

Metropolis: Design Environment for Heterogeneous Systems

Felice Balarin Jordi Cortadella Luciano Lavagno Claudio Passerone Roberto Passerone
Alberto Sangiovanni-Vincentelli Marco Sgroi Yosinori Watanabe

Abstract

Metropolis is a system design environment in which designs are carried out by a well-defined methodology that orthogonalizes and localizes design concerns, captures design decisions and requirements explicitly and unambiguously, and exploits the underlying theoretical framework to effectively apply a tool-set throughout the design phases. The project, supported by the Gigascale Silicon Research Center, is in a process of prototyping the infrastructure and the tool-set of the environment, and applying the methodology to design drivers in various application domains. This paper presents a brief overview of the project, and reports the status and future plans.

1 Introduction

Future devices will be network-connected, channeling streams of data into the infrastructure, with moderate processing on the fly. Others will have narrow, application-specific UIs. Applications will not be centered within a single device, but stretched over several, forming a path through the infrastructure. In such applications, the ability of the system designer to specify, manage, and verify the functionality and performance of *concurrent behaviors* is essential. We believe that it is most likely that the preferred approaches to the implementation of complex embedded systems will include the following aspects:

1. Design time and cost are likely to dominate the decision-making process for system designers. Therefore, design reuse in all its shapes and forms, as well as just-in-time, low-cost design debug techniques will be of paramount importance.
2. Designs must be captured at the highest level of abstraction to be able to exploit all the degrees of freedom that are available. Such a level of abstraction should not make any distinction between hardware and software, since such a distinction is the consequence of a design decision.
3. The implementation of efficient, reliable, and robust approaches to the design, implementation, and programming of concurrent systems is essential. In essence, whether the silicon is implemented as a single, large chip or as a collection of smaller chips interacting across a distance, the problems associated with concurrent processing and concurrent communication must be dealt with in a uniform and scalable manner. In any large-scale embedded systems program, concurrency must be considered as a first class citizen at all levels of abstraction and in both hardware and software.
4. Concurrency implies communication among components of the design. Communication is too often intertwined with the behavior of the components of the design so that it is very difficult to separate out the two domains. Separating communication and behavior is essential to dominate system design complexity. If in a design component behaviors and communications are intertwined, it is

very difficult to re-use components since their behavior is tightly dependent on the communication with other components of the original design.

Our aim with *Metropolis* is high: we plan to develop a system design methodology with the appropriate supporting tools that: (i) is acutely aware of the different metrics that drive and govern the design process of the envisioned embedded systems, including cost, weight, size, power dissipation, and required performance, (ii) can be applied at all levels of abstraction of the design, from conception to software and silicon implementation and packaging, with guaranteed correctness and efficient trade-off evaluation, (iii) favors re-use by identifying the requirements for real plug-and-play operation.

The *Metropolis* project, supported by the Gigascale Silicon Research Center (GSRC), involves a large number of people in different research institutions. It is based on the following principles:

- **Orthogonalization of concerns:** In *Metropolis*, behavior is clearly separated from implementation. Communication and computation are orthogonalized. Communication is recognized today as the main difficulty in assembling systems from basic components. Errors in software systems can often be traced to communication problems. *Metropolis* was created to deal with communication problems as the essence of the new design methodology. Communication-based design will allow the composition of either software or hardware blocks at any layer of abstraction in a controlled way. If the blocks are correct, the methodology ensures that they communicate correctly.
- **Solid theoretical foundations that provide the necessary infrastructure for a new generation of tools:** We believe that without a rigorous approach, the goal of correct, efficient, reliable and robust designs cannot be achieved. The tools used in *Metropolis* will be interoperable and will work at different levels of abstraction, they will verify, simulate, and map designs from one level of abstraction to the next, help choose implementations that meet constraints and optimize the criteria listed above. For this purpose, a single internal representation called *meta-model* of computation is used to represent different computation and communication semantics of a design in a unified way at all levels of abstractions. It has a precise semantics and is backed by a theoretical framework to formally describe and relate different Models of Computation (MoCs). *Metropolis* will deal with both embedded software and hardware designs since it will intercept the design specification at a higher level of abstraction. The design specifications will have precise semantics. The semantics is essential to be able to: (i) reason about designs, (ii) identify and correct functional errors, (iii) initiate synthesis processes.
- **Reduction of design time and cost by using platform:** Platforms have been a common approach to re-use.

