The Complexity Crisis
Using Modeling and Simulation for System Level Analysis and Design

François E. Cellier[1], Xenofon Floros[1] and Ernesto Kofman[2]
[1]*Swiss Federal Institute of Technology (ETH Zurich), Department of Computer Science, Zurich, Switzerland*
[2]*National University of Rosario (UNR), Department of Automatic Control, CIFASIS-CONICET, Rosario, Argentina*
{*fcellier, xenofon.floros*}*@inf.ethz.ch, kofman@fceia.unr.edu.ar*

Keywords: System Level Modeling; System Level Simulation; Modeling and Simulation Environments; Error Tolerant Modeling; Decentralized Simulation.

Abstract: This paper discusses the current state of the art of modeling and simulation environments and proposes a set of enhancements that will be necessary for such environments to meet future demands. Modeling and simulation are categorized in accordance with the size of the models to be handled: *component level modeling*, *device level modeling,* and *system level modeling*. It is shown that the requirements that modeling and simulation environments need to satisfy in order to meet the demands of modelers are vastly different at these three levels.

1 INTRODUCTION

In the 1970s and 1980s, modelers were mostly content simulating systems at component level. They studied the performance of a particular type of motor, for example. Typically, such models were characterized by half a dozen state variables and a few dozen algebraic variables. A typical general purpose modeling and simulation (M&S) environments used in those years was ACSL (Cellier, 1991; Mitchell and Gauthier, 1986). ACSL offered vertical sorting of equations and a (rather primitive) Macro language that enabled users to group the equations of sub-components together. As of the early 1980s, ACSL also offered a (fairly limited) set of tools for discontinuity handling. However, this was not the important point. The most important feature of ACSL and similar environments was that they freed the modeler from having to understand numerical ODE solution algorithms. The modeler was given a selection of integration algorithms to choose from, but did not need to understand how the model was being simulated. ACSL supported a strict separation of the model equations from the solver equations, and the modeler could thus focus on those aspects that he or she understood well, namely describing the model in mathematical terms.

For models of larger size, tools like ACSL were mostly useless. Larger models coded in ACSL turned quickly into spaghetti code similarly bad as if they had been coded in Fortran or C. Most modelers realized this, and therefore, larger models were mostly coded in general purpose programming languages, such as Fortran. Also, many physical systems do not lend themselves to causal modeling. The computational causality of the equations governing these systems depends on their embedding, and consequently, causal modeling is doomed to failure. For example, an electrical resistor is characterized by Ohm's law:

$$u = R \cdot i \tag{1}$$

but the equal sign must not be interpreted as an assignment equal sign. Depending on the embedding of the resistor in its surrounding circuit, the law may need to be turned around, leading to the equation:

$$i = \frac{1}{R} \cdot u \tag{2}$$

For this reason, ACSL could not be used to model and simulate electrical circuits, for example.

Special purpose modeling languages were designed to offer support for modeling particular classes of implicitly formulated models, such as Spice (Tuinenga, 1995) for the simulation of electrical analog circuits, or Adams (Ryan, 1990) for the simulation of mechanical multi-body systems. These tools were quite successful and found wide-spread use, but they were also limited in their applicability. If a particular component model had not been foreseen by the designers of the tool, such as maybe a Zener diode in Spice, the tool became quickly useless.

The 1990s saw the advent of a new class of M&S environments that were based on the object-oriented modeling paradigm. They offered horizontal next to vertical sorting, thereby enabling the modeler to formulate the model equations in an a-causal fashion. Although a prototype of an object-oriented M&S environment had already been implemented in the late 1970s (Elmqvist, 1978), it took 15 more years until a commercial and more mature implementation of these important new concepts had become available in the form of a commercial release of the Dymola language (Cellier, 1991; Cellier and Elmqvist, 1993). In 1996, the Dymola language was revised once again, and an open standard under the name of Modelica was introduced (Mattsson et al., 1997).

The Modelica initiative turned into a true success story, and Modelica has meanwhile become the de facto industry standard for modeling and simulating all types of physical lumped parameter systems. The Modelica Association (https://www.modelica.org/association) maintains the Modelica language definition (Modelica Association, 2007) as well as the Modelica Standard Library (MSL) (Modelica Association, 2008), to which many researchers from around the globe contribute on a regular basis, and which by now offers large numbers of models from many different application domains. In the meantime, the MSL has become as valuable as if not more valuable than the language definition itself, and a number of both commercial and open source implementations of the Modelica language are being offered that all work with the MSL. Modelica conferences are being held in frequent intervals that attract each time several hundred attendees, and large research projects to the amount of tens of millions of Euros have been funded to further enhance the Modelica language and the MSL.

