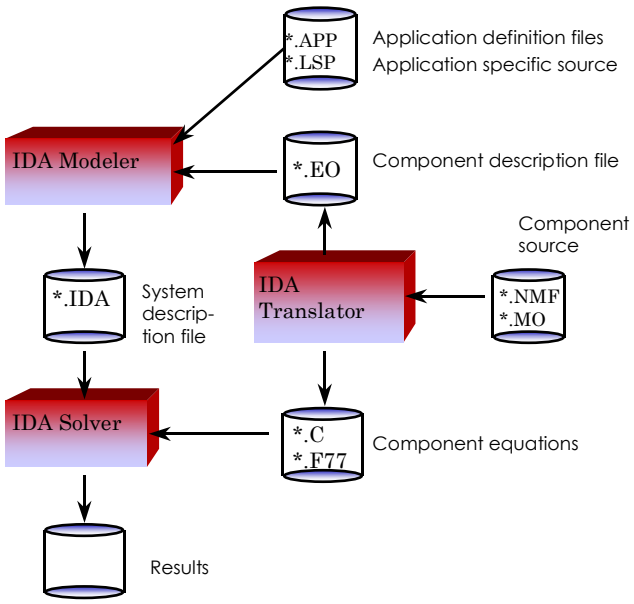

Figure 1: Structure of IDA Simulation Environment

Applications may be shipped stand alone, including an IDA runtime environment or as separate plug-ins for an existing IDA environment. Both the model library and the user interface of an application may be amended and altered by multiple extra separate installations, for customizations and application extensions. This allows efficient management of complex version structures.

The cost of the runtime environment for each installation is significantly lower than that of the full development environment, normally only a small fraction of the cost of the end-user product.

[2] A component or a *compilation unit* becomes an indivisible building block in the end-user application. The Modelica source of a component model may be a composite, hierarchical model. It is also possible to define hierarchical models in IDA Modeler containing multiple components.

IDA Simulation Environment is presently available as an off-the-shelf product only with NMF for Microsoft Windows 98 or NT 4.0 and higher. IDA Solver and Translator have previously been ported to Unix platforms but are not maintained in this setting. Modelica is presently supported only for specific customers. We will return to the state of the Modelica implementation in Section 4.

2.1 IDA Solver

In tools, such as Dymola, where equations are globally reduced prior to integration, the numerical solver will deal with a fairly dense system of equations but where each equation can be quite complex. One can generally expect equation evaluations to take some time while factorization of Jacobian matrices is likely to be faster due to the dense problem structure. In a pre-compiled setting, the situation is the opposite: functions are rather simple (simple enough to differentiate analytically!) while Jacobians are typically large and sparse.

IDA relies on standard software components for sparse Jacobian factorization. Since large sparse matrices occur in many technical and scientific applications a range of powerful solvers are readily available for scalar as well as parallel architectures. Available solvers for IDA are: SuperLU [2], MUMPS [3] and UMFPACK [4]. The graph theoretical analysis of system structure is done by these external solvers rather than in the context of a global symbolic preprocessing.

There are many implications of this difference in solution strategy. A thorough discussion of this is beyond our current scope and we will merely point out a few aspects:

+ Component structure is maintained during integration. This allows for example: (1) Exploitation of special component structure by tailored methods. (2) Component level co-simulation with external tools such as FEMLAB (see Figures 2 and 3). (3) Component level debugging.
+ Equation topology may change during simulation. Since the graph theoretical analysis may be done in each timestep, discontinuities that alter the system structure can be accepted.
+ For few-timestep simulations, global compilation may take a significant part of the total execution time.

- Pre-compiling component models precludes some operations that are natural in a setting where a global symbolic analysis is done. The most serious limitation concerns index reduction. Although index 2 systems generally can be simulated without any problems in IDA Solver, serious high-index problems are most likely better solved by means of global symbolic analysis.