We have formalized and elaborated the concept of platform to yield an approach that combines hardware and software platforms to build a system platform. An essential part of a platform is its communication architecture. Communication-based design principles have been used to define standard communication schemes, but we advocate a more abstract use of communication-based design to allow more flexible and better-specified communication architectures.

The Metropolis methodology, by leveraging the three basic principles, builds an environment where the design of complex systems will be a matter of days versus the many months needed today. Complex, heterogeneous designs will be mapped into flexible system platforms by highly optimized "design agents" and verified by "verification agents" in a formal logic framework. Formal definition of the semantics of communication are provided so that implementation choices will be correct by construction¹. The ideas will be verified on design drivers in the world of wireless communication where communication is by definition the main concern. In fact, Metropolis will be extensively used for the design of the pico-radio wireless network, one of the projects carried out at the Berkeley Wireless Research Center.

2 Design Methodology

To achieve the goals stated in Section 1, we have developed a framework of design methodologies based on the principle of separation of concerns. It is not biased towards a specific Model of Computation or a particular communication semantics. It proceeds from the highest level of abstraction to the final implementation through a series of refinement steps. Rigorous rules for communication refinement allow the designer to *consciously* take decisions and understand their impact on the overall system, and to optimize the final implementation.

At the highest level of abstraction, a system is considered as a *single process* that describes the desired function. A process is a relation from an input domain to an output codomain (e.g. from a radio signal to a base band voice signal, or from an MPEG stream to a sequence of frames). Therefore, *the initial specification of a process is denotational, rather than operational*. From this level of abstraction, the methodology consists of a series of steps that can be iterated until a final implementation satisfying the system requirements is obtained.

Functional decomposition. The first step towards an implementation is to "decompose" the initial process into a network of communicating processes. Each process is defined in terms of a relation between its input and output domains. The interconnection is the "architecture" of the decomposition. Functional decomposition can be repeated as many times as needed, where each process in the network at one level of abstraction is decomposed into a network of processes at the lower level of abstraction².

Traditional design practices call for details of the communication interfaces at the specification level thus compromising re-usability and restricting communication choices unnecessarily. In our methodology inter-process communication is totally free at the decomposition level. For example, we may decompose an MPEG encoder into three parts

that handle spatial redundancy (DCT), temporal redundancy (motion compensation), and sequential redundancy (Huffman encoding) without prescribing exactly on how these will interact (block by block, frame by frame, ...).

Behavior adaptation. Consider two processes that are connected, i.e. the output of a process, called the sender, is connected to the input of the other process, called the receiver. If the sender's output domain, defined as the set of its output signals, is different from the receiver's input domain, defined as the set of input signals for which the behavior is defined, a transformation is necessary that maps signals from one domain to the other. For example, consider a video-camera that produces video streams encoded in YUV format and connects to a monitor able to receive and reproduce videos only in RGB format. A mapping is needed to syntactically translate the YUV video stream into the RGB format that can be accepted by the receiver. We call these maps between input and output domains *behavior adapters*. Note that behavior adapters are needed independently of what kind of communication (if any) is required between the processes, i.e. they do not depend on how the data is transferred. Moreover, if components are designed from scratch to communicate among themselves, they usually already *speak the same language*, so that behavior adapters are needed only in few cases. On the other hand, if the design is based on a set of available IP's, then there is no a priori guarantee that the input-output domains of the IP's match and behavior adapters are necessary. The use of behavior adapters as separate objects favors the re-use of the IP's since the IP's themselves need not change.

