[2] B. Char, K. Geddes, G. Gonnet, B. Leong, M. Monagan and S. Watt, *First Leaves: A Tutorial Introduction to Maple V*, Springer-Verlag, New York, 1992.

[3] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, 2nd ed., Addison-Wesley, Redwood City, CA, 1991.

[4] K. Khiar and E. A. Lee, "Modeling Radar Systems Using Hierarchical Dataflow," *Proc. of IEEE Int. Conference on Acoustics, Speech, and Signal Processing*, Detroit, MI, May 8-12, 1995, pp. 3259-3262.

[5] B. L. Evans, D. R. Firth, K. D. White, and E. A. Lee, "Automatic Generation of Programs that Jointly Optimize Characteristics of Analog Filter Designs," to appear: *Proc. of European Conference on Circuit Theory and Design*, August 27-31, 1995, Istanbul, Turkey.

[6] G. P. Fettweis and L. Thiele, "Algebraic Recurrence Transformations for Massive Parallelism", *VLSI Signal Processing V*, pp. 332-341, Oct. 1992.

[7] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[8] Ptolemy Team, *Ptolemy 0.5 User's Manual*, Dept. of EECS, University of California, Berkeley, CA, 1994.

[9] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[10] M. M. Covell, *An Algorithm Design Environment for Signal Processing*, Ph. D. Thesis, Massachusetts Institute of Technology, Research Laboratory for Electronics Tech. Rep. 549, Cambridge, MA, Dec. 1989.

[11] B. L. Evans, R. H. Bamberger, and J. H. McClellan, "Rules for Multidimensional Multirate Structures," *IEEE Trans. on Signal Processing*, vol. 42, no. 4, pp. 762-771, April, 1994.

[12] C. Moler, J. Little, and S. Bangert, *MATLAB User's Guide*, The MathWorks Inc., Natick, MA, 1989.

[13] "Using the CPLEX Callable Library", CPLEX Optimization, Inc., Suite 279, 930 Tahoe Blvd., Bldg. 802, Incline Village, NV 89451-9436.

[14] A. Kalavade and E. A. Lee, "A Global Criticality/ Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem", *Proc. of CODES/CASHE, Third International Workshop on Hardware/Software Codesign*, Grenoble, France, Sept. 22-24, 1994, pp. 42-48.

[15] A. Kalavade, J. L. Pino, and E. A. Lee, "Managing Complexity in Heterogeneous System Specification, Simulation, and Synthesis", *Proc. of Int. Conference on Acoustics, Speech, and Signal Processing)*, Detroit, MI, May 8-12, 1995, pp. 2833-2836.

Design flows are specified as graphical netlists. Conditionals and iterations are supported in the flow definition. Flows can be described hierarchically. A tools is encapsulated within a Star — a basic block in Ptolemy. Tool encapsulation involves writing scripts that call various programs, possibly on remote file systems. Tools can invoke programs with independent graphical user interfaces. The tool writer need not worry about the underlying timestamps and filenames. Data attributes and flow netlists are stored within the internal Ptolemy database.

DesignMaker supports both data-driven and demand-driven flow execution. It resolves tool dependencies and automatically invokes tools. The underlying scheduler is a combination of dynamic dataflow and event-driven schedulers. *DesignMaker* detects deadlocks when iterative flows are incorrectly specified (such as forward dependencies on non-optional inputs). *DesignMaker* also supports other features such as annotation of the design process and display of the flow status (blocked, running, and ready) at any point in the design process.

The design flow in Figure 4 can be specified in the DMM domain and managed by the *DesignMaker.*

5. CONCLUSION

System design is concerned with capturing all aspects of the design of a working system. The system as well as the design process can be highly complex and heterogeneous. The design process should ideally handle systems composed of a hierarchy of sub-systems, in which each sub-systems may operate under a different computational model and may be implemented by a wide variety of hardware and software technologies. The design process should support modeling, simulation, and synthesis of heterogeneous systems.

This paper examines some of the roles that symbolic computation plays in assisting system simulation and design. At a behavioral level, symbolic computation can compute parameters, generate new models, and optimize parameter settings. At the synthesis level, symbolic computation can work in tandem with synthesis tools to rewrite cascade and parallel combinations on components in sub-systems to meet design constraints. We have encoded the symbolic operations on signals and systems in the Signal Processing Packages for Mathematica. We are currently working on connecting Mathematica to the Ptolemy environment to complement its behavioral-level modeling. We are also working on connecting Mathematica to DesignMaker to assist in the synthesis of systems.

Symbolic computation represents on type of tool that may be invoked in the iterative, recursive, and otherwise complex flow of the system design process. The paper also discusses the qualities that a formal infrastructure for managing system design should have. The paper describes an implementation of this infrastructure called DesignMaker, implemented in the Ptolemy environment, which includes the ability to manage the flow of tool invocations in an efficient manner using a graphical file dependency mechanism.

6. ACKNOWLEDGMENT

The Ptolemy project is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), Semiconductor Research Corporation (project 95-DC-324), National Science Foundation (MIP-9201605), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, Mentor Graphics, Mitsubishi, Motorola, NEC, Pacific Bell, Philips, and Rockwell. World Wide Web information about the Ptolemy Project is located at URL `http://ptolemy.eecs.berkeley.edu`.