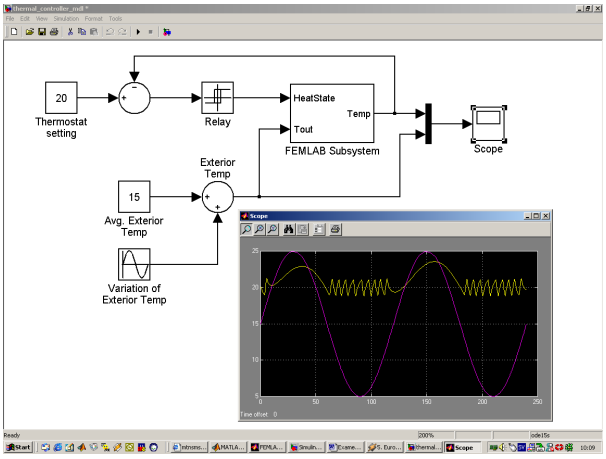

Figure 2: A FEMLAB-Simulink standard case "Thermal controller." A heat source in a 2D region is controlled by a thermostat.

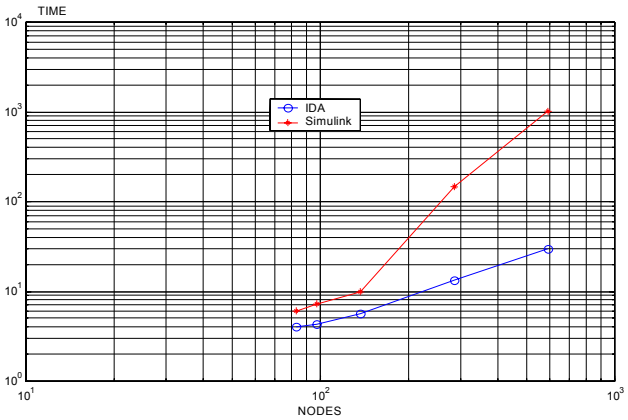

Figure 3: Execution time vs. FEMLAB spacial resolution in the "Thermal controller case". The original Simulink model is compared to an identical FEMLAB-IDA model (from [5]).

IDA Solver is a variable timestep and order solver based on the MOLCOL implicit multistep methods, which include the most common implicit methods such as BDF. Explicit methods are currently not available for the global integrator but may be implemented for individual components.

A selection of methods for initial value computation are available: damped Newton, line-search, gradient and homotopy (embedding) methods

2.2 IDA Modeler

IDA Modeler provides a framework for interface development. It may be used to write simulation oriented applications of sufficient quality for competition with tools written from scratch but at a fraction of the cost. IDA Modeler exploits the fact that many tasks are common to most simulation applications: building and presenting models, editing parameters, interacting with a data base, making simulation experiments, viewing results as diagrams and reports, checking user licenses etc.

More elaborate IDA applications, divide the user interface into three levels, to serve users with different needs and capabilities:

Wizard level:	Least demanding. Each required input is presented in a sequence of user input forms.
Standard level	Intermediate. The user is required to formulate a model, but in terms that are natural to the domain.
Advanced level	The user builds a model using equation based objects. Facilities for model checking, automatic mapping of global data, selection of given variables and similar tasks are available.

In such an IDA application, the Advanced level interface offers a model-lab work bench similar to that offered by other DAE environments, providing the user with direct contact with the individual equations, variables and parameters of the mathematical model. However, a great majority of end-users prefer the tools of the Standard and Wizard level interfaces, where the basic mental concept is that of a physical system and not of a mathematical model.

The kernel of IDA Modeler is written in Common Lisp but most application programming is done interactively or by writing native scripts. Extensive facilities are available to simplify common tasks such as: building user interfaces in multiple natural languages; defining a data bases for input data objects; report generation; data mapping etc. Some user interface elements, such as dialog boxes with complex logic, may be written via an API in other languages.