Media and MoC wrapper insertion. After the input-output domains are made consistent, communication mechanisms among the components have to be chosen and specified. Synchronization is an essential part of communication mechanisms and, as such, has to be considered in this design step. The communication takes place on the symbolic link that joins the components. A *communication medium* is used to model *how* data is transferred (e.g., via lossless FIFO, lossy queue, ...). Each medium exports a set of interfaces (e.g., reader and writer) to its user processes. Processes are required to access the communication medium through those interfaces and need to be synchronized (adapted to the semantics) through a wrapper, which we call *MoC wrapper*. The wrapper does not change the functionality of the process, but only the firing rule according to which the underlying functionality is executed (*when* data is computed and transferred).

At a very high level of abstraction, communication media need not represent a physical channel. In fact a lossless FIFO has no notion of physical implementation, it is a mathematical representation of how data are transferred. A communication medium can be as simple as the identity function, or as complex as a protocol for wireless interconnection.

Communication refinement. Communication are in general ideal connections ensuring that output signals are received without errors and delay by the receiver process. Implementing a communication medium requires to select a *physical channel* (e.g. coaxial cable, wire, ether..) that conveys the communication signals while ensuring that end-to-end quality of service (QoS) requirements are met. When a channel is selected, it may be necessary to introduce adapters (we call them *Channel Adapters*) be-

¹Of course, an appropriate notion of correctness, including embedding of partial orders in time, as well as satisfaction of Quality of Service guarantees, rather than pure equality of I/O traces, must be used.

²This process shows how the design problem is fractal in nature, that is, it repeats itself identically at each level of abstraction.

tween the behaviors of the sender and the receiver, and the channel. Consider the problem of implementing a reliable connection (no losses). If we selected an unreliable channel (e.g. a wireless channel with noise and interference), we would need to introduce adapters that will make the channel satisfy the specifications for the communication by using devices such as error correction, encoding and retransmission. We then segmented the design of the communication among components of the design into three abstract steps: Behavior Adaptation, Medium Adaptation and Channel Adaptation. Each adapter effectively encapsulates the processes defined at higher levels of abstraction so that re-usability is maximized. A change in physical channel only propagates to the Channel Adapter while all other processes remain the same. These three steps can be in turn broken into smaller intermediate steps. Each successive step is a refinement in the sense that all specified properties of the communication mechanism are maintained while new details are added and choices that exclude potential solutions are made. In the end, when no more choices are available, a precise implementation results. The successive refinement process can be supported in a number of ways: from tools that map from one level of abstraction to the next automatically, to libraries of communication mechanisms and channel implementations.

Optimization. The previous steps have been conceived mostly to maximize reusability and therefore require to clearly identify all the levels of abstraction and explicitly declare all the functions that are used. Often, efficient implementations are "flat" in the sense that an implementation that merges all the adaptors and behaviors is more likely to produce a lower-cost higher-performing solution versus an implementation that is forced to maintain the boundaries among processes. If we were to inhibit designers using the Metropolis the possibility of achieving higher performance and lower cost by blurring the boundaries of communication and computation, then we would not have achieved our goal of a design system that can support efficiency and correctness at the same time. In fact, in our methodology, the optimization step follows the other steps and restructures the network to increase optimization opportunities across processes and media. Processes and media can be merged into single entities and some of them may even disappear. In general, the behavior of processes may be changed by the optimization step: we are trading-off performance with re-use. However, we advocate to perform the optimization step only at the end of the successive refinement process outlined above, so that if some other decisions will be necessary for some of the higher levels of abstraction, the designer can easily back-track and start the design process again at the most convenient point.

