Metropolis: An Integrated Electronic System Design Environment

Based on a metamodel with formal semantics that developers can use to capture designs, Metropolis provides an environment for complex electronic-system design that supports simulation, formal analysis, and synthesis.

Felice Balarin
Yosinori Watanabe
Cadence Berkeley Labs

Harry Hsieh
University of California, Riverside

Luciano Lavagno
Claudio Passerone
Politecnico di Torino

Alberto Sangiovanni-Vincentelli
University of California, Berkeley

A solid design flow must capture designs at well-defined levels of abstraction and proceed toward an efficient implementation. The critical decisions involve the system's architecture, which will execute the computation and communication tasks associated with the design's overall specification. Understanding the application domain is essential to ensure efficient use of the design flow.

Today, the design chain lacks adequate support. Most system-level designers use a collection of unlinked tools. The implementation then proceeds with informal techniques that involve numerous human-language interactions that create unnecessary and unwanted iterations among groups of designers in different companies or different divisions. These groups share little understanding of their respective knowledge domains. Developers thus cannot be sure that these tools, linked by manual or empirical translation of intermediate formats, will preserve the design's semantics. This uncertainty often results in errors that are difficult to identify and debug.

The move toward programmable platforms shifts the design implementation task toward embedded software design. When embedded software reaches the complexity typical of today's designs, the risk that the software will not function correctly increases dramatically. This risk stems mainly from poor design methodologies and fragile software system architectures, the result of growing functionality over an existing implementation that may be quite old and undocumented. The Metropolis project seeks to develop a unified framework that can cope with these challenges.

DESIGN OVERVIEW

We designed Metropolis to provide an infrastructure based on a model with precise semantics that remain general enough to support existing computation models[1] and accommodate new ones. This *metamodel* can support not only functionality capture and analysis, but also architecture description and the mapping of functionality to architectural elements.

Metropolis uses a logic language to capture nonfunctional and declarative constraints. Because the model has a precise semantics, it can support several synthesis and formal analysis tools in addition to simulation.

The first design activity that Metropolis supports, communication of design intent and results, focuses on the interactions among people working at different abstraction levels and among people working concurrently at the same abstraction level. The metamodel includes constraints that represent in abstract form requirements not yet implemented or assumed to be satisfied by the rest of the system and its environment.

0018-9162/03/$17.00 © 2003 IEEE

Related Research and Tools

Several past and present research activities have influenced our development of Metropolis, including those that follow.

POLIS/VCC and related research

Cadence Design Systems based its system-level design environment, Virtual Component Codesign (VCC), on the idea of separation between functionality and architecture pioneered in POLIS.[1] Metropolis is based on the same ideas but implements them more flexibly. For example, VCC uses a fixed computation model, while Metropolis lets users define the communication primitives and execution rules most suitable for the application at hand. An architecture can be constructed from a predefined set of components in VCC, although it cannot handle a recursive layering of platform models. VCC uses C and C++ to define the behavior of processes, which rules out formal process analysis or optimization techniques beyond those that standard software compilers use.

Tools from Arexsys, Foresight, Artisan, and CardTools resemble VCC's spirit and implementation. They all use a separation between functionality and architectural resources as well as a mapping to derive performance information. Of these tools, only Arexsys's ArchiMate uses a formal language, SDL, to model system functionality; the others use C or C++.

System-level languages and frameworks

The Metropolis metamodel bears several similarities to systems and languages such as Ptolemy,[2] SystemC,[3] and SpecC,[4] since all share the notion of concurrent processes communicating through channels. Their primary focus is system modeling, and therefore they omit features necessary to orthogonalize functionality and architecture, such as mapping between functional and architectural networks or between different refinement levels. Further, these languages do not have the ability to explicitly represent constraints. Finally, using the underlying C/C++ semantics, particularly pointers, while convenient for modeling, hinders automated synthesis.

Commercial hardware and software coverification tools from companies such as Mentor, Vast, Virtio, and Axys can provide fast instruction-set simulation linked to various hardware simulators. They attack the functional and performance modeling problem for software-dominated embedded systems, although they do not address the issues of high-level hardware modeling and refinement.