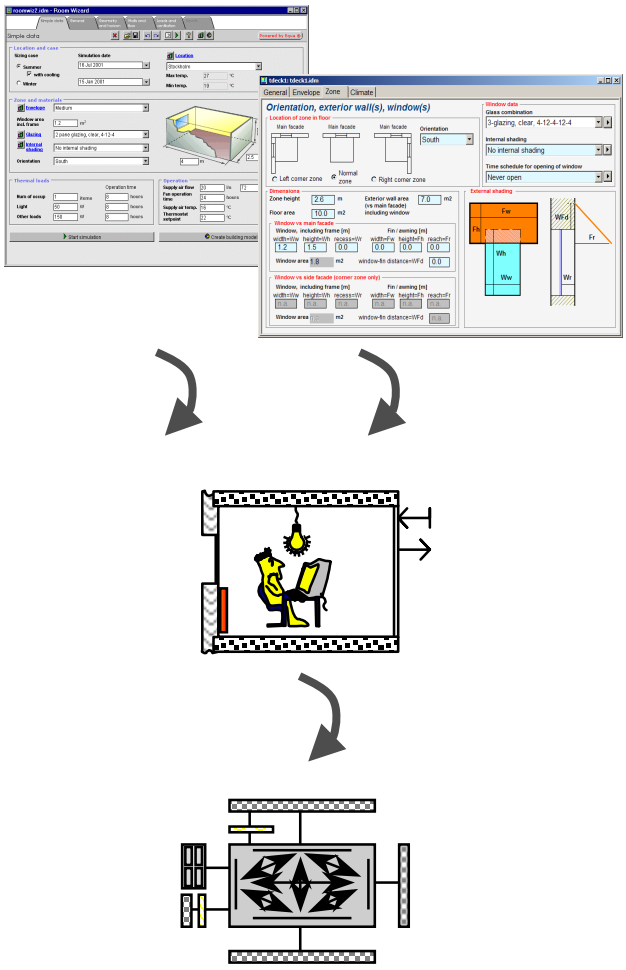

Figure 4: Applications may have multiple Wizard level interfaces for typical simulation tasks. Each interface has a separate data model and a tailored script language for data mapping between levels is provided.

Special emphasis has been laid on tools for development of web clients, running in a browser, powered by an IDA based simulation engine on the (Windows) server. A large portion of the native data structures have been mapped to Java script, facilitating advanced web development with minimum effort.

Several examples of full-complexity applications written in IDA Modeler are available. Equa markets two such applications: IDA Indoor Climate and Energy (IDA ICE) and IDA Road Tunnel Ventilation. Others have been developed for specific customers. IDA ICE is with more than 2000 users a leading international tool for thermal building simulation, available in six languages.

3 Ventilation and fire in commuter rail tunnel networks

The first full-complexity Modelica based IDA application concerns prediction of air flows in tunnels and on platforms of commuter rail networks. Results are needed for several reasons: hygienic ventilation, thermal comfort, smoke propagation in fire scenarios and for gas and particle dispersion studies.

A primarily pulsating air movement through the system is driven by train piston effect. Secondary driving forces are thermal stack effect, wind pressure on portals and openings and possible fan operation.

In this application, air has been modeled as, weakly compressible, i.e. propagating pressure waves have infinite speed but the temperature-density relationship is modeled (perfect gas law) in order to capture the stack effect. Solving the fully compressible equations is often required for rail tunnel studies to predict the effects of interacting pressure waves but this has not been done here since the solution of the resulting hyperbolic equations is likely to be time consuming and otherwise problematic.

Pressure drop in tunnels is modeled in 1D with conventional pipe flow theory: With the fluid is transported a series of fractions for computation of CO_2, age of air etc. Flows with altering directions, often fluctuating around zero, may be numerically difficult to handle in branched systems with high Reynolds number since coarse approximations of viscous losses tends to produce discontinuities. To overcome these problems Gardel [6] empirical formulae have been implemented for viscous loss coefficients, providing continuity around zero flow situations. Bulk air inertia is modeled leading to an index 2 system. Figure 5 shows a model of a four-station section.

A convenient way of expressing train traffic through the system is essential. A design principle has been to separate the models of the tunnel system from the traffic models. Input data for a train route through the system is depicted in Figure 6, including line segmentation, speed limits, accelerations, dwell times at stations etc. To add a new route, the user selects a sufficient number of objects in the direction of the traffic to unambiguously determine a path. The segmentation of the Route need not correspond to the segmentation of the physical tunnel. (The latter may e.g. depend on needed resolution of, e.g., a smoke front.)