2.1 Design example

This methodology, albeit in its early stages, was applied to the design of a system at the Berkeley Wireless Research Center: The Intercom [4]. We use that design to illustrate the main points of the methodology. The Intercom is a single-cell wireless network supporting voice communication among a number of mobile terminals in a limited area. The goal of the project was to design the network protocol that defines how network elements, such as mobile terminals and base stations, communicate. The system requirements were given in terms of the services (e.g. voice communication) that the network must support. The interaction of the network with the environment consists of requests and responses of services, e.g. connection

or conference setup/teardown, and the voice signals from the users. The first design step consisted of identifying the network components, i.e. functionally decomposing the overall network behavior into a number of components that together provide the required services. In particular, the network includes a number of mobile terminals that allow users to access the network and a base station that coordinates the network operation, e.g. manages connections among users and stores network status in an internal database. The behaviors of terminals and base station were defined so that they did not require any behavior adaptation; therefore, the next step was communication refinement. The wireless channel was selected to satisfy the user mobility requirement. However, the lossy nature of the wireless channel and the fact that it must be shared by multiple active connections, required the designers to introduce a number of channel adapters. First, an error detection and a retransmission module were used to create a reliable channel from a lossy wireless medium. Any message in the network is retransmitted if it is not positively acknowledged. Then, a Time Division Multiple Access scheme was used to define channels separated in time and allow to have multiple connections that do not interfere with each other.

From this denotational specification of the protocol stack, including only the functions needed at each level, we moved to an operational specification based on ACFSMs [1] (extended FSMs communicating via FIFO queues, as in SDL). This required first to identify objects grouping together related functions and represent them as ACFSMs. Then, we designed and implemented the communication among each pair of connected ACFSMs. The communication model of ACFSMs, asynchronous over unbounded FIFOs, needs to be refined. This requires steps similar, although on a different scale, to those followed to derive the network protocol. For example, unbounded FIFOs were refined into implementable bounded FIFOs and sized after solving rate equations (this was done by comparing the production rate of incoming voice samples and the TDMA transmission rate). Moreover, multiple communication links were mapped on a shared bus using a TDMA access scheme to guarantee shared access.

3 The mathematical framework

One of the goals of Metropolis is the ability to represent different communication semantics independently of the representation of the processes involved in the communication. Therefore, the research activity that is being carried on in the area of formal methods aims at defining a general framework on which it is not only possible to define precisely the semantics of a meta-model (the Metropolis internal representation described in Section 4), but also to study and understand the relationships among the different communication paradigms.

In this section we will discuss some formal semantics requirements for the Metropolis meta-model by relying on pure mathematical objects, that abstract from the requirements of representing the constituents of a system in a concrete syntax that is understood by analysis and simulation tools. Of course, the meta-model must be proven to conform with the formal semantics either exactly, or conservatively in cases where the construct available for the description would not be practical.

We base the definition of the formal semantics on the notion of a *trace*. A trace is a precisely defined mathematical object that represents a possible behavior of a system or a component of a system. A trace might express, for example, a sequence of actions of a process, a sequence of tokens exchanged with a communication medium, or a combination of the two. Traces are also characterized by a function that defines the operation of composition. We use this device for example to define the

notion of atomic execution, that is essential (as we will discuss in Section 4.2) to abstract away the details of how communication is implemented. We say that two actions are *relatively atomic* if and only if *the result of executing them concurrently is the same as the result of executing them in either sequence* (but not necessarily both sequences, as they may not commute in general). This notion can then be extended to the definition of atomicity for a set of actions. This definition captures the notion of atomicity abstractly, without referring to the particular construct that we use to represent this condition in the concrete meta-model syntax discussed in Section 4.2.

By using the notion of a trace we can also study the communication semantics of a process. Abstractly we characterize a communication paradigm in terms of the properties that the paradigm requires and guarantees with respect to a set of traces. For example, we might represent the communication of a static dataflow model as the *requirement* that for each action at least a given number of tokens be present at each input, and the *guarantee* that a given number of output tokens be produced by its firing and be partially ordered (as specified by the traces) with the input tokens that enabled the firing. The collection of these properties for a particular communication semantics implicitly defines a class of satisfying models; hence, we refer to the requirements and guarantees collectively as the *model of communication* of the entity.

