USING SPARK AS A SOLVER FOR MODELICA

Michael Wetter[1], Philip Haves[1], Michael A. Moshier[2] and Edward F. Sowell[3]
[1]Building Technologies Department, Environmental Energy Technologies Division,
Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA.
[2] Dept. of Mathematics, Computer Science and Physics, Chapman University, CA 92866, USA.
[3] Dept. of Computer Science, California State University, Fullerton, CA 92834, USA.

ABSTRACT

Modelica is an object-oriented acausal modeling language that is well positioned to become a de-facto standard for expressing models of complex physical systems. To simulate a model expressed in Modelica, it needs to be translated into executable code. For generating run-time efficient code, such a translation needs to employ algebraic formula manipulations. As the SPARK solver has been shown to be competitive for generating such code but currently cannot be used with the Modelica language, we report in this paper how SPARK's symbolic and numerical algorithms can be implemented in OpenModelica, an open-source implementation of a Modelica modeling and simulation environment. We also report benchmark results that show that for our air flow network simulation benchmark, the SPARK solver is competitive with Dymola, which is believed to provide the best solver for Modelica.

INTRODUCTION

In 1996, a consortium formed to develop Modelica, an equation-based, object-oriented, acausal language for modeling of large physical systems. The goal of the consortium is to combine the benefits of existing modeling languages and to define a new uniform language for model representation (Mattsson and Elmqvist 1997; Fritzson and Engelson 1998). Over the past decade, the Modelica language has gained significant adoption in various industrial sectors and has been used in demanding industrial applications. It is well posed to become a de-facto standard for modeling of complex physical systems that may contain hydraulic, thermal, control, mechanical, electrical, electronic or process-oriented subsystems. There are several modeling and simulation environments that assist modelers in developing models and that can translate models expressed in the Modelica language into an executable program. OpenModelica is an open-source implementation of such an environment.

Over the past twenty years, the Simulation Problem Analysis and Research Kernel (SPARK) has been developed (Sowell, Buhl, and Nataf 1989; SPARK 2003). SPARK uses the mathematical graph to describe and solve nonlinear, deterministic, continuous differential algebraic systems of equations (DAE systems) that are defined on the real space. This solution approach has shown to be competitive for the simulation of air flow networks that are common in building energy systems and typically difficult and time-consuming to solve (Sowell and Haves 2001).

Given the significant adoption of the Modelica language, its position to become a de-facto standard for modeling of physical systems and its benefits for expressing large complex systems (Fritzson 2004) we investigate the approach and benefits of making the SPARK solver available within the OpenModelica environment. Having available a computationally efficient free open-source implementation of Modelica will lower the barrier for adopting Modelica by the rather first-cost sensitive small engineering firms that are characteristic for the building industry.

This paper addresses therefore

1. how SPARK can be used as a solver for the OpenModelica environment, enabling OpenModelica to efficiently solve large, nonlinear sparse DAE systems, and
2. how SPARK compares in terms of reduction in problem size and execution speed with Dymola, which contains what is believed to be the best symbolic and numerical solver for Modelica (Cellier and Kofman 2006).

At the time of this writing, a prototype implementation of SPARK in OpenModelica has been created.

Our computational results are based on an air flow network of a variable air volume flow system (VAV system) that is described by Haves et al. (1996). The algebraic equation system is characteristic for the type of nonlinear equation systems encountered when simulating building heating, ventilation and air conditioning systems.

We will also discuss benefits of having additional Modelica language constructs available that moves the burden of creating efficient run-time models from the model user to the tool that automatically parses the Modelica model into executable code. Such a language construct may allow a Modelica parser to automatically pick different function inverses for the equation that relates pressure drop and mass flow rate. In our experiments, making available to the Modelica parser one or the other inverse led to a tenfold change in computation time.

In related work, Sowell and Haves (2001) compared the computation time of SPARK and HVACSim+ (Park, Clark, and Kelly 1985), for a variable air volume flow system that serves six rooms. They attribute the 15 to 20 times faster computation time of SPARK to the graph decomposition and cut set reduction. Wetter and Haugstetter (Wetter and Haugstetter 2006) showed for a multizone building energy model that Dymola was four times slower than TRNSYS (Klein, Duffie, and Beckman 1976). This despite the fact that the TRNSYS heat conduction model is based on conduction transfer functions (Seem 1987) that often lead to much faster integration than the finite difference scheme that was used in the Modelica model.