7. REFERENCES

[1] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems”, *Int. Journal of Computer Simulation*, special issue “Simulation Software Development,” Apr. 1994, vol. 4, pp. 155-182.

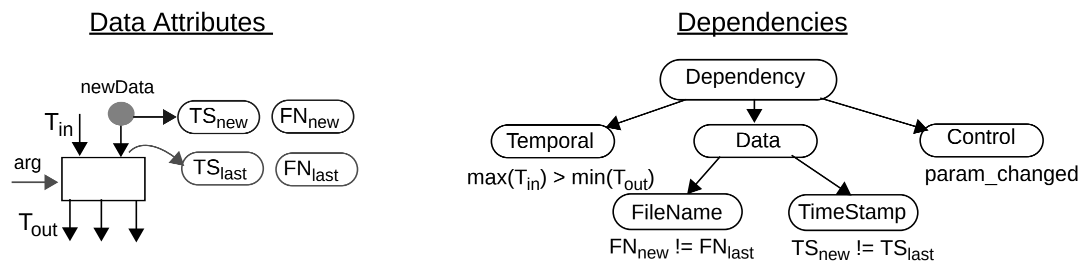


Figure 6. Dependencies used for tool management.

The design flow is represented as a distributed data structure. Associated with a tool is a flag (`Param_Changed`) that gets set when parameters of the tool change. Data associated with each port has three attributes: `File_Name`, `Time_Stamp`, and `Optional_Flag`. `File_Name` and `Time_Stamp` represent the filename and the timestamp of the data, associated with the most recent invocation of the tool. If the `Optional_Flag` is set on a port, the tool can be invoked even if data is not present on that port. Ports on which the `Optional_Flag` is set are called *optional* ports. Optional ports make it possible to represent conditionals and iterations in flows.

Dependencies

Figure 6 shows the three types of dependencies supported in DMM. *Temporal* dependencies track the timestamps on the input and output ports of tools. A tool is run if its output data is out-of-date, i.e., if any of its input data timestamps are newer than its output data timestamps. *Data* dependency ensures that a tool is run whenever the file received on its input ports has either a filename or a timestamp that is different from the previous tool invocation. *Control* dependency tracks parameter changes; a tool is run if its parameters are modified.

Flow Management

Automatic flow invocation involves analyzing tool dependencies and invoking tools if required. A tool is said to be *enabled* when all its non-optional input ports have data. Once enabled, its dependencies are examined. A tool is *invoked* (*run*) when at least one of its dependencies is alive. Two types of flow invocation mechanisms are desired: data-driven, and demand-driven (Figure 7). In the data-driven approach, the flow scheduler traverses the flow according to precedences. The process halts when all tools with live dependencies are exhausted. In the demand-driven mode, the user selects a tool for invocation. The scheduler traverses the predecessors and runs all tools with live dependencies on the path.

4.3. Implementation features

The DMM framework described in Section 4.2 has been implemented as a domain in the Ptolemy environment. The flow manager is called *DesignMaker.*

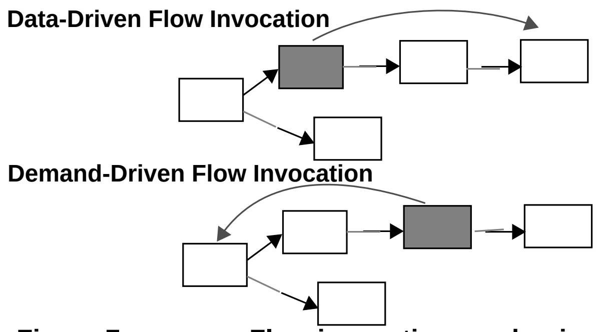


Figure 7. Flow invocation mechanisms.

- Modular and configurable flow specification mechanisms

  For example, in Figure 4, the user might be interested in first developing the algorithm, not in the final implementation. At this point only the *Parameter Generation* and *Model Generation* tools need to be invoked; subsequent tools need not be run. Unnecessary tool invocations can be avoided if the flow specification is modular.
  A number of design options is available at each step in the design process. For instance, the type of algebraic transformation could be a parameter to the *Algebraic Transformation* tool. A particular set of transformations along with their sequence could be selected by the user. Similarly, the *Hardware/Software Partition* tool can be one of: a human-intervened manual partitioning, an exact but time consuming tool such as CPLEX [13] based on integer linear programming techniques, or an efficient heuristic such as GCLP [14]. Depending on the available design time and desired accuracy, one of these is selected. This selection can be done either by the user, or by embedding this design choice within the flow. A design flow with a configurable flow specification mechanism is easily extensible.
- Mechanisms to track tool dependencies and automatically determine the sequence of tool invocations
  The user should not have to keep track of tools that have already run and those that need to be run. Also, if a specific tool is changed on the fly, the entire system need not be re-run; only those tools that are affected should be run. A mechanism that automatically determines the sequence of tool invocations is needed. For instance, if the user changes the hardware synthesis mechanism (perhaps a standard-cell based design instead of one based on field programmable gate arrays), the system should only invoke the dependencies of the hardware synthesis tool (in this case: *Netlist Generation).*
- Managing consistency of data, tools, and flows

  Detecting incompatibilities between tools and maintaining versions of the tools and design flows is necessary.