References

1. F. Balarin et al., *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Publishers, 1997.
2. J. Buck et al., "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int'l J. Computer Simulation*, Apr. 1994, pp. 155-182.
3. T. Grotker et al., *System Design with SystemC*, Kluwer Academic Publishers, 2002.
4. D. Gajski, J. Zhu, and R. Domer, *The SpecC Language*, Kluwer Academic Publishers, 1997.

Metropolis accomplishes the second design activity, analysis, through simulation and formal verification to determine how well an implementation satisfies the requirements. Proper use of abstraction can dramatically accelerate verification. The constant use of detailed representations, on the other hand, can introduce excessive dependencies between developers, reduce the interfacing requirements' understandability, and diminish the efficiency of analysis mechanisms.

Metropolis addresses the third design activity, synthesis, throughout the abstraction levels used in a design. Typical problems we address include setting the parameters of architectural elements such as cache sizes, or designing scheduling algorithms and interface blocks. We also deal with synthesis of the final implementations in hardware and software. In Metropolis, a specification may mix declarative and executable constructs of the metamodel, which are then automatically translated into the semantically equivalent mathematical models that the synthesis algorithms are applied to.

It can be argued that application domains and their constraints on attributes—such as cost, energy, performance, design time, and safety—are so different that there is insufficient economy of scale to justify developing tools to automate these design activities. With the Metropolis project, however, we seek to show that this is untrue for at least a broad class of domains and implementation choices.

The choice of technique or algorithm for analysis and synthesis of a particular design depends on the application domain and the design phase. For example, safety-critical applications may need formal verification techniques, which require significant human skills for use on realistic designs. On the other hand, formal verification tools can execute simple low-level equivalence checks between various abstraction levels in hardware design—such as logic versus transistor levels.

Thus, we do not intend for Metropolis to provide algorithms and tools for all possible design activities. Instead, it offers syntactic and semantic mechanisms to compactly store and communicate all relevant design information, and designers can use it to plug in the required algorithms for a given application domain or design flow.

Metropolis includes a parser that reads metamodel designs and a standard API that lets developers browse, analyze, modify, and augment additional information within those designs. For each tool integrated into Metropolis, a back end uses the API to generate required input by the tool from the design's relevant portion. This unified

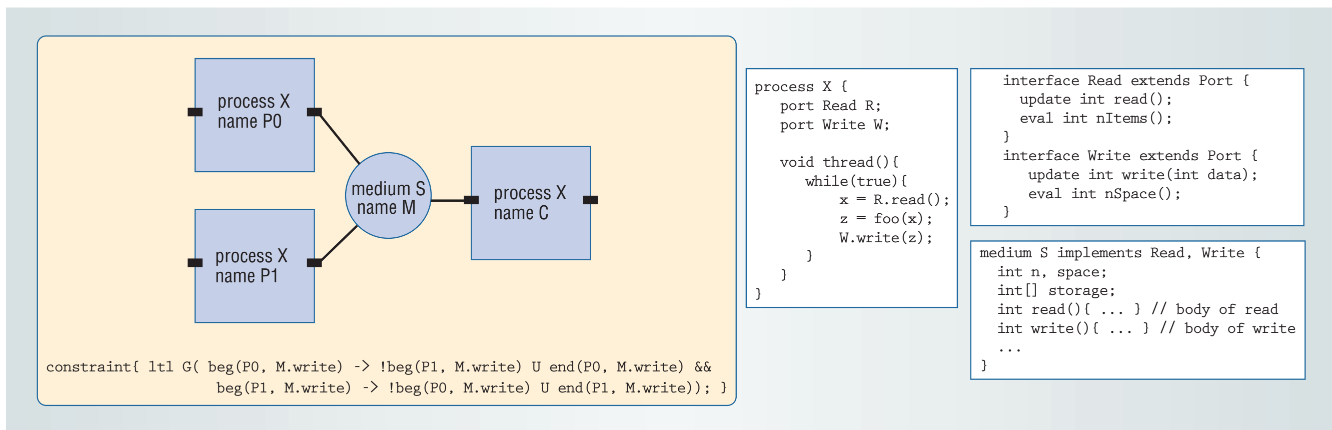

Figure 1. Functional network model. The network consists of two producer processes and one consumer process, all of which communicate through a medium.

mechanism makes it easy to incorporate tools developed elsewhere, as demonstrated by integrating the Spin[2] software verification tool into Metropolis.