SPARK METHODOLOGY

We will now give a brief description of the SPARK methodology. See Sowell et al. (2004) for more details. SPARK is best described in terms of *objects*, *models*, and *problems*. Here an *object* is an instance of an atomic class[1] which is based upon a single equation involving the variables presented at its ports.[2] Atomic classes are acausal, i.e., there is no unique input or output set, a characteristic achieved through provision of explicit inverses for as many of the variables as possible. A SPARK *model* is a collection of objects (instances of atomic classes) linked together at their ports. The links represent the model variables. Like atomic classes, models are acausal. Finally, a SPARK *problem* is defined when the user specifies an input set.

Preprocessing of a SPARK problem begins by parsing the input file, producing a more compact representation in which all macro classes have been resolved into atomic objects and redundant links are removed. Next, graph-theoretic algorithms are used to construct an efficient numerical solution sequence. First, a *bipartite graph* is constructed with one node set representing objects and the other representing variables. For each node in the object set an edge is inserted for each of its ports that has been provided an inverse, going to the corresponding node in the variable node set. A matching algorithm is then used to find a *complete matching*, whereby each object node is matched to a single variable node and vice versa. During the numerical stage, this matching indicates which equation will be used to calculate each variable. After matching, a *directed graph (digraph)* of the problem is constructed, with each node representing an object and producing the variable to which it was previously matched. The digraph is used to determine the actual numerical solution sequence.

Two reductions are performed on the digraph: *component decomposition* and *small cutset discovery*. Components are classified as *strong* or *weak*. Weak components are topologically sorted sequences of nodes, i.e., having no cycles (closed circuits, loops). A strong component is strongly connected in a graph-theoretic sense, meaning it comprises a maximal set of nodes and edges such that every node is reachable from every other node, meaning that one or more cycles are present. The set of components in the problem, including both weak and strong, along with edges between components, defines a topologically sorted *reduced graph*.[3] From a numerical perspective weak components represent strictly sequential computations, i.e., no iterations, whereas strong components require simultaneous, iterative solution. The reduced graph prescribes the overall problem solution sequence, i.e., each component is solved in the topological sequence indicated by the reduced graph, processing weak components sequentially and strong components iteratively. Note that at this point we have likely achieved the first stage of reduction since the largest strong component is often far smaller than the total number of problem unknown variables.

The second stage of reduction is small cutset discovery. Here we seek to identify a small set of nodes that break all cycles within the strong components using a contraction algorithm similar to that described by Levy and Low (1988). From the numerical perspective, the variables corresponding to the nodes in the cutset comprise the iteration vector. SPARK uses this vector and the equations in the strong component to compute a numerical approximation to the Jacobian used in Newton-Raphson iteration.[4] For many problems, including flow networks, the Jacobian is significantly smaller than the total number of variables in the strong component, thus achieving the second level of problem reduction.

The process described above often proceeds automatically, producing a viable solution without user intervention. However, sometimes no matching is found in a well-posed problem simply because needed inverses were omitted in the definitions of atomic classes. Or, a matching may be found, but one that results in a solution sequence exhibiting poor convergence properties. To deal with issues like this, the SPARK language provides optional user specifications to guide the development of a solution sequence. For example, the key word `RESIDUAL` allows the developer of an atomic class to use an *implicit*

[1] A macro class is available to represent larger modeling elements. We omit them in this discussion since they are resolved to their constituent atomic classes before graph-theoretic processing.

[2] Multi-valued atomic classes are allowed, but are omitted here for brevity.

[3] Note that due to the definition of strong component, there can be no cycles in the reduced graph.

[4] This is the basic solving method; several other strategies are employed if convergence is not progressing adequately.

inverse when an explicit one is not available, and the key words `MATCH_LEVEL` and `BREAK_LEVEL` can be used to give hints to the matching and cutset algorithms. Another powerful feature is `PREDICT_FROM_LINK` which allows an iteration variable to be initialized based on some other problem variable, perhaps calculated from a simplified auxiliary equation. Seasoned users often use these features to improve solution efficiency.

DYMOLA METHODOLOGY

When translating Modelica models into executable code, Dymola uses *partitioning* and *tearing* to reduce the dimensionality of the system of equations. Partitioning discovers which equations are coupled to each other and hence need to be solved simultaneously. Partitioning can conceptually be described as converting the structure incidence matrix[5] to block lower-triangular form. (Since the memory requirements to store the structure incidence matrix is prohibitive for large system, the matrix is not actually formed but rather used here for the sake of explanation.) If partitioning yields a true lower-triangular form (i.e., a matrix with only scalars on the diagonal, and zeros everywhere above the diagonal), then all equations can be solved individually. In general, however, the diagonal contains matrices, although their dimension is typically considerably smaller than the dimension of the original system. The partitioning algorithm in Dymola is based on the algorithm of Tarjan (1972).