By uniformly representing different models of communication we can more easily study their interaction. For example, we can define a synchronous finite state machine type of interaction by (1) imposing a total order on the elements of the traces, (2) requiring one token per input, and (3) guaranteeing that one token per output be produced *simultaneously*. Since this guarantee is stronger than that of static dataflow, it conforms to the definition of unirate static dataflow, and hence processes originally defined for the synchronous FSM model of communication can be proven to behave correctly in a different one, e.g., unirate static dataflow. This kind of reasoning is essential in a framework in which the communication must be considered separately from the computation.

4 Metropolis Meta-Model

The infrastructure of Metropolis is a *meta-model* of computation, which models various computation and communication semantics in a unified way. The Metropolis meta-model is used internally to represent system behavior throughout the design phases, to generate executables for simulation, and as input to formal methods incorporated into Metropolis. The model is internal as we plan to translate specifications given in existing languages automatically to the meta-model, while in principle one can specify system behavior directly using the syntax of the model.

4.1 Objects and Networks

The Metropolis meta-model defines system behavior as interaction of three types of objects: *media*, *processes*, and *schedulers*.

A *medium* is used to specify a communication semantics. It is defined with a set of *ports*, variables, and functions. Values of the variables constitute *states* of the medium, and thus the variables are called *state variables*. They are referenced only by the functions defined in the medium. If a port is defined, it is connected to another medium. In this case, a function of the first medium may evaluate or change the states of the second medium by calling its functions.

A *process* is defined with a set of ports, state variables, and a sequential program called a *thread*. Values of the state vari-

ables and the program counter of the thread define the states of the process. In order to establish communication with other processes, the process calls functions of media connected to the ports to change the states of the media. Hence, communication is always modeled through media. In this view, a process is an active object, in the sense that it specifies what it computes and what it communicates, while a medium is passive, as it only provides a means for communication.

The result of this modeling is a concurrently interacting set of sequential processes. In general, however, there is a need to coordinate the concurrent executions of processes. This may be because certain properties need to be satisfied in the resulting behavior, or because resources are limited in the target architecture. To model such coordination, the Metropolis meta-model provides two means: schedulers, and the `await` primitive described in Section 4.2. A *scheduler* is defined with a set of ports, state variables, and a scheduling policy. It evaluates the states of media, processes, and other schedulers connected to the ports, while its policy controls the execution of processes. States of a scheduler are given by values of the state variables, which often reflect scheduling results (e.g., the last executed process in a round-robin scheduler). Currently, the scheduling policy is defined by a sequential program, while we also plan to use logic expressions so that one can define properties of valid executions without committing to a specific scheduling algorithm.

The functions of a medium are grouped into a set of *interfaces*. As with object-oriented languages, an interface declares function prototypes, while a medium defines their implementations. An object declares a port with a particular interface, and only the functions of the interface can be called in the object for communication through the port. Any medium can be connected to the port as long as it implements the corresponding interface. This is convenient for refining communication, since a medium that models semantics at a high level of abstraction can be replaced with one for the more detailed semantics, without changing the code of the processes.

To define a network of these objects and its execution, the meta-model provides five primitive operations. These allow one to (1) instantiate an object (process or medium), (2) start a process, (3) stop a process, (4) connect a port of an object with a medium, and (5) disconnect a port of an object from a medium³ Any object can call these primitives, so that the network structure can be dynamically changed.

4.2 Conditional Atomic Execution

As already pointed out, interacting concurrent processes often need coordination during execution. We provide a primitive, called `await`, which allows a designer to impose atomicity over a given piece of code. As opposed to the use of a scheduler, it allows to build the required atomicity constraint into the specification of a process or communication medium, independent of the rest of the system. Moreover, it also implements a mechanism to conditionally halt the execution of a thread.