By means of the object-oriented modeling paradigm that is being embraced by Modelica, it has become possible to connect component models reliably to form larger models; thus, Modelica has become the tool to use for device level modeling and simulation. Typical Modelica models now are comprised of several dozens of state variables and several hundreds of significant algebraic equations, after all of the trivial connection equations of the type:

$$a = b \tag{3}$$

have already been eliminated.

Most Modelica implementations offer sophisticated symbolic preprocessing, including algorithms for index reduction (Cellier and Elmqvist, 1993; Pantelides, 1988; Zimmer, 2010), tearing of algebraic loops (Elmqvist and Otter, 1994; Zimmer, 2010), discontinuity handling (Cellier, 1979; Elmqvist et al., 1993; Otter et al., 1999), and static as well as dynamic state selection (Mattsson et al., 2000), to the extent that the generated simulation code bears little resemblance to the original model equations any longer. The generated simulation code is at least as efficient as if not more efficient than the spaghetti codes of the past, and Modelica models can even compete in simulation efficiency with the special purpose modeling languages (Spice, Adams) designed earlier.

Information hiding is one of the most important features of object-oriented modeling. Implementation details are hidden inside component models that the end user rarely needs to consult. A graphical user interface (GUI), spearheaded once again by the designers of Dymola but later standardized as part of the Modelica language, offers the modeler the ability to hierarchically structure his or her models in such a way that the implementation details of the component models of lower levels of the hierarchy are encapsulated and hidden from users who interface with their model at a higher level of the modeling hierarchy. At each abstraction level, the models can be designed to fit on a single screen, which helps significantly with model understandability, debugging, and maintenance.

Yet, and contrary to the special purpose languages of the past, all physical knowledge is encoded at the Modelica modeling level itself as part of the MSL. Hilding Elmqvist, the originator of Dymola, once proudly stated:

> The most important feature of my language is that Dymola does not understand physics.

The only knowledge that is hard encoded in the Modelica compilers is algorithmic knowledge of how to convert implicitly formulated sets of differential and algebraic equations (DAEs), possibly containing structural singularities and algebraic loops and most likely containing discontinuities, to sets of explicitly formulated ordinary differential equations (ODEs) that can be processed by ODE solvers (Cellier and Kofman, 2006). All of the physical knowledge is soft encoded in the MSL. This is indeed a very important feature as it makes Modelica (Dymola) much more flexible than any of the special purpose languages of the past.

As models with several dozens of state variables are almost invariably stiff, most Modelica environments offer a stiff system solver as their default ODE or DAE solver. The most commonly used default solver is DASSL (Petzold, 1983), a code based on a variable-step, variable-order series of backward difference formulae (BDF) of different orders of approximation accuracy that are well suited for the simulation of stiff systems. DASSL is chosen as the default solver, because it is the most robust stiff system solver on the market today. It can simulate almost any model thrown at it. It does not always do so in a highly efficient manner, but robustness counts for much more than efficiency.

Most Modelica environments also offer a set of alternative solvers to choose from, but the large majority of users never select a different solver, because they don't understand numerical methods anyway and therefore have no inkling as to which solver might be best suited for their particular model. Earlier versions of Dymola offered the user many compiler switches that could help the user in optimizing the efficiency of his simulation runs. All of these compiler options (except for the selection of the solver itself) have meanwhile been thrown out again (or at least have been hidden behind undocumented internal compiler switches that only some employees at Dynasim know about), because they only confused their engineering end users. One should never ask users to supply pieces of information that they do not understand, because otherwise, they get frustrated and walk away to another supplier who does not ask questions that they don't know how to answer. Offering a palette of different algorithms is fine, but only as long as the M&S environment is smart enough to figure out on its own, which of these methods to use under what conditions.

Neither Dymola nor any other Modelica implementation has been successful in entering the market for simulations of distributed parameter systems. Although the method of lines (Cellier and Kofman, 2006) makes it easy to convert partial differential equations (PDEs) to sets of ODEs, the resulting ODE models cannot be simulated efficiently using Dymola. The reason is that PDEs, unless we are dealing with toy problems, invariably lead to ODE models comprised of hundreds if not thousands of ODEs, and Dymola is not currently capable of simulating such ODE systems efficiently, at least not if the resulting equation systems are stiff.