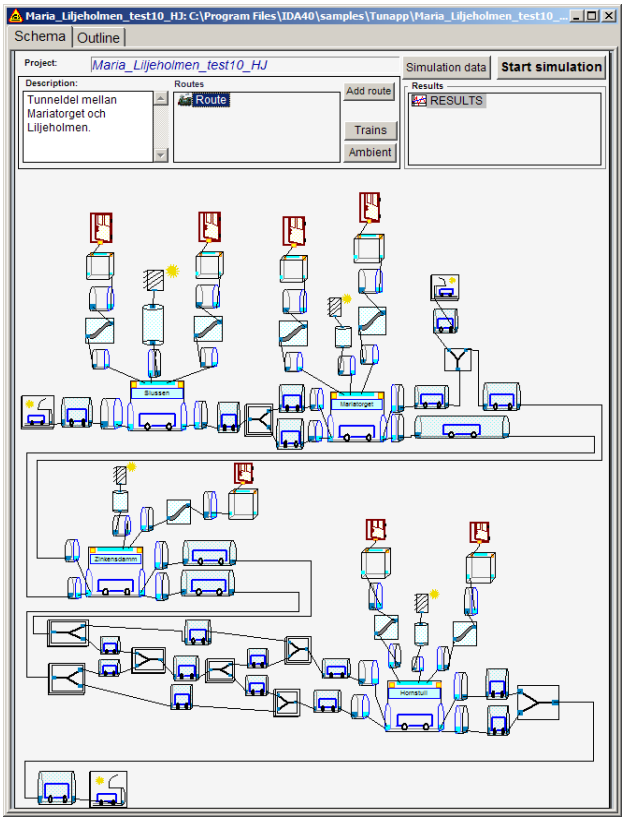

Figure 5: A model of a four-station underground section of the Stockholm subway. Tunnels and other airflow paths are modeled.

Each Route through the system is contained in a single instance of the Route block (code extract in Appendix). This block is then automatically connected to each segment of the physical tunnel using application specific code. The connection lines are not visible, since the number of tunnel segments may be exorbitant.

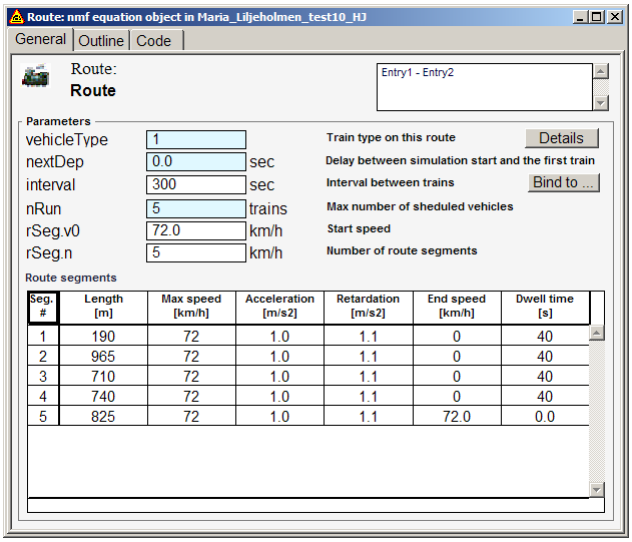

Figure 6: The IDA form for description of a train route through the system.

The management of train routes is a good example of application specific programming, where the standard drag, drop and connect functionality needs to be complemented. The Route form in Figure 6 is an example of a native IDA form, which first has been automatically generated and then subsequently interactively altered. In the Outline tab, the user can see all available parameters, variables and interfaces of the block and in the Code tab, the Modelica code can be browsed (but not edited).