4.2. Framework for design methodology management

A design methodology specifies a sequence (flow) of tools that operate on data. Design methodology management (DMM) is defined as "definition, execution, and control of design methodologies in a flexible and configurable way" [15]. Flow definition, dependency analysis, and automated flow execution are the critical issues in DMM. In this section we describe the components of our DMM framework.

Flow, Tool, and Data Model

Figure 5 illustrates the key components of our DMM framework. A design *flow* is specified as a directed graph, where nodes represent tools, and arcs specify the ordering between tools. *Tools* encapsulate actual programs; tool parameters specify the arguments to these programs. A tool exchanges filenames with other tools via its *ports*.

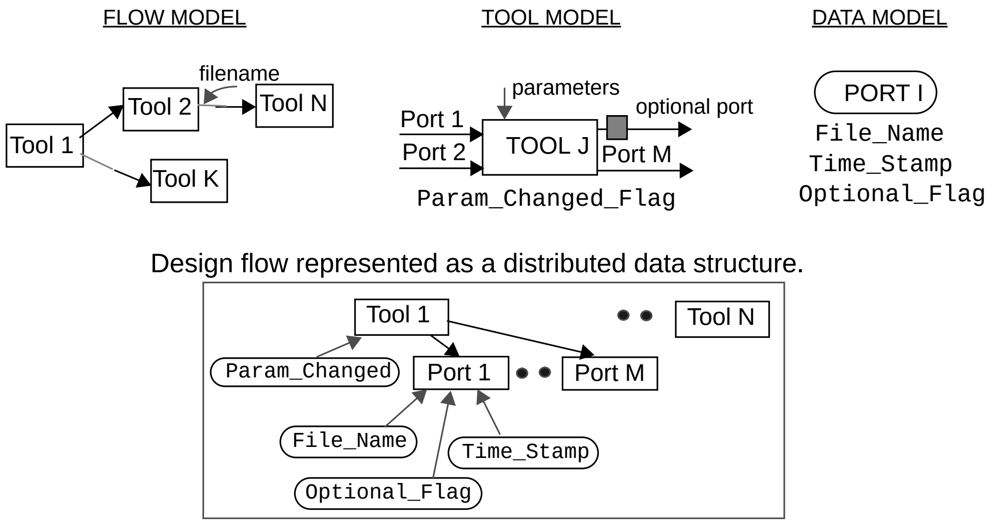


Figure 5. Components of our DMM system.

ger matrices using an algebraic transformation called Smith form decomposition (analogous to eigendecomposition). These kinds of algebraic operations are readily available in a symbolic computation environment such as Maple or Mathematica.

Like transformation rules at a fine level of computational granularity, transformation rules at a coarse level of computational granularity rewrite components in an algebraically equivalent, but not necessarily numerically equivalent, manner. When transformation rules at a coarse level are applied to improve the performance of a sub-system's implementation, they could interfere with the transformation rules applied at a fine level. To avoid interference, the mechanism applying the transformation rules should base its decisions about what rules to apply on feedback from the tools that will generate the implementation. In other words, the objective function being minimized by applying coarse-grain transformation rules should be based on accurate measures of implementation costs. The design space of alternate implementation could be searched by hooking up a symbolic computation engine that can apply transformation rules to a tool that manages the conversion of system designs into implementations. We have encoded a variety of transformation rules in Mathematica and are working on connecting the rewriting facility to the design methodology management tool described in the next section.

4. DESIGN SPACE EXPLORATION

A typical design flow for algorithm-level design is shown in Figure 4. A number of tools operate on the original algorithm specification to generate the final implementation. These include tools for algorithm design, simulation, and synthesis. The algorithm design process could involve one or more of the algebraic computations discussed in Section 2, such as parameter generation and model optimization. The system can be simulated at different levels of abstraction. Algebraic transformations, such as those described in Section 3, may be applied to optimize the design before synthesis. Finally, a number of hardware and software synthesis tools may be used to implement the system.

4.1. Requirements for efficient design space exploration

System-level design is not a push-button design process. It requires exploration of the design space and involves considerable user interaction. Managing the complexity of this design process is a hard problem. The features needed for efficient design space exploration include:

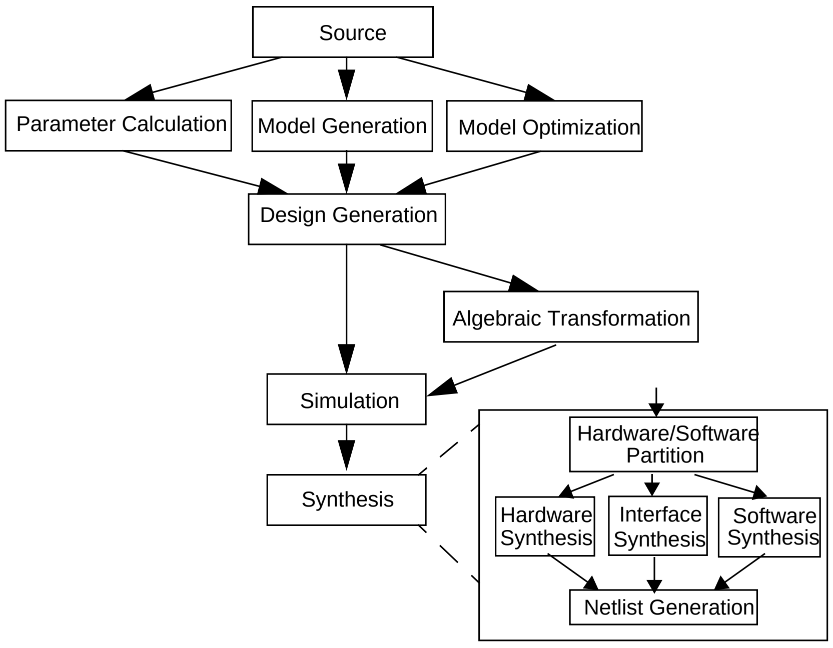
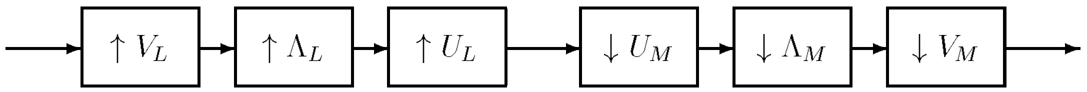