As the step from simple component level models involving half a dozen ODEs to device level models containing several dozens of ODEs opened up a large set of new questions and called for drastically different solutions, the same happens once again when we proceed from several dozens of ODEs to several hundreds or possibly thousands of ODEs, i.e., when we advance from device level models to system level models. These models call once again for drastically different solutions, and that is what this paper is all about.

2 SYSTEM LEVEL MODELING

We are building ever more complex systems. The first Airbus 380 was delivered to the customer with a delay of roughly one year, because the producers of the aircraft were not capable of testing the aircraft in time in all of its potential modes of operation. The overall system has become too complex. The electrical system alone is of frightening complexity. The Airbus 380 cable tree is 500 km long and features tens of thousands of connections. If we are no longer capable of testing such an aircraft in all of its potential modes of operation before handing it over to the customer, we should at least be able to simulate it using a realistic full-scale model of the aircraft capturing all features of the vehicle using realistically accurate mathematical models. Such a system level description will definitely feature many hundreds of state variables.

An air-tower training simulator should be able to simulate not only one aircraft, but the entire fleet of air vehicles moving about the airport. As airplanes arrive, their models should become successively more complex on their own as they approach touch-down, whereas the models of departing aircraft should become simpler as they disappear from the vicinity of the airfield. Some pilots may be humans in training, whereas other pilots are algorithms that form part of the simulation. The same goes for the air traffic controllers. We need to simulate simultaneously the motion of the air vehicles, the communication channels, and even weather patterns, such as wind velocity and direction, and we should feed the visible scenery in the form of optical information to the air tower controllers training in the air-tower simulator as well as to the pilots training in the aircraft simulator that is located next to the air-tower simulator.

The human body is incredibly complex. At the current time, we are barely capable of simulating one single organ in isolation with a halfway satisfactory degree of realism. Now the call is out for full-body simulations. This is the only way how we can begin to understand how our metabolism deals with drugs, for example; which side effects are being produced in the process. When a surgeon opens up a patient, the body reacts violently and in many different ways to the intervention. The heart beat rises and the blood pressure may either rise or fall rapidly. Hormones are being produced and dispatched by many organs in parallel. When difficult interventions need to be made, an anesthesiologist supports the surgeon and tries to counteract the reactions of the patient, keeping the vital signs of the patient within safe bounds. Yet, the anesthesiologist operates in a strictly reactive mode with only limited understanding of what happens inside the patient. It is only his or her experience with similar cases that ensures that he or she will take the correct decisions quickly and reliably. Full-body simulations may help to improve this situation without putting a real patient at risk, but we are still far from being able to offer realistic full-body simulations. What we know for a fact is that these models will invariably be of high complexity and feature hundreds if not thousands of state variables.

Object-oriented modeling scales generally well. If we knew all of the equations that we need, we could probably encode such a complex model in Modelica without too much difficulties. Yet, even Dymola, the most advanced of the current Modelica implementations, would fail miserably in dealing with any of these models both at compile time and at simulation time. We shall now discuss why this is the case.

3 VARIABLE STRUCTURE MODELS

One of the models proposed above mentioned the need to simplify aircraft models on the fly. Currently, Modelica does not offer a convenient way of describing the exchange of models at event times, and Dymola is incapable of compiling and simulating such models, even if they have been described in an indirect fashion. Neither Modelica nor Dymola are currently designed to handle variable structure models. This is a major shortcoming of Modelica, as variable structure systems are quite common. Any mechanical system with a clutch is a variable structure system. When the clutch is engaged, the two axles connected by the clutch move in unison, and consequently, the total number of differential equations is smaller than when the clutch is disengaged and the two axles move separately one from the other.

Dirk Zimmer recently proposed a remedy to this shortcoming in his Ph.D. dissertation (Zimmer, 2010). He designed a variant of the Modelica language, that he called Sol, that allows to formulate models that can be activated and deactivated and even duplicated at event times. He offered a prototype implementation of a Sol compiler and a prototype run-time environment to simulate the compiled Sol models.

Yet, this is only a first step. In the example of the simplified aircraft models, for example, it should not even be necessary to manually code aircraft models of different complexity and encode the conditions under which a model change is to occur. In principle, it should suffice to encode the most complex aircraft model and leave it up to the compiler to generate simplified versions thereof on its own using automated model pruning techniques (Sen et al., 2009). Tests could be automatically generated that look at the dynamics of the trajectories produced by the simulation, and when the fast transients have died out, automatically switch to a simplified version of the model.