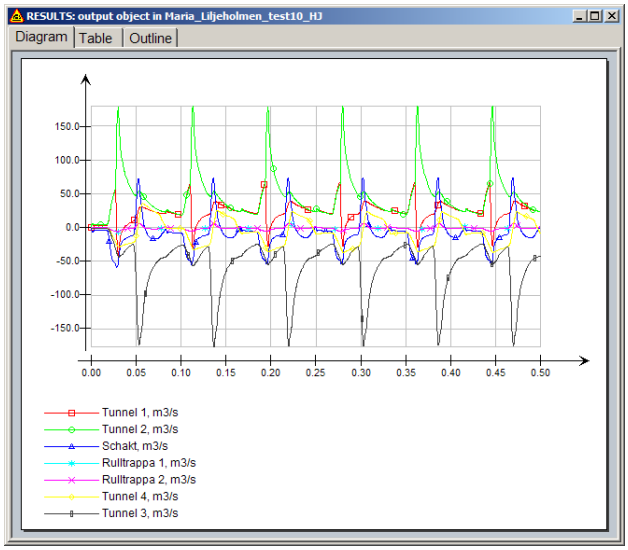

Figure 7: Computed airflows at station Mariatorget, with five minute traffic of C20 trains in one direction.

4 Present state of Modelica in IDA

The current IDA Modelica implementation has been developed to cater to the immediate modeling needs of ongoing projects like the mentioned subway ventilation study. It is our intention to continue to enhance the tools in the scope of cooperative modeling projects and then, at some future point, release an off-the-shelf product.

The design of the Modelica language itself has for natural reasons been centered around the only presently available implementation by Dynasim. In this section, we will outline a few issues where the present Modelica design is less well suited to usage in the pre-compiled setting of IDA and where Modelica extensions have been introduced. Present shortcomings of the implementation are also discussed.

4.1 Interpretation of Modelica code

The IDA Translator compiles classes, not complete systems. Compiled models normally contain:

- public connectors
- more variables than equations
- outer elements
- arrays with non-constant sizes

All public non-partial and non-local classes declared with keywords `class`, `model` or `block` are compiled to IDA components. Blocks are presently compiled to IDA algorithmic models. Public non-partial and non-local atomic types and connector classes are similarly compiled to IDA quantity and link types.

A compiled model may be extended after compilation by inserting and connecting submodels.

Public top-level connectors in compilation units are preserved by the compiler available for connections.

Compilation units may contain unresolved outer components. Such compiled models should be used only as elements of models that contain corresponding inner components. Unresolved outer classes are not supported.

For each compilation unit, a symbolic analysis is performed where as many variables as possible are solved for symbolically, effectively removing them from the global system of equations. Resulting equations are differentiated and code for evaluation of analytical Jacobians is generated. Although principally possible, no index reduction is currently done at this stage.

It is possible to allow the IDA Translator to process entire simulation problems, resulting in just a single compilation unit. However, this is not the intended usage of the tool since the topological flexibility of being able to re-configure precompiled units is an essential feature of most IDA applications.

4.2 IDA driven Modelica extensions

Events in functions and pre operator

The previous IDA language, NMF, supports events in functions, also in foreign functions. This is possible because the variables that monitor events are explicit in NMF models. In Modelica, these variables are automatically generated and not available for the programmer.

We have implemented events using the special function mo_event(*var*, *expr*). The variable *var* is a special kind of variable (called *assigned state* in NMF) that keeps its value from the previous timestep. The function modifies the value of *var* and generates an event whenever it changes sign. In order to be used in a function, the previous value of *var* should be passed to the function and the modified value should be returned. To make this possible, we have changed the semantic of the **pre** operator. In our implementation, pre(*v*) is always the value of *v* at the previous successful time step; this is also valid for non-discrete variables.

The modified **pre** operator may also be used for several other purposes, for example:

- To calculate a maximum value during the simulation:
  ```
  xMax = max(x, pre(xMax));
  ```
- To break an algebraic loop in order to simplify solution of an equation with weak dependences:
  ```
  RhoAir = 1/287.0 * pre(PAir) / Tair;
  ```
- To implement local integration methods, for efficency or for limiting numerical dissipation in PDE:s

A full account of the arguments for the extension of the **pre** operator is beyond the scope of this paper. However, uncontrolled numerical dissipation due to large and variable timesteps is a fundamental problem for many Modelica applications that should be further discussed.