To reduce the size of an individual coupled system of equation, Dymola uses a process called *tearing*. Suppose there is a system of equations with an unknown $x \in \mathbb{R}^n$, with $n > 1$, and let x be partitioned into the two vectors x^1 and x^2. In tearing, the system of equation is rewritten in the form

$$Lx^1 = f^1(x^2), \quad (1)$$
$$0 = f^2(x^1, x^2), \quad (2)$$

where L is a lower triangular matrix with constant nonzero diagonals and x^2 is called the *tearing variable*. Now, the solver provides a guess value for x^2, obtains x^1 from (1) and computes a new value for x^2 using (2). This procedure is repeated iteratively until x^2 converges to a solution. Dymola's tearing algorithm guarantees that the selection of tearing variables never leads to a division by zero at run-time, which is non-trivial to implement since parameter values can change after compilation. The details of the tearing algorithm implemented in Dymola are unpublished. For a more detailed discussion of Dymola's algorithms, including its algorithms for index reduction and inline integration, which have not been used in our algebraic problems, we refer to Cellier and Kofman (2006) and Elmqvist, Otter, and Cellier (1995).

[5] For a system of equations $f(x) = 0$, the structure incidence matrix is a matrix whose element (i, j) is 1 if $f^i(\cdot)$ depends on x^j, and zero otherwise.

APPROACH FOR IMPLEMENTING SPARK IN OPENMODELICA

To understand the potential for integrating the SPARK analyzer and solver into OpenModelica, one must first recognize that these components of SPARK are independent of the SPARK language. The algebraic aspects of SPARK analysis requires only that the DAE system of equations be presented in a general form that includes two main specifications:

- A bipartite dependency graph indicating which variables appear in which equations.
- For each individual equation occurring in the model, a list of inverses, i.e., C++ functions that represent a solution of the equation in the form $y = f(x)$ where $f\colon \mathbb{R}^n \to \mathbb{R}$ for some $n \in \mathbb{N}$.

Importantly but somewhat subtly, an inverse is not required to be explicit. That is $y = \widehat{f}(x, y)$ where $\widehat{f}\colon \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ for some $n \in \mathbb{N}$ is permissible and can be automatically generated using the `RESIDUAL` keyword. The analyzer will generally be able to construct a more efficient solution if fewer implicit inverses are present, but SPARK manages them with a graceful degradation of performance.

The analyzer performs the matching described above. After the matching, the system is essentially a directed graph in which a node represents an equation in the form $y = f(x)$ and an edge from such a node represents the appearance of the variable y in the independent variables of another inverse. SPARK then performs the graph-theoretic equivalent to Dymola's partitioning algorithm, and finally constructs a small cut-set.

The cut-set construction serves the same purpose as the tearing algorithm of Dymola. That is, it identifies a subset of the variables occurring in a partition that breaks all dependency cycles in the partition.

Integration of SPARK into the OpenModelica system involves the following main tasks.

1. Constructing the bipartite graph of variables versus equations from a Modelica model.
2. Deriving, or in some way specifying, inverses of equations.
3. Expressing heuristic information `MATCH_LEVEL` and `BREAK_LEVEL` in the Modelica model, and communicating it to the SPARK analyzer.

The first of these is essentially a task of exploiting the OpenModelica open source architecture to gain access to information that encodes which problem variables appear in which equations. All of this information is already present in OpenModelica's parser, so the task is simply

to extract it. The second task would, in principle, be implemented best using an open source symbolic algebra system such as SAGE[6]. In the first version, however, we support the derivation of inverses for equations built from a short list of basic functions. For more complicated equations, we require the user to provide the inverses explicitly in the form of annotations on equations. In later versions, we anticipate integrating a full symbolic algebra system with the SPARK/OpenModelica system. For the third task, we again use Modelica annotations. Because the annotations are maintained by OpenModelica's symbol table, this is also a relatively simple task.

Although experienced SPARK users exploit `PREDICT_FROM_LINK` and other features to improve the numerical behavior of integration, these features are not supported in the current integration of SPARK in OpenModelica. We anticipate supporting these features through the annotation mechanism of Modelica.

Currently, we have modified the OpenModelica parser to emit a form of the bipartite graph. This confirms that the open source architecture can be successfully exploited for this integration task. However, putting this information in a format that is usable by SPARK is not yet completed. Also, generation of inverses and communication of the annotated information regarding `MATCH_LEVEL` and `BREAK_LEVEL` to the SPARK analyzer are not yet implemented. The generation of inverses is a big design task, so in the earliest version we will produce inverses by hand.