4 MODEL DEBUGGING SUPPORT

Anyone who has already modeled a decently complex system using Modelica knows that debugging a Modelica model can be quite difficult. Coding a model is easy, but when the compiler tags the model as singular, meaning that the number of equations does not equal the number of unknowns, locating the error in the model can be a challenge.

Tools were developed that help us reduce the likelihood of obtaining cryptic compiler error messages, e.g. by balancing the model connectors (Olsson et al., 2008). Yet the problems are still formidable. This happens in today's device level models. If the size of the model grows once again by one order of magnitude, as this will invariably happen when we proceed from device level simulations to system level simulations, modeling errors may occur much more frequently, and finding them just by looking at the code may become a hopeless undertaking. We shall need help in this task (Pop et al., 2012).

One possible approach might be to invoke automated model pruning. When the compiler tags a model as singular, it automatically simplifies one submodel after another. If the singularity disappears in the process, the error is most likely to be found in the device model that has just been simplified. Thus, the compiler can offer help in better reporting compile-time errors.

5 MODEL CONNECTION COMPATIBILITY SUPPORT

Not all models can work correctly with one another. The trick used often today to avoid that incompatible models get connected to each other is to make their connectors incompatible. If a plug doesn't fit into a socket, that is a good indication that the device behind that plug should not get connected to the system at this location.

However, the compatibility issues are often more subtle. We need to be able to check that a variable never assumes a negative value, for example. It could also be that a connection between two models is only valid for positive power flow from the first model to the second, but not when the power flow is reversed. It may be that a motor model is only valid up to a certain maximum rotational speed or a maximum level of armature voltage. Modelica offers already now assert statements that can be used to catch various types of incompatibility issues, but many users are hesitant to use assert statements on variables rather than parameters, because these will slow down the simulation.

Also in the context of variable structure models, the current assert statements may not be powerful enough. We must be able to encode under what conditions we are allowed to switch from one model to another, for example.

Finally in the context of automated model pruning, it is not clear how assert statements need to be adjusted in the process of model pruning. This issue needs to be studied.

6 MODEL PLAUSIBILITY SUPPORT

When checking the results obtained from simulating a model, we usually only look at those few variables that are most important to us. If the trajectories generated by the model look plausible, we accept the model as probably correct. However, it may happen that the simulation generates entirely implausible trajectory behavior for some other variables that we failed to look at. We would have rejected the model had we looked at any of those trajectories.

The problem gets worse as the size of the model increases. In a system level simulation, we shall still not have time to look at more than a handful of trajectories. Thus, we need language support to prevent implausible trajectory behavior from remaining unnoticed. It should be possible to specify behavioral patterns for variables that we expect them to exhibit. If the simulations fail to generate the predicted patterns, these variables should get flagged by the simulation so that the modeler will look at the generated trajectories explicitly before accepting the model as being plausible.

A similar feature is also needed for fault detection. In a watchdog monitor (Cellier et al., 1992; de Albornoz, 1996), real-time simulation results are compared with measurement data obtained from the monitored plant. If the simulated trajectories deviate from the observed behavior in significant ways, an alarm is issued that alerts the plant operator to a possible fault in the system (Escobet et al., 1999). In this case, the model had been validated before, and it is the plant that is expected to be at fault rather than its model.

7 THE CURSE OF LARGE-SCALE SYSTEM SIMULATION

Models of large-scale systems are invariably stiff. Among the state variables simulated, some will always change faster than others. This is particularly true for multi-energy domain simulations. For example in a mechatronic system, the electrical time constants are always faster than the mechanical ones by about two orders of magnitude. If the thermal domain is to be studied as well, thermal states will have time constants that are slower than the mechanical ones again by one or two orders of magnitude. Thus, such multi-energy domain models are always stiff.

ODE solvers that can be used to simulate stiff systems must be implicit solvers. Explicit solvers will need to use step sizes throughout the simulation that are small in comparison with the fastest time constants captured by the model as otherwise the simulation will become numerically unstable (Cellier and Kofman, 2006). The numerical stability domains (Cellier and Kofman, 2006) of stiff system solvers loop in the right-half complex $\lambda \cdot h$ plane. This avoids numerical stability problems during simulation when simulating analytically stable systems. All explicit solvers have numerical stability domains that loop in the left-half complex $\lambda \cdot h$ plane, and only some (but not all) implicit solvers feature numerical stability domains that loop in the right-half complex $\lambda \cdot h$ plane (Cellier and Kofman, 2006).