The `await` primitive is defined as follows:

```
await [M] (C) { S }
```

where M is a set of ports (connected to media), C is any boolean expression (which may include function calls, and in particular evaluate the states of media through ports) and S is a block of sequential code, also called *atomic* or *critical section*. The semantics is that the execution of the construct blocks until C becomes true. Once C holds, S is allowed to execute. While S

³Explicit object destruction is currently not provided, and may be added in the future.

is executing, the states of the media listed in M may not be observable, i.e. no interface function can be executed for the media of M during the execution of S , except for those called directly by the thread executing S . Whether S is immediately executed or not when C is true depends on the scheduling constraints specified. The semantics guarantees that when the atomic section is started, C holds, and remains so until the thread itself changes the states of M .

In practice, `await` allows to deal with two important issues:

1. *Firing Rules*: having a blocking semantics, it allows to precisely define when and on which condition a given piece of code must be executed. Any mix of AND and OR semantics can be specified. This aspect is particularly useful when defining the Model of Computation. For example, it's easy to describe (and mix in the same system) both a Discrete Event and a Data Flow semantics.
2. *Atomicity*: by making the state of the listed communication media unobservable, S is practically executed atomically with respect to other processes trying to access the same communication media. This ability is very important when designing resources that are shared among different processes or to guarantee correctness in a concurrent environment. For instance, it can be used in the interface functions of a communication medium modeling a bus, where only one writer has access at any time. To choose which writer, a scheduler should be used.

A very simple example of the meta-model is shown in Figure 1(a). It is a system formed by one process and two media. The process reads integers from the input medium and writes them to the output medium after multiplying them by 2.

5 Metropolis Intermediate Code

When synthesis and verification algorithms are applied, it is often necessary to first capture the system behavior by mathematical models and then apply the algorithms to the models. For this purpose, the meta-model is translated further into a much simpler code representation that we call Intermediate Code (IC), which can be generated from the meta-model automatically by compilation. The IC is then used to derive mathematical models suitable to particular synthesis or verification algorithms to apply.

The simplicity of the IC allows static analysis techniques to be applied naturally. Mathematical models, such as Petri nets, can be automatically derived from the IC, e.g., to generate a static schedule using Quasi Static Scheduling [3], and to detect unreachable control paths by using abstract interpretation [2]. The key concept of the IC is *black boxes*. A black box is used to encapsulate parts of the system that are not represented in the IC, because (1) they are not relevant to the particular formal methods, (2) they are being abstracted, or (3) the IC is not capable of representing them, e.g. a construct that operates directly on complex numbers. With this mechanism, it is possible to achieve the needed simplicity of the IC without sacrificing the expressiveness of the meta-model.

The instruction set of the IC consists of a small number of 3-address instructions, very close to those used by optimizing compilers. A black box is treated as a special instruction. Each instruction performs a precisely defined action and has a simple and uniform behavior, i.e. few side-effects or special cases. Simplicity and unambiguity are the keys to ease the analysis of the model.

Figure 1(b) illustrates the Intermediate Code derived from the simple meta-model example shown in Figure 1(a).

```

process p1 {
stack<int> in,out;
void thread() {
while (1)
await[in](in.n())>=1) {
out.write(in.read()*2);
}
}
}

medium stack <type elem> {
elem data[];
int top;
elem read() {
elem value;
await[this](this.top>=1) {
value = data[top];
top = top - 1;
}
return value;
}
void write(elem e) {
await[this](true) {
top = top + 1;
data[top] = e;
}
}
}

process p1 {
stack<int> in,out;
void run() {
int i, mul;
START:
await(in.p>=1);
i = in.read(); mul = i * 2; out.write(mul);
endawait;
goto START;
}
}

medium stack <type elem> {
unbounded place p;
elem data[];
int top;
elem read(){
elem value;
await(this.p>=1);
value = data[top]; p = p - 1; top = top - 1;
endawait;
return value;
}
elem write() {
await(this.p>=0);
p = p + 1; top = top + 1; data[top] = e;
endawait;
}
}