Two additional open issues remain to be resolved in future versions. First, to be fully functional as a part of OpenModelica, the SPARK solver must communicate its results and error messages back to OpenModelica via the same API as the system's native solver. In principle, this will be possible, but we have deferred implementation until we have an otherwise functioning prototype. Second, the Modelica language supports discrete variables and variables that are set via algorithms expressed in the Modelica language as opposed to being constrained by acausal equations. SPARK provides limited indirect support for discrete variables via the `PREDICT_FROM_LINK` feature. It also provides algorithmic control over variables, but the algorithms are expressed in C++. Thus these present significant design problems for the SPARK/OpenModelica integration. In principle, a code generator can be implemented to translate Modelica algorithms into C++ code usable by SPARK, but we have not yet engaged in the design of such a translator.

[6] `http://www.sagemath.org/`

NUMERICAL BENCHMARKS

Problem Definition

The simulation model represents the airflow network of a variable air volume flow system in building E 51 at the Massachusetts Institute of Technology. The models are described in detail by Haves et al. (1996). To facilitate the model implementation in SPARK, we modeled the pressure and mass flow distribution, but not the temperature, enthalpy and species concentration. In Haves et al. (1996) as well as in our implementations, the airflow network is described by a system of algebraic equations, i.e., the pressure dynamics of the room is neglected. All flow resistances are based on a partial model that computes $\dot{m} = f(\Delta p)$ or its inverse (in an explicit form), $\Delta p = g(\dot{m})$, where $f(\cdot)$ and $g(\cdot)$ are implemented using the functions `regSquare2` and `regRoot2` from the package `Modelica_Fluid` 1.0β2. The medium, however, is implemented using a class of type `record` that defines the density and viscosity[7].

Fig. 1 shows the Dymola representation of the air flow network of the system model. The red connectors are constant pressure boundary conditions. The green connectors are control signals, which are defined as ramps as shown in Fig. 3. The blue lines are air flow paths, with dashed lines indicating connections that connect the ports of different instances of a vectorized model of five rooms. The five rooms have been vectorized to scale the problem size. We call this five rooms a *suite*, and denote with the parameter $N_{sui} \in \{1,2,\ldots,5\}$ how many instances we used. Fig. 2 shows the flow network of one suite. Since there is one more room at the end of the last suite (shown as the model `roo50` in Fig. 1), the number of rooms is between six and 26, and is equal to $5N_{sui}+1$. The model has constant properties for the atmospheric pressure.

Experiments of Symbolic and Numerical Solver

We conducted experiments using Dymola 6.0d and SPARK 2.10v7. The experiments were run on Windows XP on an Intel Core 2 Duo Processor 6600 @ 2.40 GHz.

In Dymola, an integration algorithm needed to be selected even for the algebraic problem. We selected the Euler method. In Dymola and SPARK, we simulated the problem on the time interval $t \in [0,\ 1]$ using a fixed step size and output interval of 0.01 seconds. We set the solver tolerance to 10^{-5} and the number of suites to $N_{sui} = \{1,2,\ldots,5\}$. For the simulation with $\varepsilon = 10^{-5}$, the maximum relative error between the supply fan mass flow rate computed by Dymola and SPARK was 0.04% for $N_{sui} = 1$ and 1% for $N_{sui} = 5$, which shows that both

[7] Future versions of our library will be based on `Modelica.Media`, but we used a simpler implementation to facilitate the implementation of a model with identical physics in SPARK.