Figure 4. A typical design flow.

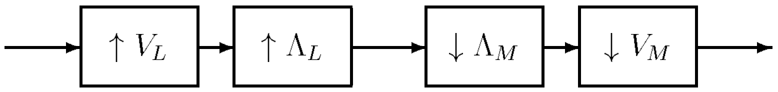
would be to perform the resampling in stages of resampling by 3:2, 7:5, and 7:16, the factors of 147 and 160 [8]. (A demonstration by Tom M. Parks of the CD-to-DAT rate changer exists in the Ptolemy environment [8].)

Transformation rules for rewriting cascades and parallel combinations of signal processing structures have been developed for one-dimensional operators (e.g. [7], [9], and [10]) and multidimensional operators (e.g. [7] and [11]). Transformation rules are expressed solely in terms of algebraic properties of operators, or are based on the mathematical definition of specific operators. The former transformation rules rely on algebraic properties such as additivity, associativity, and commutativity. The latter transformation rules depend on the mathematical definition of each specific operator and how that operator interacts with other operators. When a new operator is added, the transformation rules based on algebraic properties do not have to be updated, but new transformation rules must be defined describing those interactions with other operators not captured in terms of algebraic properties. An example of the former is that the order of any two linear time-invariant operators in cascade can be switched, and an example of the latter is that upsampling by N followed by downsampling by N is the identity system.

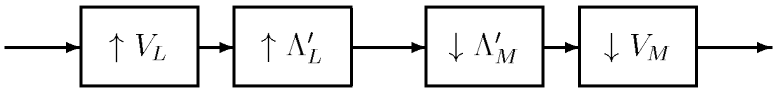
The applicability of those transformation rules that are based on the mathematical definition of operators may depend on sophisticated algebraic operations. Figure 3 shows an example of a few multidimensional transformation rules. The applicability of these rules depends on the decomposition of the downsampling and upsampling integer matrices into simpler inte-



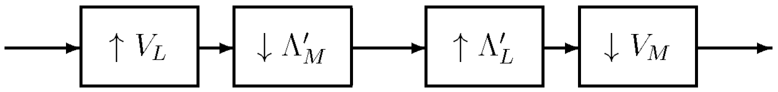
(a) Cascade in Smith form



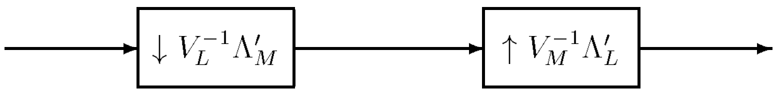
(b) Simplified cascade *if* $U_M = U_L$



(c) Simplified cascade from (b) with common factors removed



(d) Reversing the order of operations in (c) since Λ'_L and Λ'_M are coprime


$\downarrow V_L^{-1}\Lambda'_M$
$\uparrow V_M^{-1}\Lambda'_L$

(e) Combining operations in (d) so that downsampling comes first

Figure 3. Multidimensional transformation rules for a upsampler by integer matrix L in cascade with a downsampler by integer matrix M. [11]

and β is a constant that must be calculated for each set of pole locations. For optimal filter design, the poles will be spread out in the *s*-plane and duplicate poles will not occur, so A_k can be written in a closed differentiable form. The peak time can be substituted in the step response to compute peak overshoot. We programmed these definitions in the symbolic algebra program Mathematica.

With these definitions, we can now experiment with a variety of cost functions and numerical optimization procedures. In [5], we matched the analog filter design problem to a sequential quadratic programming (SQP) problem. SQP requires that the objective function and constraints are twice differentiable, and works best when the gradients of the objective function and constraints are explicitly computed and given to the program. We then programmed Mathematica to

- verify the differentiable real-valued formulas given above,
- define an objective function and constraints,
- compute the gradient of the objective function and constraints symbolically,
- convert the objective function, constraints, and their gradients into MATLAB [12] source code,
- generate a MATLAB script to run the optimization procedure.

The optimization procedure is based on SQP routine, `constr`, in the MATLAB Optimization Toolbox. We used `constr` to verify the generated MATLAB code for the gradients of the objective function and the constraints.