Throughout Metropolis's development, we have drawn on related work in this field—some of which we describe in the "Related Research and Tools" sidebar—both as a basis for our own work and as a reference point for further improvement.

METROPOLIS METAMODEL

A language for specifying networks of concurrent objects that each take actions sequentially, the Metropolis metamodel formally defines a network's behavior by the language's execution semantics.[3] The metamodel can be used to represent all the key ingredients in the design flows: function, architecture, mapping, refinement, abstraction, and platforms.

Function modeling

A system's function is the set of objects that concurrently take actions while communicating with one another. We call such an object a *process* in the metamodel and associate it with a sequential program called a *thread*. A process communicates through the ports defined in it. A *port* is specified with an *interface*, which declares a set of methods that the process can use through the port.

An interface may be implemented in different ways, and we refer to objects that implement port interfaces as *media*. Any medium can be connected to a port if it implements the port's interface. This mechanism lets the metamodel separate computation by processes from communication among them. This separation is essential to facilitate the description of the objects to be reused for other designs. Figure 1 shows a network of two producer processes and one consumer process that communicate through a medium.

Once we have established a network of processes, we use the metamodel's semantics to precisely define the network's behavior as a set of *executions*. First, we define a process's execution as a sequence of *events*, which are a program's entries or exits to some piece of code. For example, for Figure 1's process X, the beginning of the call to R.read() is an event, as is its termination.

We then define a network execution as a sequence of event vectors in which each vector has at most one event for each process to define the set of events that happen altogether. The metamodel can model nondeterministic behavior, useful for abstracting a part of the design, thus there may be more than one possible execution of the network.

Constraints, written in logic formulas, further restrict the set of executions defining the set of *legal executions*.[3] For example, the constraint in Figure 1 specifies the mutual exclusion of the two producers when one calls the medium's write method. Constraints describe the coordination of processes or relate the behavior of networks through mapping or refinement.

Architecture modeling

Two aspects distinguish architectures: the functionality they can implement and that implementation's efficiency. In the metamodel, we model functionality as a set of services that an architecture offers to the functional model. Services are just methods, bundled into interfaces.[4]

To represent an implementation's efficiency, we must model the cost of each service. We do so by first decomposing each service into a sequence of events, then annotating each event with a value representing the event's cost.

To decompose the services into sequences of events, we use networks of media and processes, just as in the functional model. These networks often correspond to the physical structures of implementation platforms. For example, Figure 2 shows an architecture consisting of *n* processes, *T1*, …, *Tn*,—and three media—CPU, BUS, MEM. The architecture in Figure 2 also contains *quantity managers*, represented by diamond-shaped symbols.