Conversion to strings

In Modelica 2.1 there are functions that converts scalar values to strings, but there are no functions for converting arrays and matrices. We have implemented automatic conversion of non-strings to strings. Example:

```
assert(x>0, "x = "+ x + " should be positive")
```

Graphics

- More named colors
- Arrow: Closed, Left, Right, {type,side}. The size may be a vector
- lineThickness=0 - non-scaled minimal thickness
- Transformation: negative scale and aspectRatio may be used instead of flip.

4.3 Features yet to be implemented

The following list is intended to give a flavor of the present state of development.

Available variable and parameter types

- All variables and non-scalar parameter declared as Integer or Boolean are converted to Real. These variables cannot be used as arguments of function calls.

- Boolean scalar parameters are converted to Integer.
- String variables are not implemented (string parameters are supported)
- Attributes (except value and start) should be constant. They cannot be used in expressions.
- Attributes displayUnit, fixed, enable, nominal, stateSelect are not used.

Connections

- Connection of subconnectors is not yet supported

Modification and redeclarations

- Modifications of class elements are not supported (i.e., when instantiating or extending a class, it is not possible to modify local classes in that class).
- No subtype checking in redeclarations. The constraining clause is ignored.
- Choice annotations not supported.

Expressions

- Record constructors are not supported.

Iterations

- Multiple iterations (separated by ",") not yet supported.
- Ranges with step from:step:to are not supported.
- Vectors in indices only partially supported.
- The index **end** is not supported.
- Deduction of range is not implemented.

Arrays

- Array expressions (not instances) may not be used as arguments to non-built-in functions.

Functions

- Optional arguments are not supported (except in some built-in functions)
- Record arguments are not supported.
- Protected variables in functions are not supported.
- The annotation derivative is not yet supported.
- Some restrictions on external functions.
- Not all Modelica utility functions are implemented.
- External objects are not implemented.

Initialization

- Initial equation/algorithm not implemented

Built-in functions and operators

- Not implemented functions: `initial, terminal, smooth, sample, edge, change, reinit, terminate, div, rem, integer, cardinality`.

Graphics

- Attribute visible and smooth is ignored.
- Cylinders and Sphere fill patterns are not supported.
- BorderPattern shown as rectangle with 3D border
- No line pattern if lineThickness >= 0.375
- Text rotation is not implemented
- Filled text is not implemented.
- Bitmaps: may be rotated by 90 degrees only, imageSource not implemented, fileName just copied (no directory information added).

5 Summary and further work

The present IDA Modelica implementation is a sufficient base for complex application development and delivery. Several partner projects are underway, where Equa supports developers with needed new functionality. Perceived user demand will determine when a public product is released.

Equation based simulation is presently limited by fragmentation into disparate single-vendor user communities. As a technology, Modelica is sufficiently neutral and powerful to break the presnet status quo. Hopefully, another reasonably complete independent implementation will aid this process. However, it is vital that the present Modelica community focuses on the truly critical success factors rather than on yet another intriguing technical issue.

References

1. P.Sahlin, E.F.Sowell, „A Neutral Format for Building Simulation Models", Proceedings of the IBPSA Building Simulation '89 conference, Vancouver, Canada, 1989
2. J.W. Demmel, J.R. Gilbert and X.S. Li, "SuperLU User's Guide", Technical Report, UC Berkeley, USA, 1997
3. P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, "MUMPS Multifrontal Massively Parallel Solver v. 2.0", Technical Report, CERFACS, France, 1998
4. T.A. Davis, "UMFPACK v. 4.0 User Guide", Technical Report, Univ. of Florida, Gainesville, USA, 2002
5. C. Panagiotopoulos "Finite element models in a lumped model simulation environment. An interface between FEMLAB and IDA S.E." Technical Report, KTH, Stockholm, 2001
6. Gardel, A. (1957), "Les Pertes de Charge dans les Ecoulements au Travers de Branchements en Te", Bull. Tech. De la Suisse Romande, 83, 123-130, 144-148, 1957