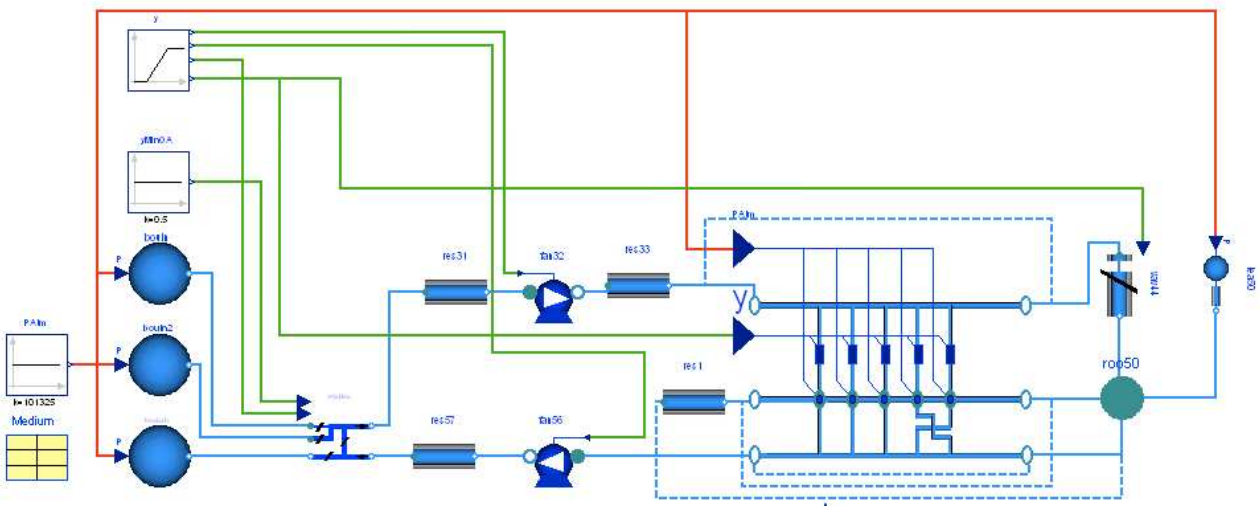

Figure 1: System model of the air flow network as implemented in Dymola.

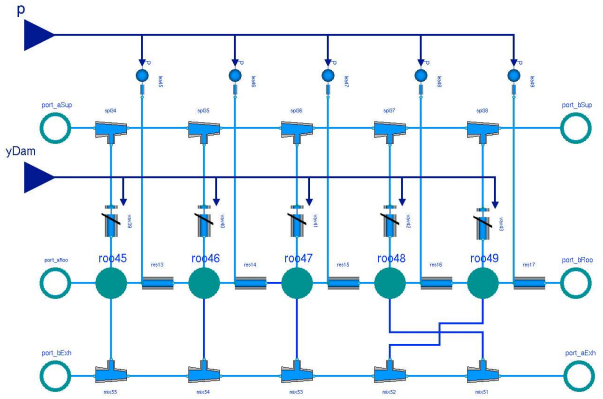

Figure 2: Model of the air flow network of one suite as implemented in Dymola.

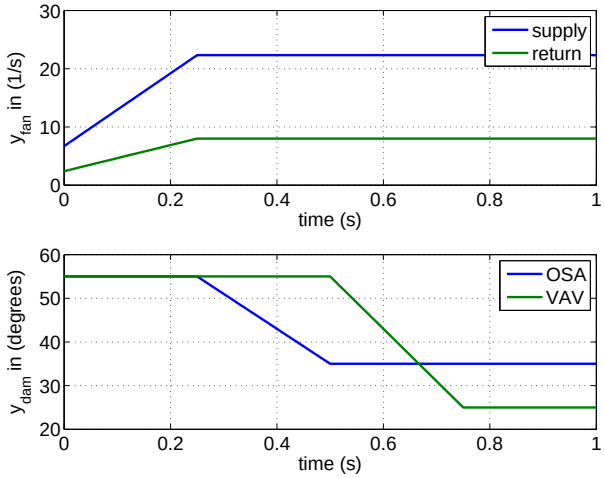

Figure 3: Control signals used in the numerical experiments. The top figure shows the fan revolutions, and the bottom figure the damper opening angle. OSA denotes the outside air dampers and VAV denotes VAV box damper.

simulators implemented the same model.

In Dymola, we run the following experiments:

1. For the flow relation, we set `from_dp = true` for all flow resistances. Hence, Dymola computes $\dot{m} = f(\Delta p)$.
2. For the flow relation, we set `from_dp = false` for all flow elements. Hence, Dymola computes $\Delta p = g(\dot{m})$.

In SPARK, we run the following experiments:

1. No use of the SPARK `MATCH_LEVEL` keyword has been made.
2. In the flow splitter, the SPARK `MATCH_LEVEL` keyword was set to aid the symbolic formula manipulation in tearing the equation graph.

All SPARK cases were run with default values for solving methods, i.e., they employ non-sparse LU decomposition, method 2.[8] For the problem with 5 suites, three other methods have been tested: Gaussian gives about the same execution time and Sparse LU is a little slower. The SVD method fails to solve at about time=0.73.

Results

Tab. 1 shows the dimension of the biggest nonlinear system of equations. For SPARK, there was only one

[8]Note that the SPARK reference manual wrongly says the default is Gaussian.

Table 1: Dimension of biggest nonlinear system of equations. For SPARK, no ML denotes no `MATCH_LEVEL` *and with ML denotes with* `MATCH_LEVEL`.