Using our procedure, we optimized a fourth-order all-pole lowpass Butterworth filter in terms of its peak overshoot and deviation from linear phase response in the passband. The optimization essentially trades off magnitude response for lower peak overshoot and a more linear phase response, while keeping the magnitude response within specification. For the optimized filter, the peak overshoot was reduced by 50% (from 16% to 8%) and the phase response in the passband became nearly linear. The objective function used was continuous, real-valued, non-negative, and twice differentiable. It had an initial value of 1.17 which almost entirely due to the peak overshoot, and a final value of 4.7×10^{-5}, of which ninety-seven percent of its value reflected the deviation from linear phase. Because we can compute the final value of the gradients, we could determine that the one of the second-order sections was twice as sensitive to perturbations than the other. [5]

3. TRANSFORMATIONS BASED ON ALGEBRAIC IDENTITIES

The previous section discussed symbolic computation at the behavioral level of system design. Symbolic computation can also be used at the synthesis level by applying transformations based on algebraic identities. At a fine grain level, for example, single-step recurrence recursive computations involving one *associative* binary operation (e.g. addition) can be transformed into an algebraically equivalent M-step recursion. The M-step recursion is not necessarily numerically equivalent to the original recurrence equation, but it has M degrees of parallelism compared to one degree of parallelism in the original recurrence relation. That is, the throughput can be increased by a factor of M, or alternatively, the clock rate can be reduced by a factor of M. This strategy is used in carry look-ahead adders. The approach works for any associative binary operator, and has been applied to multiplication and addition operations in linear filters, and the min/max and addition operators in the Viterbi algorithm and dynamic programming [6].

In this section, we discuss applying transformation rules based on algebraic identities at a coarse level, i.e., at the level of upsample and filter operators. For example, in one-dimensional sampling rate changing systems, one can rewrite the cascade of an upsampler, filter, and downsampler in polyphase form so that the filtering is performed at the input sampling rate instead of at the higher upsampled rate [7]. The polyphase form minimizes the number of addition and multiplication operations in the resampler. Transforming a rate changer to polyphase form involves exploiting several algebraic identities of the operators in the cascade. The single stage polyphase form, however, may not always be the best solution. If the upsampling factor L or downsampling factor M is a large integer, then it would become nearly impossible to design a proper lowpass anti-aliasing and anti-imaging filter, because the filter's cutoff frequency is min(π/L, π/M). For example, converting compact disc music sampled at 44.1 kHz to digital audio tape sampled at 48 kHz in one stage would require a resampling ratio of 147:160, and a narrowband lowpass digital filter with a passband of 0 to 20 kHz and a stopband of 24 to 3528 kHz. The better approach the Laplace representation below. From the Laplace representation, we derive the magnitude, phase, and step responses written as real-valued differentiable functions.

$$G(s) = \prod_{l=1}^{r} \frac{(s-z_l)\,(s-z_l^*)}{|z_l|^2} \cdot \prod_{k=1}^{n} \frac{|p_k|^2}{(s-p_k)\,(s-p_k^*)}$$

We will now define the magnitude response, phase response, quality factors, and peak overshoot based on the transfer function. We will derive these quantities as differentiable real-valued functions of real-valued variables to take advantage of a wider variety of optimization procedures. As a differentiable real-valued function, the magnitude is given by the following formula, in which we have factored the polynomials under the square root sign into Horner's form because of its better numerical properties:

$$|G(j\omega)| = \prod_{l=1}^{r} \frac{\sqrt{\left(\omega^2 + 2\,(c_l - d_l)\right)\omega^2 + \left(c_l^2 + d_l^2\right)^2}}{c_l^2 + d_l^2} \cdot \prod_{k=1}^{n} \frac{a_k^2 + b_k^2}{\sqrt{\left(\omega^2 + 2\,(a_l - b_l)\right)\omega^2 + \left(a_l^2 + b_l^2\right)^2}}$$

The unwrapped phase response expressed as a real-valued differentiable function is

$$\angle G(j\omega) = -\sum_{l=1}^{r} \left(\operatorname{atan}\left(\frac{\omega - d_l}{c_l}\right) + \operatorname{atan}\left(\frac{\omega + d_l}{c_l}\right)\right) + \sum_{k=1}^{n} \left(\operatorname{atan}\left(\frac{\omega - b_k}{a_k}\right) + \operatorname{atan}\left(\frac{\omega + b_k}{a_k}\right)\right)$$

The quality factor for each complex conjugate pole pair measures how close the pole is to the $j\,\omega$ axis. The lower the quality factor, the less likely that the pole will cause oscillations in the output (e.g., as a response to noisy input). The quality factor Q_k for one complex conjugate pole pair and the effective quality factor Q_{eff} for the entire filter are given below:

$$Q_k = \frac{\sqrt{a_k^2 + b_k^2}}{-2a_k} \qquad Q_{eff} = \left(\prod_{k=1}^{n} Q_k\right)^{\frac{1}{n}} \qquad Q_k, Q_{eff} \ge 0.5$$