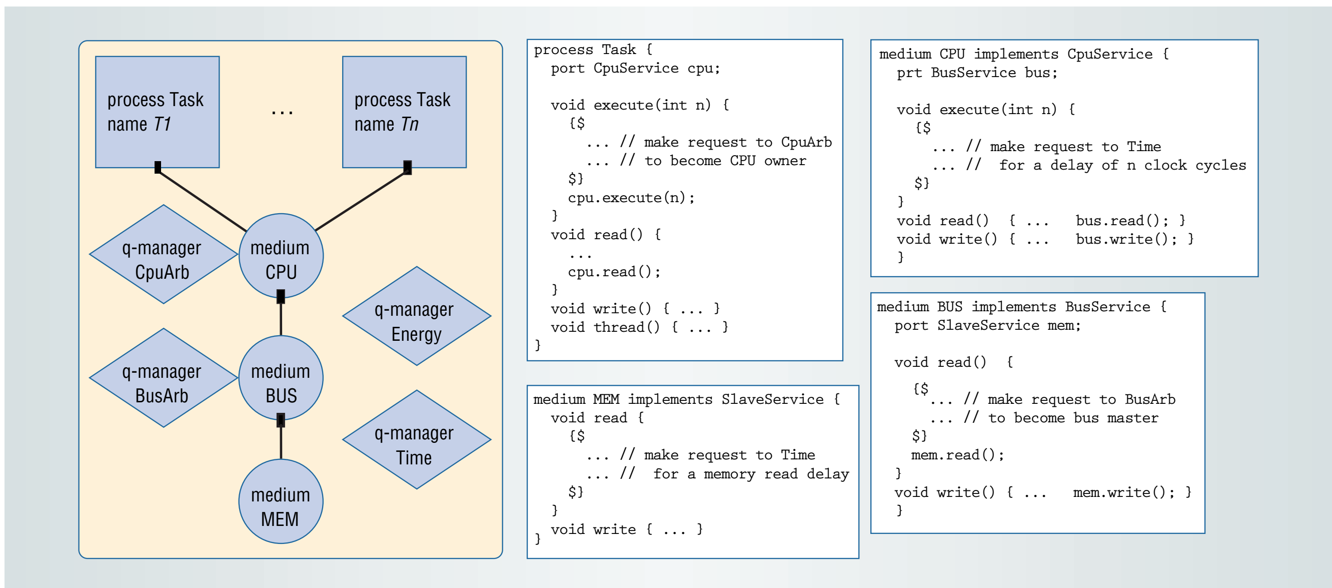

Figure 2. Architectural network model. Several processes model software tasks that execute on the CPU; three media model the CPU, bus, and memory; and four quantity managers measure the cost in system resources of processing individual events.

The processes model software tasks executing on a CPU, while the media model the CPU, bus, and memory.

The services that this architecture offers are the execute, read, and write methods that the Task processes implement. The *thread* function of a Task process repeatedly and nondeterministically executes one of the three methods. In this way, we model that the tasks can execute these methods in any order. The actual order will become fixed only after the designer finishes function-mapping to the architecture so that each task implements a particular process of the functional model.

While a Task process offers its methods to the system's functional part, the process itself uses services that the CPU medium offers, which, in turn, uses the BUS medium's services. In this way, the system decomposes the top-level services that the tasks offer into sequences of events throughout the architecture.

The metamodel includes the notion of using quantity to annotate individual events and using values to measure cost. For example, Figure 2 uses the energy quantity to annotate each event with the energy required to process it. To specify that a given event takes a given amount of energy, we associate with that event a request for that amount. The system makes these requests to the quantity manager object, which collects all requests and, if possible, fulfills them. An event can be processed if its request is filled; otherwise it will wait until the manager grants the request.

Quantities can also be used to model shared resources. For example, in Figure 2 the CpuArb quantity labels every event with the current CPU owner's task identifier. Assuming that a process can progress only if it is the current CPU owner, the CpuArb manager effectively models the CPU scheduling algorithm. The metamodel has no built-in notion of time, but developers can model the time as yet another quantity that puts an annotation—in this case a time stamp—on each event.

For efficiency's sake, Metropolis provides standard libraries for managing common quantities, such as time. In addition, design flow developers can write quantity managers to support quantities relevant to a specific application domain.

Mapping

Evaluating a particular implementation's performance requires mapping a functional model to an architectural model. The metamodel can do this without modifying the functional and architectural networks. It does so by defining a new network to encapsulate the functional and architectural networks and relating the two by synchronizing events between them. This new network, called a *mapping network*, can be considered a top layer that specifies the mapping between the function and architecture.

The synchronization mechanism roughly corresponds to an intersection of the execution sets for the functional and architectural models. Functional-model executions specify a sequence of events for each process, but they usually allow arbitrary interleaving of the concurrent processes' event sequences, as their relative speed is undetermined.