N_{sui}	SPARK		Dymola	
	no ML	with ML	$\dot{m} = f(\Delta p)$	$\Delta p = g(\dot{m})$
1	21	18	41	26
2	36	34	71	47
3	51	50	101	66
4	66	65	131	87
5	81	80	161	106

strong component with the dimension of the nonlinear system of equations as shown in Tab. 1. In addition, there were three weak components. For Dymola, for the experiments that use the function $\dot{m} = f(\Delta p)$, the nonlinear system of equations is solved for the pressure at ports of the flow elements, whereas for the experiments that use $\Delta p = g(\dot{m})$, the unknowns are the mass flow rates and some pressures variables. There is one nonlinear system of equations of the size shown in Tab. 1. In addition, for all Dymola experiments, there are also $5N_{Sui}+1$ one-dimensional system of equations for the flow coefficients of all variable air volume flow boxes.

For the experiments with $\Delta p = g(\dot{m})$, Dymola finds a symbolic expression for all Jacobian matrices, as it does for all one-dimensional algebraic equations. For the other experiments, Dymola computes a numerical approximation for the largest Jacobian, i.e., the Jacobian with the dimension listed as in Tab. 1. SPARK always computes a numerical approximation to the Jacobian.

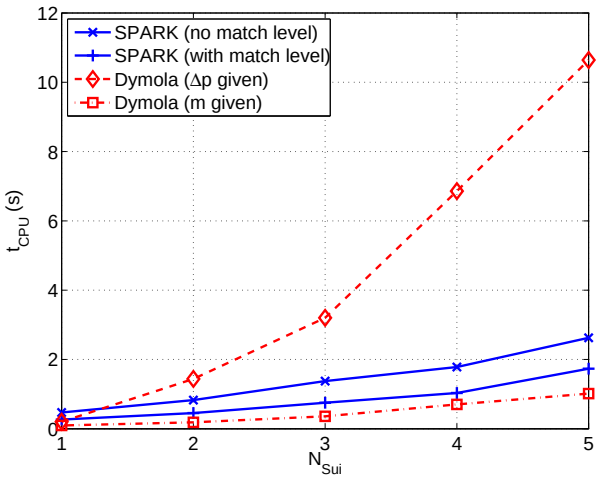

Figure 4: Comparison of computation time.

Fig. 4 shows the computation time for Dymola and SPARK, both with a solver tolerance of 10^{-5}. Run times are shown vs. problem size, the latter expressed as number of suites.

In Dymola, changing the value of the parameter `from_dp` so that Dymola uses $\Delta p = g(\dot{m})$ for all flow resistances yields a tenfold reduction in computation time for $N_{sui} = 5$. In view of this reduction in computation time and in the dimensionality of the nonlinear system of equations, a Modelica language extension that allows a symbolic processor, instead of the user, to select the most appropriate inverse for complicated functions that the symbolic processor cannot invert automatically would be desirable.

In contrast, in SPARK atomic classes, a model builder routinely provides multiple inverses, and the matching process determines which of these inverses gets selected for use in the numerical stage. As a performance-enhancing feature, a user can provide hints for the preferred matchings. Based on experience with pressure/flow networks in general we reasoned that the first resistance in each zone branch should be used to calculate mass flow rate from pressure drop, since that would allow the pressure drop across all others in the branch to be calculated sequentially. To express this domain knowledge the `MATCH_LEVEL` key word on that particular resistance in all zones was used to encourage a matching that favored this strategy. The solution sequence thereby determined used the expected matching in all but one of the zones. Apparently, other portions of the system, e.g., the fans and mixing box, exert their own influence on the matching process, working against our effort to impose a strategy based only on the zone flow branches. Nonetheless, the found matching produced a solution strategy that gave shorter solution times relative to the no-hint solutions.

Returning to the plot in Fig. 4, we can see that both Dymola and SPARK yield solutions for all experiments. Interestingly, the solution times for the two tools are quite different if Dymola is instructed by the user to use $\dot{m} = f(\Delta p)$, but become comparable for the other configurations. We will now discuss reasons for the different computation time.

To explain these difference, we consider the steps in the solution process and their contribution to solution time. When using Newton-based solution schemes for nonlinear equations, as SPARK and Dymola do, the main components of the time to perform a single Newton iteration are: (a) evaluation of functions (residuals or inverses), (b) computation of the elements of the Jacobian, and (c) solution of the linear set to arrive at the next estimate of the break variables. Consequently, overall solution time can

be improved by more efficient performance of any one of these components or by reducing the number of Newton iterations.