The time at which the peak value of the step function occurs can be approximated by the formula:

$$t_{peak} = \beta \frac{1}{n} \sum_{k=1}^{n} t_{peak_k} \qquad t_{peak_k} = -\left(\frac{1}{b_k} \cdot \operatorname{atan}\left(\frac{\gamma_k \cdot b_k}{a_k^2 + \gamma_k a_k + b_k^2}\right)\right) \qquad \gamma_k = \frac{C_k}{D_k}\left(a_k^2 + b_k^2\right)$$

where

$$C_k = 2|A_k|\cos(\angle A_k) \qquad D_k = -2|A_k|\,(a_k \cos(\angle A_k) + b_k \sin(\angle A_k)) \qquad A_k = [G(s)\,(s-p_k)]_{s = p_k}$$

input power constrained to be less or equal to P_{av}. Assuming that the input to the discrete-time channel has zero mean, the optimization problem becomes

$$\underset{f_Y(y)}{max} \; H(Y) \;=\; \underset{f_Y(y)}{max} \int_{-\infty}^{\infty} f_Y(y)\,(\log_2(f_Y(y)))\,dy$$

subject to the constraints

$$\int_{-\infty}^{\infty} f_Y(y)\,dy \;=\; 1 \qquad \int_{-\infty}^{\infty} y^2 f_Y(y)\,dy \le P_{av} + \sigma^2$$

Applying Calculus of Variations, the resulting probability density function (pdf) and channel capacity, respectively, are

$$f_Y(y) \;=\; \frac{1}{\sqrt{2\pi(P_{av}+\sigma)}}\; e^{\frac{y^2}{2(P_{av}+\sigma)}} \qquad C_s \;=\; \frac{1}{2}\;\log_2\left(1+\frac{P_{av}}{\sigma^2}\right)$$

The algebraic computations in this example yield the formula for the pdf of the output of the channel. We can then convert the pdf into a behavioral model that outputs random numbers according the pdf. From the pdf, we compute the channel capacity as a formula. Any system that uses these calculations should be linked to the algebraic calculations so that if they change, then the new values would be automatically propagated. The algebraic calculations should, in turn, be linked to the choice of the channel. Now, if a different channel model is used, then a new pdf and channel capacity would be automatically computed, and any systems depending of these calculations would be updated.

2.3. Optimization of Behavioral Models Governed by Mathematical Equations

When the behavior of a sub-system or a component of a sub-system can be described in terms of mathematical equations, symbolic computation tools may be used to manipulate the underlying equations into a form suitable for use by a numerical optimization program. Since the symbolic tool can perform the necessary algebra on the equations and then convert the equations to source code, the design procedure can be carried out automatically, and without mathematical errors and errors transcribing the formulas to source code. Furthermore, the procedure may allow a variety of alternative forms and numerical optimization techniques to be tried based on the same behavioral equations.

As an example, we will consider computing pole and zero locations of an analog lowpass filter to jointly optimize the filter for multiple characteristics subject to multiple types of constraints. In particular, we will jointly optimize for magnitude response, phase response, quality factors, and peak overshoot, subject to constraints on the magnitude response, quality factors, and peak overshoot [5]. The underlying representation of the behavioral model is a set of n complex conjugate poles $p_k = a_k \pm j\, b_k$ (such that $a_k < 0$) and r complex conjugate zeroes $z_l = c_l \pm j\, d_l$ (such that $r < n$ and $c_l < 0$ for all l). The underlying mathematical representation is the Laplace domain of the filter transfer function. We give the factored form of

2.1. Parameter Calculations

Many parameters in signal processing and communications systems are determined by algebraic computation. In a typical (binary) digital communications systems, the receiver output is the result of a matched filter, a sampler, and a threshold device. The optimal value of the threshold level depends on the detection method used (e.g. maximum likelihood), the mean values of the two transmitted pulses, and the probabilities of the occurrence of the two symbols. When the detection method is changed, the underlying algebra that solves for the closed form of the optimal threshold changes. Other calculations might involve indefinite integration, symbolic differentiation, differential equation solving, and calculus of variations. We would like to maintain a record of the algebraic calculations with the sub-system being designed.

A powerful application of symbolic parameter calculation is to compute parameters that depend on inherited parameters. The values of the inherited parameters are not known until the overall system is simulated or synthesized, so the inherited parameters are manipulated as symbolic quantities. This approach can add flexibility to components which are replicated, e.g. in parallel or cascaded combinations, a parameterized number of times. Figure 2 shows an example in the Ptolemy environment of a sub-system to generate a square wave based on a finite number of its Fourier components. The formula for the Fourier coefficients is fixed in the definition of the system as $4 / (\pi (2 i - 1))$, where i is the instance number of the replicated structure. The value of the instance number is set by the higher-order function when it replicates (maps) the functional gain block [4]. In this example, the signal being approximated is fixed in the definition of the system, and so is its Fourier series formula. With support for symbolic computation, the signal could be a parameter of the system, and the Fourier series can be computed when the sub-system is simulated or synthesized.