On the other hand, architectural model executions typically specify each service as a timed sequence of events, but they exhibit nondeterminism with respect to the order in which they perform services and on what data. The mapping eliminates all executions from the two sets except those in which the events that should be synchronized always appear simultaneously. Thus, the remain-

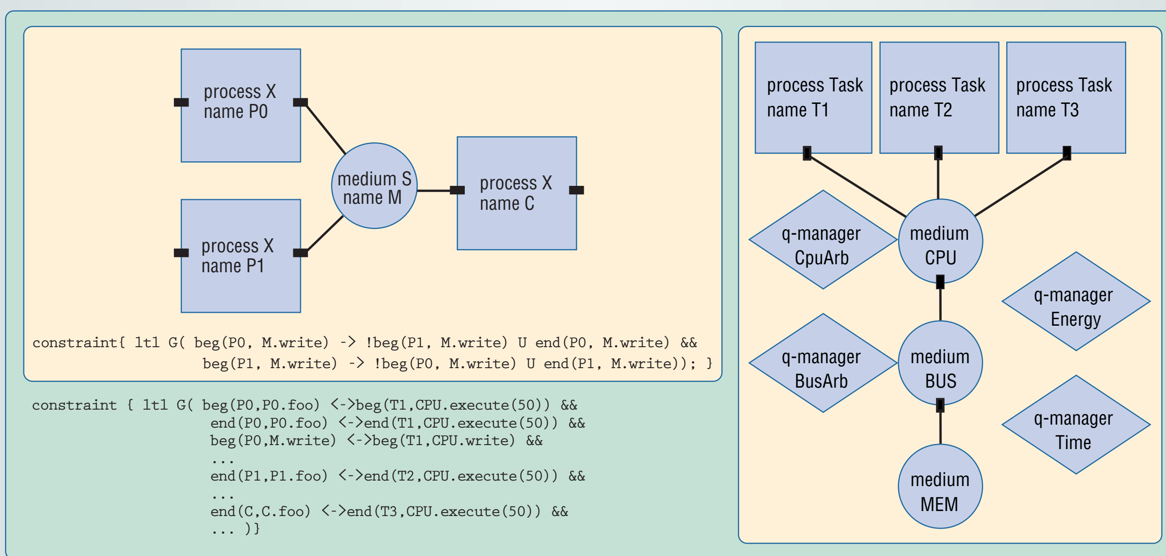

Figure 3. Mapping network. This network encapsulates the functional and architectural networks and relates them to define an implementation of the function. The network itself can be considered an architecture of a service: The functional network specifies the service's algorithm, while the architectural network defines its performance.

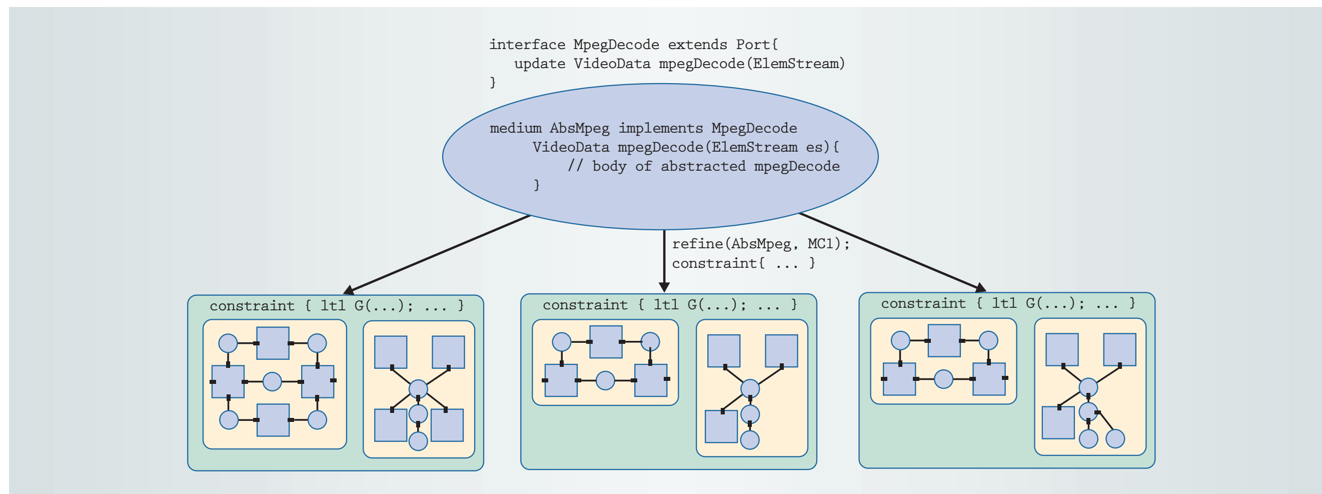

Figure 4. Multiple mapping netlists refine the same service to create a platform. The method declaration interface at the top defines methods that specify a service, while the oval medium in the middle implements the interface at the desired abstraction level. The refinement constraints help the metamodel relate the medium to each network.

ing executions represent timed sequences of events for the concurrent processes.