Both programs do symbolic reduction in an effort to reduce component (c), so we look first at this component. For $N_{sui} = 5$, we observe that the largest cutset found by Dymola drops from 161 to 106 when the selection of flow-pressure drop inverse is changed, resulting in a 34% reduction in the size of the linear set. Since solution of linear systems is typically $O(n^2)$ to $O(n^3)$, one might expect the solution time to drop by a factor of $10(= 1/0.34^2)$ to $25(= 1/0.34^3)$. We indeed observed a tenfold reduction in computation time.

The user intervention with SPARK changes the cutset only from 80 to 81, so we should not expect much improvement in component (c). This is the principal reason we see far less dramatic change in solution time. That which we do see must be due to either component (a), (b), or reduction in Newton iterations. Optional diagnostic reports allow us to investigate these possibilities. Fig. 5 shows the total solution times (upper curves), and the time spent solving the linear set in the Newton step (lower curves), what we have called the component (c) time. Note that the latter is essentially independent of user effect, as we should expect because the cutset size changed very little. Thus all of the solution time reduction was due to reduction of component (a) or (b). Moreover, we see that the component (c) time is a small fraction of the total time, meaning that any further significant reductions must come from these components, or reduction of the number of Newton iterations.

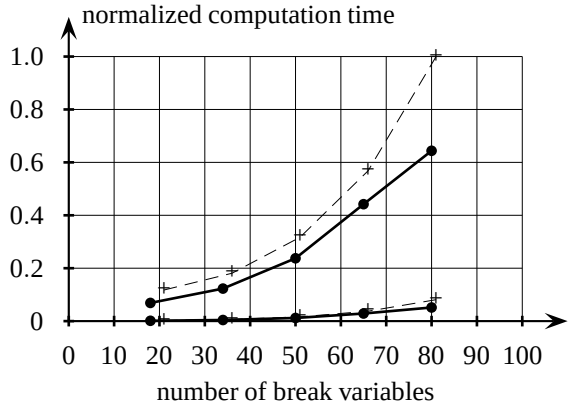

Figure 5: Normalized SPARK solution time for the whole simulation (upper two curves) and normalized time spent to solve the linear system of equations of the Newton iteration (lower two curves). The bold solid curves are with `MATCH_LEVEL` *specified, the thin dashed curves are without* `MATCH_LEVEL` *specification.*

The next step towards understanding the results we show in Fig. 6 the average number of Newton iterations as a function of cutset size. We see a weak effect of cutsize, but a difference of about 1 to 2 iterations for each solution, or about 25% for the largest problem size,[9] explaining about half of the observed 50% solution time reduction for that case. The underlying reason is probably that the new matching reversed the direction of calculation around circuits in the computation graph, thus improving convergence properties.

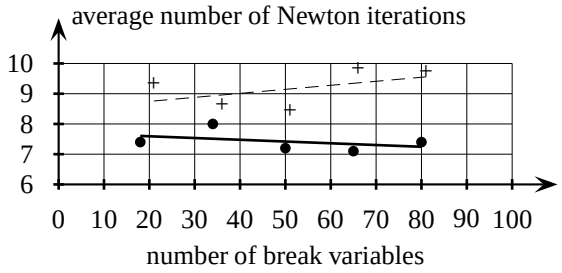

Figure 6: Average number of Newton iterations with `MATCH_LEVEL` *(lower points) and without* `MATCH_LEVEL` *(upper points) and linear fit.*

As a final step we analyze the number of function evaluations for solving the strong component. In Fig. 7, the upper two curves are for the total number of evaluations, while the lower two are for calculations related to updating the break values. The two lower curves are essentially the same because the cutset size change was small. The difference between the upper and lower curves is more important, because it represents the function evaluations needed to calculate the numerical approximation to the Jacobian. Since Dymola reportedly calculates the Jacobian matrices symbolically for the experiments with `from_dp = true`, this suggests that the main difference in solution times may be attributable to that methodology.

CONCLUSIONS

The work performed in developing a prototype SPARK/OpenModelica implementation confirms that the OpenModelica software architecture can be used to emit the bipartite graph for subsequent use by SPARK. The prototype work did not reveal any fundamental problems for such an integration, although significant design challenges with regard to addressing discrete variables need to be addressed.

Our numerical experiments give us confidence that SPARK's performance is for the solution of algebraic set of equations defined by the air flow problem comparable with Dymola, which is believed to provide the best symbolic and numerical solver for Modelica. In view of the prototype work and the encouraging numerical results, we

[9] Keep in mind that the system is solved at 101 steps in time.