Implicit solvers (including all stiff system solvers, such as DASSL (Petzold, 1983) or Radau (Hairer and Wanner, 1999)) need to iterate over a number of iteration variables in each integration step using Newton iteration (Cellier and Kofman, 2006). The iteration variables usually include all state variables plus a few algebraic (so-called tearing) variables that are needed to break algebraic loops that remain in the model (Cellier and Kofman, 2006).

The simulation is formulated in the following way:

$$\mathcal{F}(\mathbf{z}_{k+1}) = 0 \tag{4}$$

where $\mathbf{z}$ is the vector of iteration variables to be calculated at the end of the next integration step, i.e., at time t_{k+1}, and $\mathcal{F}(.)$ is a set of zero functions that need to be solved for the unknown iteration variables. The Newton iteration can be formulated as follows:

$$\mathbf{z}_{k+1}^{l+1} = \mathbf{z}_{k+1}^{l} - \mathcal{H}^{-1} \cdot \mathcal{F} \tag{5}$$

where $\mathcal{H} = \partial\mathcal{F}/\partial\mathbf{z}$ is the Hessian matrix (Cellier and Kofman, 2006). We start out with initial guesses of the values of the iteration variables, $\mathbf{z}_{k+1}^{0}$. The $(l+1)^{st}$ iteration can then be computed from the l^{th} iteration using the above formula.

The Newton iteration can be set up as follows:

$$\mathcal{H} \cdot \delta z = \mathcal{F} \tag{6}$$
$$\mathbf{z}_{k+1}^{l+1} = \mathbf{z}_{k+1}^{l} - \delta z \tag{7}$$

Thus, inside each Newton iteration of a non-linear model, a set of linear equations needs to be solved during each iteration step. The convergence speed of the Newton iteration, i.e., the number of iterations needed for convergence, depends heavily on the accuracy with which the elements of the Hessian matrix are being estimated and on the accuracy with which the linear equations are being solved.

The Newton iteration is the crucial part of the simulation as this is where the simulation spends most of its time. A first improvement over the current state of the art would therefore be to invoke a linear system solver that exploits the multi-core architecture of modern computers. Also the estimation of the elements of the Hessian matrix can be easily parallelized. Different columns of the Hessian matrix can be computed in parallel on separate cores. These performance improvements have not been crucial up until now, because most Dymola models formulated today at device level simulate within a few seconds of real time anyway.

Unfortunately, the execution time needed for the Newton iteration, and thereby for the entire simulation, grows cubically in the number of iteration variables. Thus, if today's device level simulations take seconds to simulate, system level simulations that involve ten times as many iteration variables will take hours to simulate. This is why Dymola is not yet competitive when simulating distributed parameter models (Link et al., 2009).

The situation has recently improved by the addition of sparse matrix solvers to Dymola for dealing with large sets of linear equations (Braun et al., 2012). A speed-up of a factor of up to 20 was reported on some test problems. Yet, this is still insufficient to make Dymola competitive for the simulation of larger PDE models.

Future generations of computers will offer larger numbers of cores, and making use of a linear system solver and a Hessian generator that fully exploit the multi-core architecture can push the barrier yet a bit further, but this alone will not overcome the complexity issue. The simulation time still grows cubically in the number of iteration variables. For stiff system simulations to scale well, we shall need to get away from large Newton iterations.

The multi-core architecture can also be exploited in other ways. First attempts at distributing transmission line Modelica models over a multi-core architecture were recently reported by Sjölund (Sjölund et al., 2010). Sjölund writes in his article that it is very important that the distribution of the model over the architecture be accomplished in a fully automated fashion by the Modelica compiler. Otherwise the technique will become unmanageable when dealing with large-scale models. We shall explore this avenue further in the next section of this article.

8 QUANTIZED STATE SYSTEM SIMULATION

In recent years, a new class of ODE solvers has been developed that is based on state quantization rather than time slicing (Cellier and Kofman, 2006; Kofman, 2003). Rather than asking the question:

> What values will the state variables assume at the next sampling instant, t_{k+1}?

we ask the question:

> At what time will the state variable x_i change by more than $\pm\delta x_i$ in comparison with its current value?

A quantized state system (QSS) solver operates naturally in an asynchronous mode. Each state variable carries its own simulation clock. If the states of a subsystem change very little, the model equations capturing the dynamics of that subsystem will be executed rarely. In the context of QSS simulations, automated model pruning may become less of an issue and may in fact turn out to be unnecessary. A dormant model occupies space on the disk, but it does not slow down the simulation, as its equations will not get executed. As we proceed from device level simulation to system level simulation, the relative advantage of QSS solvers will become more noticeable, as they escape the curse of complexity.