Figure 3 shows a mapping network that combines the functional network from Figure 1 with the architectural network from Figure 2. The mapping network's constraint clause synchronizes the two networks' events. For example, executions of execute, read, and write by *T1* synchronize executions of foo, read, and write by *P0*. Since *P0* executes its actions in a fixed order, while *T1* chooses its actions nondeterministically, synchronization forces *T1* to follow *P0*'s decisions, while *P0* inherits *T1*'s quantity annotations. Thus, mapping *P0* to *T1* makes *T1* become a performance model of *P0*. Similarly, *T2* and *T3* become performance models of *P1* and *C*, respectively.

Recursive paradigm of platforms

Suppose that we obtain a mapping network such as the one shown in Figure 3. We can consider the network itself an implementation of a certain service. The functional network provides the algorithm for implementing the service, while the architecture counterpart defines its performance.

In Figure 4, the interface at the top defines methods that specify a service. The medium in the middle implements the interface at the desired abstraction level. Underneath the medium, a map-

The Metropolis metamodel's recursive platform-based design paradigm employs formal semantics that precisely define network behavior.

ping network provides a more detailed description of the service's implementation. The metamodel can use the construct refine and constraints to relate the medium and each network. For example, the constraints may say that the begin event of the medium's mpegDecode is synchronized with the begin event of vldGet for the network's VLD process, and that the end of mpegDecode synchronizes with the end of yuvPut in the OUTPUT process, while the value of the variable YUV at the event's execution agrees with the mpegDecode output value.

Many mapping networks can exist for the same service, each with different algorithms or architectures. Such a set of networks, together with constraints on event relations for a given interface implementation, constitute a *platform*. The platform elements provide the same service with different costs, with given design requirements favoring one over another.

This concept of platforms appears recursively in the design process. Generally, an implementation of what one designer conceives as the entire system represents a refinement of a more abstracted service model, which can in turn be employed as a single component of the larger system.

For example, a mapping network might give a particular implementation of an MPEG decoder, but its service may be modeled by a single medium in which the events generated in that medium are annotated with performance quantities to characterize the decoder. A company that designs broadband set-top appliances could use such a medium as part of its product's architectural model. The company would use this medium to evaluate its set-top box design, while the MPEG decoder's provider might use the same medium for its design requirements.

Similarly, the MPEG decoder can use a bus model in its architecture, which another company provides as a medium. For the company that provides it, the bus design itself forms the entire system, as given by another mapping network that refines the medium. Yet this mapping network may be only one candidate in the communication service's platform, and the MPEG decoder's designer may explore various options based on the design criteria.

The Metropolis metamodel uniformly describes this recursive paradigm of platform-based design. Its key contributions are that it employs formal semantics that precisely define network behavior, and it provides a mechanism for relating different networks' events in terms of quantities annotated to the events.

TOOLS

The Metropolis framework includes tools for verification, simulation,[5] and synthesis. Here we highlight three of them: tools for formal and simulation-based property verification, and schedule synthesis for concurrent computation.

Formal property verification

Both academia and industry have long studied formal property verification, but the state-explosion problem restricts its usefulness to protocols or other high abstraction levels. At the implementation or other low abstraction levels, hardware and software engineers have used simulation monitors as basic tools to check simulation traces while debugging designs.

Verification languages, such as Promela, which the Spin model checker uses,[2] allow only simple concurrency modeling and are not amenable to system design specification, which requires complex synchronization and architecture constraints. In contrast, Metropolis, with its formal semantics, automatically generates verification models for all the design's levels.[6]

Our translator automatically constructs the Spin verification model from the metamodel specification, taking care of all system-level constructs. For example, it can automatically generate a verification model for the example in Figure 1 and verify the medium's nonoverwriting properties. Further, as the translator refines the design through structural transformation and architectural mapping, it can prove more properties, including throughput and latency. This kind of property verification typically requires several minutes of computation on a 1.8-GHz Xeon machine with 1 Gbyte of memory. When the state space complexity becomes too high, Metropolis uses approximate verification and provides the user with a confidence factor on the passing result.