Appendix - Structure of commuter rail model

The Traffic connector transmits information about train location, speed and acceleration between the Route block and the physical tunnel model:

```
connector Traffic  "Traffic line in tunnel segment"
  Velocity  speed(start=0)  "traffic speed";
  Real      nFront          "no of vehicle fronts per segment";
  Real      nBack           "no of vehicle backs per segment";
  Length    lBody           "total length of vehicles per segment";
  Acceleration acc(start=0) "traffic acceleration";
end Traffic;
```

Below is the template for a Tunnel system. The end user may add instances of different models (sections, platforms, ventilation shafts, traffic routes) into a compiled Tunnel system and then connect and simulate the system (see Figure 5).

```
// The template for Tunnel document
model Tunnel "Tunnel Document"
  inner parameter ArraySize nFract = 2   "Number of air fractions";
  inner parameter ArraySize nVeh=1       "Number of vehicle types";
  inner parameter Vehicle[nVeh] veh      "Description of vehicles";
  inner parameter Fraction[nFract] fract "Description of air fractions";
  Ambient amb                            "Properies of ambient air";
end Tunnel;
```

A traffic route is modeled as a Modelica block. Each instance describes a route in one direction. The model is connected (using `traffic` connector) with segments in tunnel sections and platforms (a tunnel section may consist of several segments). The connection is done by the application; the user only draws the route on the tunnel schema.
The route block is translated to an *algorithmic* model. It does not add equations to the tunnel system, but only supplies the system with input data series (like a table). IDA SE supports also *post-processing* algorithmic models, used for collecting and transforming measurements on a model.

```
block Route
// Array sizes
  parameter ArraySize
    nSched = 2 "Number of points in route schedule",
    nSeg   = 1 "Number of tunnel segments",
    nRun   = 5 "Max number of scheduled vehicles";
 final parameter ArraySize nPos = nSeg + 1 "Number of segment ends";

// Route schedule
  parameter Time tSched[nSched]     = {0, 3600} "time column in schedule";
  parameter Velocity vSched[nSched] = {10, 10}  "speed column in schedule";
  parameter Length xSched[nSched]               "position column in schedule";
  parameter Length xSched0          = 0         "start position for schedule";
// Tunnel segments
  parameter Length lSeg[nSeg] "segment lengths";
  parameter Boolean reverse[nSeg] = fill(false,nSeg) "traffic direction";
  parameter Length xSeg[nPos] "segment ends";
// Time schedule
  Integer lastRun(start=0) "last scheduled vehicle";
  discrete Time
     nextDep(start=time.start)     "Next departure time",
     interval(start=300)           "departure interval",
     depTime[nRun] (each start=-1) "Departures time";

  parameter input Integer vehicleType = 1;
  output Traffic[nSeg] traffic;
  outer parameter ArraySize nVeh;
  outer parameter Vehicle[nVeh] veh  "Description of vehicles";

protected
  Length xFront, xBack, xF, xB;
  Velocity v;
  Acceleration  a;
  parameter Length lVeh = veh[vehicleType].length;
  parameter Time tMax "max route time";
  parameter Time tFront[nPos], tBack[nPos];
```

```
// parameter processing
algorithm
// Calculate the train position at scheduled time points
  xSched[1] := xSched0;
  for i in 1:nSched-1 loop
    xSched[i+1] := xSched[i] + 0.5*(vSched[i]+vSched[i+1])*(tSched[i+1]-tSched[i]);
  end for;
// maximal time per route
  tMax := tSched[nSched] +
          (if vSched[nSched]==0 then 0 else lVeh/vSched[nSched]);
// segment lengths
  lSeg := xSeg[2:nSeg+1] - xSeg[1:nSeg];
// the time then the train passes tunnel segments
  for i in 1:nPos loop
     tFront[i] := RouteTime(xSeg[i], nSched, tSched, xSched, vSched);
     tBack[i] := RouteTime(xSeg[i]+lVeh, nSched, tSched, xSched, vSched);
  end for;

algorithm