```

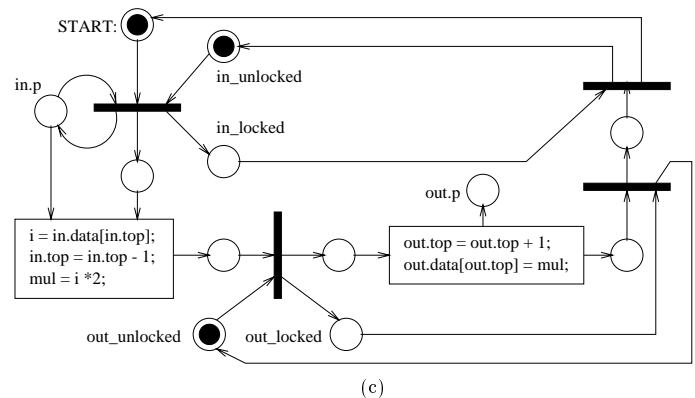


Figure 1: Example of derivation of a Petri net: (a) Meta model, (b) Intermediate Code model, (c) Petri net model

A parser and a library to manipulate Intermediate Code has been developed, and libraries to analyze Intermediate Code are being designed. We also plan to provide a translator into Petri nets along the lines discussed next.

5.1 Deriving Petri net models

By imposing some constraints on the expressive power of communication media, we can derive from the IC a Petri net that specifies the synchronization, causality, concurrency and choice skeletal behavior of the system, while abstracting away data values and operations. Petri nets are particularly interesting in our context, because they offer a suitable trade-off between expressiveness power and availability of efficient methods for analysis and synthesis [5].

The level of abstraction in which a system is modeled also affects the properties that can be verified and the computational complexity of the applied methods. With Petri nets, the execution flow within sequential processes and the synchronization among them can be analyzed at a reasonable level of accuracy. On the other hand, data calculations must be abstracted out by

including additional non-determinism that conservatively covers all possible behaviors. Thus, properties like absence of deadlock, and finite memory computation, can be verified by state-of-the-art methods.

In particular, the execution flow within a process can be modeled by a state-machine component of a Petri net that holds a token mimicking its program counter. Communication media can be modeled by places that hold messages (tokens). Finally, mutual exclusion mechanisms and schedulers can be modeled by tokens that grant access to shared resources.

By using syntax-directed translation methods, a Petri net can be automatically generated from the IC. For that, we require that the state of each communication medium that is relevant for the control structure of media and processes (e.g., used as a condition for `await` statements) be represented by a special class of variables, called *place variables*, that can only be increased/decreased by constants. This class provides the appropriate abstraction of the state of a medium that enables a direct mapping into Petri net primitives.

The `await` statement also requires some special constraints in order to be modeled by Petri nets, namely that it evaluates only conjunctions of test operations on `place` variables. With this restriction, testing the predicate of such an `await` can be simply modeled as a transition in a Petri net.

Figure 1(c) illustrates the derivation of a Petri net model from the IC model shown in Figure 1(b).

6 Status and Future Plans

We have outlined the underlying mechanisms and theoretical foundation of the Metropolis design environment and illustrated basic concepts of the design methodology built on top of them. The methodology is experimented with design drivers of various application domains, including multi-media, automotive, wireless communication, complex hardware systems, so that its applicability to a wide variety of specific issues related to domain applications is evaluated. The core mechanism is the internal representation called meta-model, which is used throughout the design phases to represent system behavior. It has precise semantics and a small number of constructs for unambiguous representations of various computation and communication semantics in a unified way. The semantics is backed by the mathematical framework based on trace structures, with which interactions of different models of computation can be formally captured.