Simulation monitors

Simulation monitors offer an attractive alternative to formal property verification. In Metropolis, designers can use *logic of constraints* (LOC) formulas[3] to specify quantitative properties. The system can automatically translate the specification to simulation monitors in C++,[7] thus relieving designers from the tedious and error-prone task of writing monitors in the simulator's language. The monitors analyze the traces and report any LOC formula violations. Like any other simulation-based approach, this one can only disprove a LOC formula if it finds a violation—it can never prove conclusively the formula's correctness because that

would require analyzing traces exhaustively. The automatic trace analyzer can be used in concert with model checkers. It can perform property verification on a single trace even when other approaches would fail because of their excessive memory and space requirements.

In our experience with applying the automatic LOC-monitor technique to large designs with complex traces, we have found that in most cases the analysis completes in minutes and consumes only hundreds of bytes of data memory to store the LOC formulas. The analysis time tends to grow linearly with the trace size, while the memory requirement remains constant regardless of the trace size.[7]

Quasi-static scheduling

We have developed an automatic synthesis technique called *quasistatic scheduling*[8] to schedule a concurrent specification on computational resources that provide limited concurrency. QSS considers a system to be specified as a set of concurrent processes communicating through FIFO queues and generates a set of tasks that are fully and statically scheduled, except for data-dependent controls that can be resolved only at runtime. A task usually results from merging parts of several processes together and shows less concurrency than the initial specification. Moreover, QSS allows interprocess optimizations that are difficult to achieve if processes remain separated, such as replacing interprocess communication with assignments.

This technique proved particularly effective and let us generate production-quality code with improved performance. Applying QSS to a significant portion of an MPEG-2 decoder resulted in a 45 percent increase in overall performance.

The assumptions that QSS requires for the input specification form a subset of what the metamodel can represent. Therefore, when integrating QSS into the Metropolis framework, we addressed two main problems: how to verify if a design satisfies the required set of rules and how to convey all relevant design information to the QSS tool.

We addressed the first problem by providing a library of interfaces and communication media that implement a FIFO communication model. Those parts of the design optimized with QSS need to use these communication primitives.

To convey relevant design information to QSS, we use a back-end tool that translates a design to be scheduled with QSS into a Petri net specification, which is QSS's underlying model. QSS then uses the Petri net to produce a new set of processes. These new processes show no interprocess communication because QSS removes it. The processes communicate with the environment using the same primitives implemented in the library. The new code can thus be directly plugged into the metamodel specification as a refinement of the network selected for scheduling.

Metropolis avoids imposing a specific language or flow model on designers, instead making their preferred approach more robust and rigorous.

The Metropolis metamodel's formal semantics allow embedding computation models in a rigorous framework that favors design reuse and design chain support. The system's features can facilitate the dialog among designers with different knowledge domains. With Metropolis, we seek to avoid imposing a specific language or flow model on designers, instead making their preferred approach more robust and rigorous. Metropolis also offers a set of analysis and synthesis tools that show how designers can use the framework to integrate flows.

We plan to add tools to Metropolis as we address different application domains. Currently, we are exploring the automotive, wireless communication, and video application domains in collaboration with our industrial partners. As we identify the design's critical parts and determine what must be supported to facilitate design hand-offs, we plan to tune the metamodel and increase the infrastructure's power to support successive refinement. We also plan to make Metropolis and its components open domain to expose these ideas to the academic and industrial communities. ■

Acknowledgments

We thank the MARCO GSRC program, which partially supported the development of Metropolis; Cadence Berkeley Labs's researchers for helping to develop Metropolis; and our partners PARADES, the Berkeley Wireless Research Center, Intel, Cypress Semiconductor, Nokia, Philips, STMicroelectronics, BMW, and Magneti Marelli for providing support and collaboration during the various phases of Metropolis's development. We also thank the many people involved in Metropolis and related projects, including Jerry Burch, Luca Carloni, Rong Chen, Xi Chen, Robert Clariso Viladrosa, Jordi Cortadella, Erwin deKock, Doug Densmore, Alberto Ferrari, Daniele Gasperini, Gregor Gössler, Timothy Kam, Arjan Kenter, Mike Kishinievski, Alex Kondratyev, Wido Kruijtzer, Dunny Lam, Alberto La Rosa, Radu Marculescu, Grant Martin, Trevor Meyerowitz, John