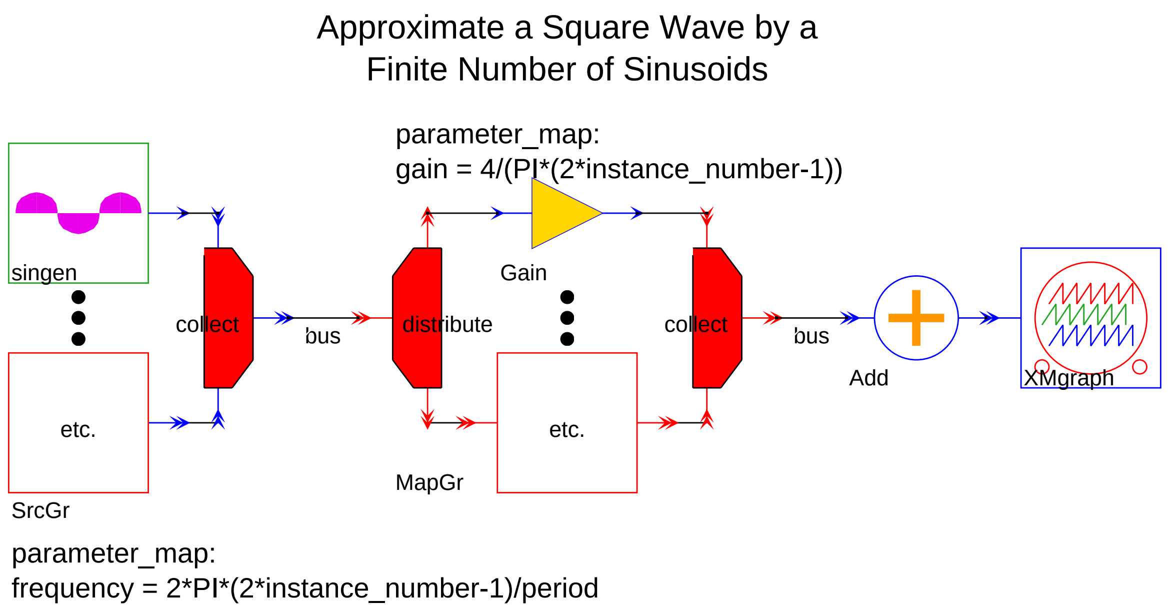

Approximate a Square Wave by a
Finite Number of Sinusoids
parameter_map:
gain = 4/(PI*(2*instance_number-1))
singen
collect
bus
distribute
Gain
collect
bus
Add
XMgraph
etc.
etc.
MapGr
SrcGr
parameter_map:
frequency = 2*PI*(2*instance_number-1)/period

Figure 2. **Square wave generation using a finite number of Fourier components.**

2.2. Model Generation

Symbolic calculation can be used to generate a complete model for a subsystem component. For example, we could derive a model for the output Y of a discrete-time channel operating at full capacity, given that the channel has continuous-valued inputs, corrupts the inputs with zero-mean additive white Gaussian noise with variance σ, and has its average