The meta-model serves as input for all the tools built in Metropolis. The tools that we are building fall in the traditional decomposition into verification and synthesis. Verification has been our primary concern at this time and we have developed a simulator developed for an executable language. It first parses the meta-model and provides an implementation of each construct of our model using primitives of the language so that the result is semantically compatible. This is then simulated by a simulator of the language. As a prototype, we have developed a Java-based simulator, in which the `await` construct is implemented using primitives like `synchronized` or `notify`. We also plan to provide another simulator based on System C 2.0. As with VCC [6], the meta-model can be annotated with performance information generated from the architecture specification, so that the same simulator can be used to conduct performance simulation for mapped functions.

The second step of our verification plan is to integrate a wide set of tools for formal methods. Those tools will address all three aspects of a system description: computation (e.g., synthesizing a hardware for a process), communication (e.g., automatic protocol verification), and coordination (e.g., gener-

ating a quasi-static schedule consistent with a higher level of abstraction). Rather than restricting the expressiveness of the meta-model to suite these tools, we have decided to add mechanisms to it that enable particular tools applicable to subsets of the system. Section 5 has outlined the intermediate code representation and a concept called black boxes as such a mechanism. In addition, we plan to add a mechanism to formally specify properties on a part of the network that have to hold throughout the life time of the sub-network in the system. Some properties are static, e.g. the maximum number of objects to which a medium is connected, while others are dynamic, e.g. a set of valid orders of executions of the functions defined in the medium. This will be used by formal verification tools. Further, the meta-model has constructs that make it possible to syntactically analyze properties of the behavior. For example, the `bounded_loop` construct guarantees that a particular portion of the system representation satisfies some property, e.g. finite states. This information can be used to check whether a given tool can be applied to the portion.

Synthesis has large potential for decreasing design time dramatically by eliminating human errors in the successive refinement process. We believe that adapters and interfaces can be automatically built in an efficient way especially because we have used formal techniques to specify the refinement steps thus making possible to define rigorously the transformations from one level of abstraction to a subsequent one.

As we pointed out above, it is likely that the formal verification and the synthesis tools will be more powerful when applied to subsets of all the systems that can be captured by the Metropolis framework. Restricting the models of computation will provide us with a rich machinery already available in the literature. However, we would like to extend the reach of the traditional techniques for manipulating, verifying and synthesizing data-flow and finite-state machines to more complex situations where the interaction among components imply the need to include both data and control. Preliminary results lead us to believe that this approach is feasible and indeed can be the basis for a revolutionary step in system level design.

Acknowledgment

The authors are grateful to Rong Chen and Robert Clarisó for developing prototypes of the meta-model and the intermediate code, Jerry Burch and Alberto La Rosa for insightful discussion, and Ellen Sentovich and Bassam Tabbara for helpful comments in writing the initial manuscript. This work is supported by GSRC.

References

- [1] A. Sangiovanni-Vincentelli and M. Sgroi and L. Lavagno. Formal Models for Communication Based Design. In *Proceedings of Concurrency '00*, August 2000.
- [2] P. Cousot. Abstract interpretation: Achievements and perspectives. In *Proceedings of the SSRR 2000 Computer & eBusiness International Conference*, Compact disk paper 224, L'Aquila, Italy, July 31 - August 6 2000. Scuola Superiore G. Reiss Romoli.
- [3] J. Cortadella et al. Task generation and compile-time scheduling for mixed data-control embedded software. In *Proceedings of the Design Automation Conference*, June 2000.
- [4] J. Silva, M. Sgroi, F. De Bernardinis, S.F. Li, A. Sangiovanni-Vincentelli, and J. Rabaey. Wireless Protocols Design: Challenges and Opportunities. In *Proceedings of CODES '00*, May 2000.
- [5] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541-580, April 1989.
- [6] Virtual Component Co-Design Tools, version 2.0 (VCC 2.0). <http://voyager.cadence.com/techpubs/vcc/vcc20/library.html>.