Moondanos, Andy Ong, Roberto Passerone, Claudio Pinello, Alessandro Pinto, Sanjay Reikhi, Ellen Sentovich, Marco Sgroi, Greg Spirakis, Laura Vanzago, Ted Vucurevich, Howard Wong-Toi, and Guang Yang.

References

1. E. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Dec. 1998, pp. 1217-1229.
2. G.J. Holzmann, "The Model Checker Spin," *IEEE Trans. Software Eng.*, May 1997, pp. 258-279.
3. F. Balarin et al., "Modeling and Designing Heterogeneous Systems," *Concurrency and Hardware Design*, J. Cortadella, A. Yakovlev, and G. Rozenberg, eds., Springer, 2002, pp. 228-273.
4. S. Solden, "Architectural Services Modeling for Performance in HW-SW Co-Design," *Proc. Workshop on Synthesis and System Integration of Mixed Technologies*, IEEE Press, 2001, pp. 72-77.
5. F. Balarin et al., "Concurrent Execution Semantics and Sequential Simulation Algorithms for the Metropolis Metamodel," *Proc. 10th Int'l Symp. Hardware/Software Codesign*, IEEE CS Press, 2002, pp. 13-18.
6. X. Chen et al., "Formal Verification of Embedded System Designs at Multiple Levels of Abstraction," *Int'l Workshop High-Level Design, Validation and Test*, IEEE CS Press, 2002, pp. 125-130.
7. X. Chen et al., "Automatic Generation of Simulation Monitors from Quantitative Constraint Formula," to be published in *Proc. Design Automation and Test in Europe 2003*, ACM Press, 2003.
8. J. Cortadella et al., "Quasi-Static Scheduling of Independent Tasks for Reactive Systems," *Proc. 23rd Int'l Conf. Application and Theory of Petri Nets*, Springer, 2002, pp. 80-99.

Felice Balarin *is a research scientist at Cadence Berkeley Labs. His research interests focus on the development and application of formal methods for the design, verification, control, and timing analysis of embedded systems implemented both by hardware and software. Balarin received a PhD in electrical engineering and computer science from the University of California, Berkeley. Contact him at felice@cadence.com.*

Yosinori Watanabe *is a research scientist at Cadence Berkeley Labs. His research interests focus on methodologies and synthesis and verification of system designs. Watanabe received a PhD in electrical engineering and computer science from the University of California, Berkeley. Contact him at watanabe@cadence.com.*

Harry Hsieh *is an assistant professor at the University of California, Riverside. His research interests include all aspects of electronic systems design, with emphasis on computer-aided design and verification, embedded systems architecture, and embedded software. Hsieh received a PhD in electrical engineering and computer science from the University of California, Berkeley. Contact him at harry@cs.ucr.edu.*

Luciano Lavagno *is an associate professor in the Department of Electronics at the Politecnico di Torino. His research interests include synthesis and testing of asynchronous circuits and concurrent design of mixed hardware and software embedded systems. Lavagno received a PhD in electrical engineering and computer science from the University of California, Berkeley. Contact him at lavagno@polito.it.*

Claudio Passerone *is a research associate in the Department of Electronics at the Politecnico di Torino. His research interests include system-level design, electronic system simulation, and reconfigurable hardware. Passerone received a PhD in electrical and communication engineering from the Politecnico di Torino. Contact him at claudio.passerone@polito.it.*

Alberto Sangiovanni-Vincentelli *is a professor at the University of California, Berkeley, and is chief technology advisor to Cadence. His research interests include computer-aided analysis and design and hybrid and embedded-system design. Sangiovanni-Vincentelli received a DrEng degree in engineering from Politecnico di Milano, Italy. He is a member of the National Academy of Engineering. Contact him at alberto@eecs.berkeley.edu*