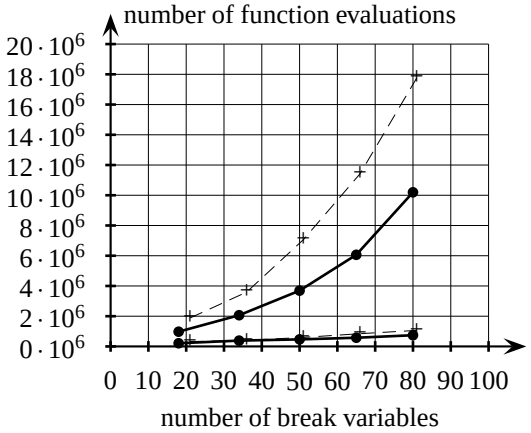

Figure 7: Total number of SPARK function evaluations to solve the strong component (upper two curves) and number of function evaluations to compute the residuals of the strong component (lower two curves). The bold solid curves are with `MATCH_LEVEL` *specified, the thin dashed curves are without* `MATCH_LEVEL` *specification. The difference between the two sets of curves are the number of function evaluations to compute the numerical approximation to the Jacobian.*

believe that integrating SPARK into OpenModelica would be a viable path for providing a free Modelica implementation to building energy analysts which are typically employed by small companies that are sensitive to software costs, thereby decreasing the barrier for adopting Modelica by the building energy community.

ACKNOWLEDGMENT

This research was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technologies of the U.S. Department of Energy, under Contract No. DE-AC02-05CH11231.

REFERENCES

Cellier, François E., and Ernesto Kofman. 2006. *Continuous System Simulation*. Springer.

Elmqvist, H., M. Otter, and F. Cellier. 1995, June. "Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential– Algebraic Equation Systems." *Keynote Address, Proc. ESM'95*. European Simulation Multiconference, Prague, Czech Republic, xxiii–xxxiv".

Fritzson, Peter. 2004. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons.

Fritzson, Peter, and Vadim Engelson. 1998. "Modelica – A Unified Object-Oriented Language for System Modeling and Simulation." *Lecture Notes in Computer Science*, vol. 1445.

Haves, P., L. K. Norford, M. DeSimone, and L. Mei. 1996. "A Standard Simulation Testbed for the Evaluation of Control Algorithms & Strategies." Final report 825-RP, ASHRAE, Atlanta, GA.

Klein, S. A., J. A. Duffie, and W. A. Beckman. 1976. "TRNSYS – A Transient Simulation Program." *ASHRAE Transactions* 82 (1): 623–633.

Levy, Hanoch, and David W. Low. 1988. "A contraction algorithm for finding small cycle cutsets." *Journal of Algorithms* 9 (4): 470–493.

Mattsson, Sven Erik, and Hilding Elmqvist. 1997, April. "Modelica – An international effort to design the next generation modeling language." Edited by L. Boullart, M. Loccufier, and Sven Erik Mattsson, *7th IFAC Symposium on Computer Aided Control Systems Design*. Gent, Belgium.

Park, Cheol, Daniel R. Clark, and George E. Kelly. 1985, August. "An overview of HVACSIM+, a dynamic building/HVAC/control systems simulation program." *Proceedings of the 1st International IBPSA Conference*. Seattle, WA, 175–185.

Seem, J. E. 1987. "Modeling of Heat Transfer in Buildings." Ph.D. diss., University of Madison-Wisconsin.

Sowell, Edward F., W. Fred Buhl, and Jean-Michel Nataf. 1989, June. "Object-Oriented Programmingg, Equation-Based Submodels, and System Reduction in SPANK." *Proceedings of the Second International IBPSA Conference*. Vancouver, BC, Canada, 141–146.

Sowell, Edward F., and Philip Haves. 2001. "Efficient solution strategies for building energy system simulation." *Energy and Buildings* 33 (4): 309–317.

Sowell, Edward F., Michael A. Moshier, Philip Haves, and Dimitri Curtil. 2004, August. "Graph-theoretic Methods in Simulation Using SPARK." Technical Report LBNL-55522, Lawrence Berkeley National Laboratory, Berkeley, CA.

SPARK (Lawrence Berkeley National Laboratory and Ayres Sowell Associates Inc.). 2003. *SPARK, Reference Manual*. Berkeley, CA, USA: Lawrence Berkeley National Laboratory and Ayres Sowell Associates Inc.

Tarjan, Robert. 1972. "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing* 1 (2): 146–160.

Wetter, Michael, and Christoph Haugstetter. 2006, August. "Modelica versus TRNSYS – A Comparison Between an Equation-Based and a Procedural Modeling Language for Building Energy Simulation." *Proc. of SimBuild*. IBPSA-USA, Cambridge, MA.