QSS solvers can easily exploit a multi-core architecture. Since each state variable minds its own business, the solvers associated with the individual state variables can be easily distributed over multiple cores. This needs to be done intelligently in order to minimize inter-processor communication, but QSS solvers don't suffer the fate of the centralized stiff system solvers of the past. This is a research area that we are currently exploring (Ph.D. dissertation of Xenofon Floros to be defended in 2013). In order to solve the synchronization problem between the different simulators running on separate cores, each state variable synchronizes its own simulation clock with a scaled version of the real-time clock, but the different simulators don't synchronize their individual clocks among each other (Bergero et al., 2013).

Recently, first versions of stiff QSS solvers have been developed (Migoni et al., 2013; Migoni et al., 2012). Although stiff QSS solvers are necessarily implicit, they do not require Newton iteration. The reason is that the next state is known in advance. If the current state has a value of x_i, the next state will assume a value of $x_i \pm \delta x_i$. Thus in the worst case, we need to explore two alternatives. The average simulation step of a stiff QSS solver therefore consumes only about 20% more real time in comparison with its non-stiff twin.

These solvers nevertheless still carry the risk of a combinatorial explosion, because transitions in one state may trigger immediate transitions in other variables as well. Thus in the worst case, we may need to check for 2^n potential next states. To prevent combinatorial explosion, the current generation of stiff QSS solvers has been implemented in a linearly implicit fashion only (Migoni et al., 2013). This often works well, but if the stiffness of the model is not caused by large and small elements on the diagonal of the Hessian matrix, e.g. in discretized parabolic PDEs, the "stiff" QSS solvers lose their stiffness property, and therefore, stiff QSS solvers are not yet as robust as e.g. DASSL. More research in this area is still needed. Distribution over multiple cores may help also in this respect. Since the different columns of the Hessian matrix can be computed in parallel, it should be possible to also limit the combinatorial explosion in similar ways.

First results of simulating large-scale models using stiff QSS solvers were recently published (Bergero et al., 2012). We simulated a long transmission line (not a stiff system) with a load that made the overall system very stiff. Thus, Dymola, using centralized simulation, has no choice but to make use of a stiff system solver on all state variables and suffers the consequences of a large Hessian matrix. The stiff QSS solver simulated this system 1000 times faster than Dymola, even without exploiting the multi-core architecture, which is the type of performance that we need when proceeding from device level to system level simulation.

First simulation results using parallel implementations of stiff QSS solvers distributed over a multi-core architecture showed very promising results as well (Bergero et al., 2013). Parallel implementations of QSS solvers scale very well. Whereas most parallel implementations of dynamic system simulations saturate quickly as more cores are being added (due to the overhead associated with inter-processor communication), we did not notice any saturation effects yet when distributing the transmission line simulation over 12 parallel processes (six cores operating each in an interleaved mode).

In fact, the speed-up was even slightly faster than linear, because the overhead associated with managing the event queue currently grows quadratically in the number of events being managed (it could grow with $n \cdot log(n)$, but the event handler has not yet been implemented in an optimal fashion in the current release of the tool). As each processor carries its own event queue, the average length of the individual event queues is smaller in the parallel setup, which explains the faster-than-linear speed-up.

QSS solvers are also more efficient than classical solvers in handling discontinuities as they do not require an iteration to localize event times. Power-electronic circuits with high frequency switching simulate currently about 20 times faster using QSS solvers than using DASSL. However,also this feature will become more important when proceeding from device level simulation to system level simulation. Event times (discontinuities) can be assumed to occur in a statistically independent fashion. Therefore we can assume that a model that is ten times as big features on average about ten times as many discontinuities per time unit. Thus, the event density grows linearly with the size of the model. The execution time of classical solvers grows quadratically with the event density, whereas that of QSS solvers grows only linearly with the number of events per time unit. This is true even for non-stiff models using non-stiff solvers for their simulation (Grinblat et al., 2012).

For all of the above reasons, we expect that system simulation tools of the future will employ decentralized multi-core implementations of QSS solvers as their default simulation algorithms, but the QSS solvers need to become more robust before this can happen.

9 CONCLUSIONS

In this article, it was shown that the demands on a modeling and simulation environment vary widely with the complexity of the systems to be modeled and the size of the models to be simulated. Modern M&S environments are useful for modeling devices and for simulating device level models, but they are not yet useful for modeling complex systems and for simulating system level models. A series of avenues were sketched outlining how some of the shortcomings of current M&S environments may be overcome.