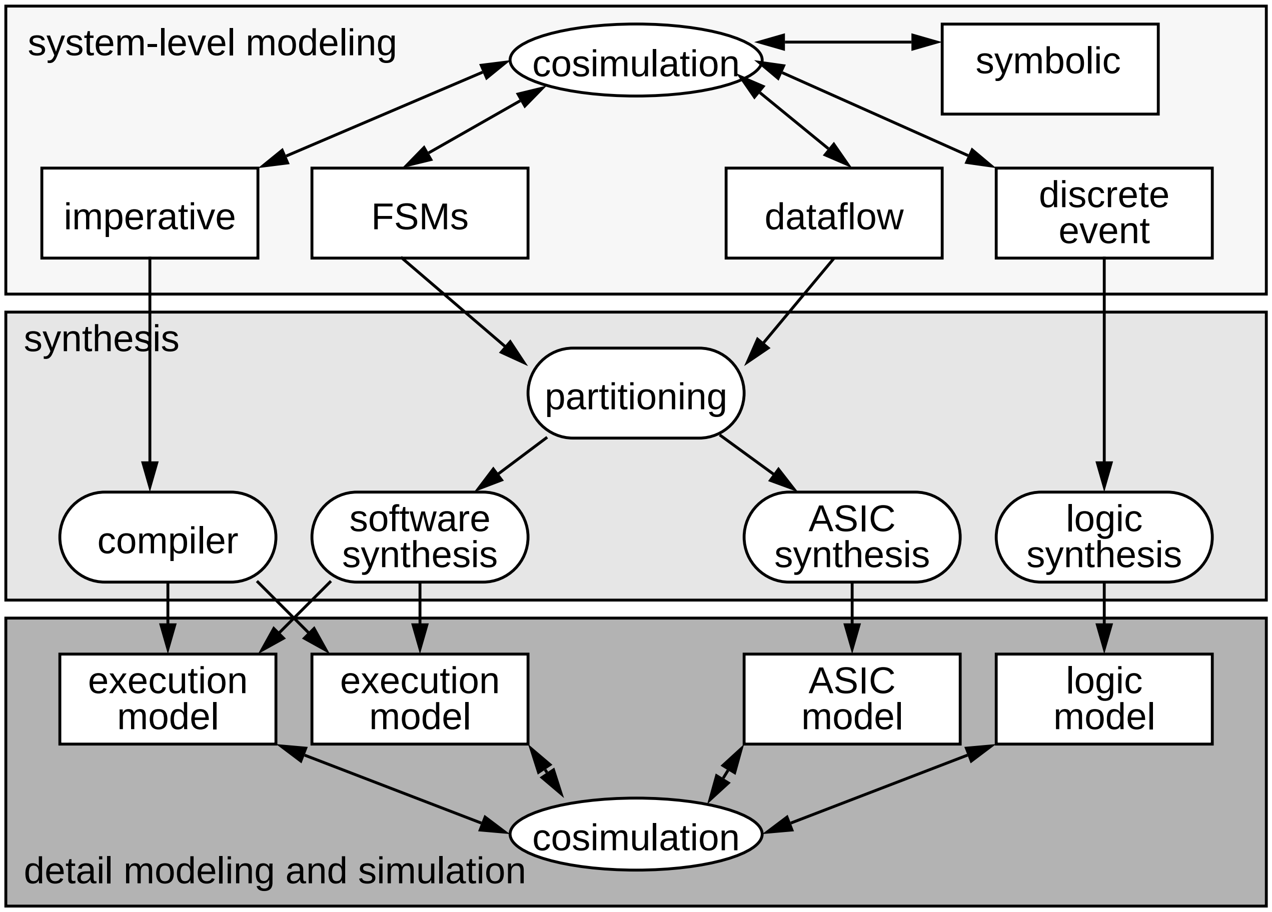


Figure 1. Heterogeneity in system-level design of signal processing and communications systems.

els described by mathematical equations, as discussed in Section 2. Section 3 discusses applying algebraic identities in simplifying and rearranging combinations of components of behavioral models as one method to explore the design space of alternate implementations, e.g. in system synthesis. These two sections discuss automating the kinds of calculations that engineers might do on scratch paper and then discard. The symbolic calculations can be saved for future reference and modification. The next step is to have these calculations performed whenever the underlying assumptions change. Tracking parameter dependencies and tool invocations is part of an overall design management infrastructure. Design management issues also include specifying and managing complex design flows and maintaining consistency of design data and flows. In order to manage the complexity of this design process, an infrastructure that manages these issues, transparent to the user, is presented in Section 4. These design management concepts have been implemented in the Ptolemy [1] environment.

2. ALGEBRAIC COMPUTATION

Many symbolic tools can perform symbolic algebra and symbolic calculus, e.g. Maple [2] and Mathematica [3]. Based on these abilities, the symbolic tools can work with signals and systems, and their parameters, in terms of formulas. In this section, we discuss the use of symbolic tools to compute free parameters (Section 2.1), derive new behavioral models (Section 2.2), and optimize behavioral models (Section 2.3). Ultimately, we would store the symbolic computations as part of the system design process so that designers could follow and modify the underlying computations.

Proc. 1995 SPIE Int. Sym. on Advanced Algorithms, Architectures, and Implementations

Symbolic Computation in System Simulation and Design

Brian L. Evans, Steve X. Gu, Asawaree Kalavade, and Edward A. Lee

Dept. of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720-1770, USA
{ble,sgu,kalavade,eal}@eecs.berkeley.edu

ABSTRACT

This paper examines some of the roles that symbolic computation plays in assisting system-level simulation and design. By symbolic computation, we mean programs like Mathematica that perform symbolic algebra and apply transformation rules based on algebraic identities. At a behavioral level, symbolic computation can compute parameters, generate new models, and optimize parameter settings. At the synthesis level, symbolic computation can work in tandem with synthesis tools to rewrite cascade and parallel combinations on components in sub-systems to meet design constraints. Symbolic computation represents one type of tool that may be invoked in the complex flow of the system design process. The paper discusses the qualities that a formal infrastructure for managing system design should have. The paper also describes an implementation of this infrastructure called DesignMaker, implemented in the Ptolemy environment, which manages the flow of tool invocations in an efficient manner using a graphical file dependency mechanism.

Keywords: behavioral modeling, design methodology management, behavioral optimization, analog filter design

1. INTRODUCTION

This paper discusses aspects of the design and simulation of complex systems. At the top level, the system may be composed of many sub-systems, and those sub-systems in turn may be composed of other sub-systems. Each sub-system may operate under a different computational model and may ultimately be mapped to one or more hardware and software implementation technologies. A system may therefore be heterogeneous in the computational models and implementation technologies it uses.

The primary issues involved in designing heterogenous systems are system modeling, simulation, and synthesis. *Modeling* involves grouping algorithms operating under the same computational model into sub-systems and grouping the sub-systems into a hierarchy to describe the behavior of the overall system. *Simulation* is necessary to validate the behavioral modeling of the system, as well as any synthesized implementations of the system. Because simulation requires the interaction of many different computational models through the exchange data and control information, it is often called cosimulation. *Synthesis* involves mapping the entire system into hardware and software. The computational models used by some sub-systems may lend themselves to either a hardware or software implementation, but other computational models are better implemented by partitioning the sub-system components into hardware and software. Many possible choices exist for a viable hardware and software implementation.

Figure 1 depicts one view of heterogeneity in system-level design for signal processing and communications systems. In these systems, the processing of data can often be described in terms of dataflow graphs, and the control logic in terms of finite state machines (FSMs). Timing circuitry can be modeled using discrete-event semantics. Many of the operations that do not fall into one of these models may be captured using an imperative specification, such as the C programming language or a scripting language. For synthesis, imperative languages are naturally matched to software implementations, and discrete-event semantics are naturally matched to hardware implementations. FSM and dataflow sub-systems, however, could be partitioned into hardware and software.

Symbolic computation plays a variety of roles in system design and simulation. By symbolic computation, we mean performing symbolic algebra and applying transformation rules based on algebraic identities. In system modeling, symbolic computation can perform algebraic calculations of parameters and optimize parameter values of behavioral mod-