  // calculate the traffic parameters on each segment
  // the tunnel segments reads them (using traffic connector)
// Launch the next train
  when time>=nextDep then
   lastRun := mod(lastRun, nRun) + 1;
   assert(depTime[lastRun]<0, "The max number of scheduled trains is exceeded");
   depTime[lastRun] := nextDep;
   nextDep := nextDep + interval;
  end if;
// Initialize output variables
  for iSeg in 1:nSeg loop
    traffic[iSeg].speed := 0.0;
    traffic[iSeg].nFront := 0.0;
    traffic[iSeg].nBack := 0.0;
    traffic[iSeg].lBody := 0.0;
    traffic[iSeg].dSpeed := 0.0;
    traffic[iSeg].acc := 0.0;
  end for;
  // loop over all running trains
  for iRun in 1:nRun loop
    if depTime[iRun]>=0 then  // if not removed
      if time >= depTime[iRun] + tMax then
        // the train is out of tunnel, remove it
        depTime[iRun] := -1;
      else
        // calculate the position, speed, and acceleration
        (xFront, v, a) :=
            RouteInt(time - depTime[iRun], nSched, tSched, xSched, vSched);
        xBack := xFront - lVeh;
        // loop over tunnel segments
        for iSeg in 1:nSeg loop
          // calculate the position of the train in the segment
          xF := xSeg[iSeg+1];
          xB := xSeg[iSeg];
          // is the train on the segments (with events)?
          if time>depTime[iRun]+tFront[iSeg] and time < depTime[iRun]+tBack[iSeg+1] then
            traffic[iSeg].speed := if reverse[iSeg] then -v else v;
            traffic[iSeg].acc   := if reverse[iSeg] then -a else a;
            if time<=depTime[iRun]+tFront[iSeg+1] then
              // count the train fronts
              xF := xFront;
              traffic[iSeg].nFront := traffic[iSeg].nFront + 1;
            end if;
            if time>depTime[iRun]+tBack[iSeg] then
              // count the train backs
              xB := xBack;
              traffic[iSeg].nBack := traffic[iSeg].nBack + 1;
            end if;
            // count the total length
            traffic[iSeg].lBody := traffic[iSeg].lBody + (xF - xB);
          end if;
        end for;
      end if;
    end if;
  end for;
protected
  function RouteInt "Integrates the train movement along the route"
    input Time t            "time elapsed from the start point";
```

```
    input Integer n         "number of intervals in the schedule";
    input Time tp[n]        "time column in schedule";
    input Length xp[n]      "position column in schedule";
    input Velocity vp[n]    "speed column in schedule";
    output Length x         "train position";
    output Velocity v       "train speed";
    output Acceleration a   "train acceleration";
  external;
  end RouteInt;
  function RouteTime "Returns the train time at given position"
    output Time t           "the calculated train time";
    input Length x          "the given train position";
    input Integer n         "number of intervals in the schedule";
    input Time tp[n]        "time column in schedule";
    input Length xp[n]      "position column in schedule";
    input Velocity vp[n]    "speed column in schedule";
  external;
  end RouteTime;
end Route;
```

The tunnel segments and platforms are connected using `TunnelCut` connector:

```
connector TunnelCut
  outer parameter ArraySize      nFract  "Number of air fractions";
                  Pressure       P;
  flow            MassFlowRate   m_dot(start=0);
                  Temp_C         T(start=10);
  flow            HeatFlowRate_M Q;
                  Real           vf[nFract];
  flow            MassFlowRate   vf_dot[nFract];
end TunnelCut;
```

The bi-directional flow of air with fractions (of CO2, NO, dust, smoke etc.) is modeled in a similar way as in the Modelica Fluid package, but the implementation is different.
Here the end-user (working with pre-compiled components) is able to define media properties, especially number of air fractions. Therefore the number of fractions `nFract` is defined as a parameter and not as a constant as in the Modelica Fluid package.