ACKNOWLEDGMENTS

This research has in part been funded by CTI grant No. 12101.1;3 PFES-ES and supported by the OPENPROD-ITEA2 project. The support is gratefully acknowledged.

REFERENCES

Bergero, F., Floros, X., Fernandez, J., Kofman, E., and Cellier, F. (2012). Simulating Modelica Models with a Stand-alone Quantized State Systems Solver. In *Proceedings 9th International Modelica Conference*, pages 237–246, Munich, Germany.

Bergero, F., Kofman, E., and Cellier, F. (2013). A Novel Parallelization Technique for DEVS Simulation of Continuous and Hybrid Systems. *Simulation*.

Braun, W., Gallardo-Yances, S., Link, K., and Bachmann, B. (2012). Fast Simulation of Fluid Models with Colored Jacobians. In *Proceedings 9th International Modelica Conference*, pages 247–252, Munich, Germany.

Cellier, F. (1979). *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*. PhD thesis, Swiss Federal Institute of Technology.

Cellier, F. (1991). *Continuous System Modeling*. Springer–Verlag, New York, NY, USA.

Cellier, F. and Elmqvist, H. (1993). Automated Formula Manipulation Supports Object–oriented Continuous System Modeling. *IEEE Control Systems*, 13(2):28–38.

Cellier, F. and Kofman, E. (2006). *Continuous System Simulation*. Springer–Verlag, New York, NY, USA.

Cellier, F., Schooley, L., Zeigler, B., Doser, A., Farrenkopf, G., Kim, J., Pan, Y., and Williams, B. (1992). Watchdog Monitor Prevents Martian Oxygen Production Plant from Shutting Itself Down During Storm. In *Proceedings International Symposium on Robotics and Manufacturing*, pages 697–704, Santa Fe, NM, USA.

de Albornoz, A. (1996). *Inductive Reasoning and Reconstruction Analysis: Two Complementary Tools for Qualitative Fault Monitoring of Large–scale Systems*. PhD thesis, Polytechnical University of Barcelona.

Elmqvist, H. (1978). *A Structured Model Language for Large Continuous Systems*. PhD thesis, Lund Institute of Technology.

Elmqvist, H., Cellier, F., and Otter, M. (1993). Object–oriented Modeling of Hybrid Systems. In *Proceedings ESS'93, European Simulation Symposium*, pages xxxi–xli, Delft, The Netherlands.

Elmqvist, H. and Otter, M. (1994). Methods for Tearing Systems of Equations in Object–oriented Modeling. In *Proceedings European Simulation Multiconference*, pages 326–332, Barcelona, Spain.

Escobet, A., Nebot, A., and Cellier, F. (1999). Model Acceptability Measure for the Identification of Failures in Qualitative Fault Monitoring Systems. In *Proceedings European Simulation Multi-Conference*, pages 339–347, Warsaw, Poland.

Grinblat, G., Ahumada, H., and Kofman, E. (2012). Quantized State Simulation of Spiking Neural Networks. *Simulation*, 88(3):299–313.

Hairer, E. and Wanner, G. (1999). Stiff differential equations solved by Radau methods. *Journal of Computational and Applied Mathematics*, 111(1-2):93–111.

Kofman, E. (2003). Quantization-based Simulation of Differential Algebraic Equation Systems. *Simulation*, 79(7):363–376.

Link, K., Steuer, H., and Butterlin, A. (2009). Deficiencies of Modelica and its Simulation Environments for Large Fluid Systems. In *Proceedings 7th International Modelica Conference*, pages 341–344, Como, Italy.

Mattsson, S., Elmqvist, H., and Broenink, J. (1997). Modelica – An International Effort to Design the Next Generation Modeling Language. *Journal A, Benelux Quarterly Journal on Automatic Control*, 38(3):16–19.

Mattsson, S., Olsson, H., and Elmqvist, H. (2000). Dynamic Selection of States in Dymola. In *Proceedings Modelica Workshop*, pages 61–67, Lund, Sweden.

Migoni, G., Bortolotto, M., Kofman, E., and Cellier, F. (2013). Linearly Implicit Quantization-based Integration Methods for Stiff Ordinary Differential Equations. *Simulation Modelling Practice and Theory*.

Migoni, G., Kofman, E., and Cellier, F. (2012). Quantization-based New Integration Methods for Stiff ODEs. *Simulation*, 88(4):387–407.

Mitchell, E. and Gauthier, J. (1986). *ACSL: Advanced Continuous Simulation Language – User Guide and Reference Manual*. Mitchell & Gauthier Assoc., Concord, MA, USA.

Modelica Association, . (2007). The Modelica Language Specification Version 3.0. Technical report.

Modelica Association, . (2008). The Modelica Standard Library. Technical report.

Olsson, H., Otter, M., Mattsson, S., and Elmqvist, H. (2008). Balanced Models in Modelica 3.0 for Increased Model Quality. In *Proceedings 6th International Modelica Conference*, pages 21–33, Bielefeld, Germany.

Otter, M., Elmqvist, H., and Mattsson, S. (1999). Hybrid Modeling in Modelica Based On Synchronous Data Flow Principle. In *Proceedings IEEE, International Symposium on Computer Aided Control System Design*, pages 151–157, Kohala Coast, Hawaii.

Pantelides, C. (1988). The Consistent Initialization of Differential–Algebraic Systems. *SIAM Journal of Scientific and Statistical Computing*, 9(2):213–231.

Petzold, L. (1983). A Description of DASSL: A Differential/Algebraic Equation Solver. In Stepleman, R., editor, *Scientific Computing*, pages 65–68. North–Holland, Amsterdam, The Netherlands.

Pop, A., Sjölund, M., Asghar, A., Fritzson, P., and Casella, F. (2012). Static and Dynamic Debugging of Modelica Models. In *Proceedings 9th International Modelica Conference*, pages 443–454, Munich, Germany.

Ryan, R. (1990). ADAMS – Multibody System Analysis Software. In Schiehlen, W., editor, *Multibody Systems Handbook*, pages 361–402. Springer-Verlag.

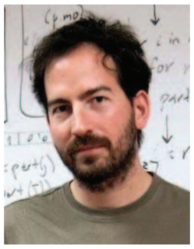
Sen, S., Moha, N., Baudry, B., and Jézéquel, J. (2009). Meta–model Pruning. In *Model Driven Engineering Languages and Systems*, pages 32–46. Springer–Verlag, Berlin and Heidelberg, Germany.

Sjölund, M., Braun, R., Fritzson, P., and Krus, P. (2010). Towards Efficient Distributed Simulation in Modelica Using Transmission Line Modeling. In *Proceedings 3rd International Workshop on Equation-based Object-oriented Languages and Tools*, pages 71–80, Oslo, Norway.

Tuinenga, P. (1995). *Spice: A Guide to Circuit Simulation and Analysis Using PSpice - Third Edition*. Prentice Hall, Saddle River, NJ, USA.

Zimmer, D. (2010). *Equation–based Modeling of Variable–structure Systems*. PhD thesis, Swiss Federal Institute of Technology.

François E. Cellier received his BS degree in electrical engineering in 1972, his MS degree in automatic control in 1973, and his PhD degree in technical sciences in 1979, all from the Swiss Federal Institute of Technology (ETH) Zurich. Dr. Cellier worked at the University of Arizona as professor of Electrical and Computer Engineering from 1984 until 2005. He returned to his home country of Switzerland and his alma mater in the summer of 2005. Dr. Cellier's main scientific interests concern modeling and simulation methodologies, and the design of advanced software systems for simulation, computer-aided modeling, and computer-aided design. Dr. Cellier has authored or co-authored more than 200 technical publications, and he has edited several books. He published a textbook on Continuous System Modeling in 1991 and a second textbook on Continuous System Simulation in 2006, both with Springer-Verlag, New York. He is a fellow of the Society for Modeling and Simulation International (SCS).



Xenofon Floros completed his undergraduate studies in the school of Electrical and Computer Engineering of the National Technical University of Athens in 2006. Since 2007, he is pursuing a PhD degree at ETH Zurich under the supervision of Proff. Joachim M. Buhmann and François E. Cellier on efficient numerical algorithms for the simulation of dynamical systems. His research interests include also computational biology and medical imaging.

Ernesto Kofman received the Electronics Engineering degree in 1999 and the Ph.D. degree in 2003, both from the Universidad Nacional de Rosario, Argentina. He currently works as an adjunct professor at the Universidad Nacional de Rosario, Argentina, and holds a tenured research position from the Argentine Research Council (CONICET). His research interests include continuous systems simulation and quantized and sampled-data control systems. He co-authored the textbook on Continuous System Simulation with Dr. Cellier.