Metropolis: A Development Environment to Facilitate Platform-Based Design

Abhijit Davare   Douglas Densmore   Guang Yang   Haibo Zeng
Alberto Sangiovanni-Vincentelli

Abstract—Metropolis is an integrated development environment for system-level design that embodies the concepts of the platform-based design methodology. Metropolis is based on a Metamodel with formal semantics and allows not only design capture, but also formal analysis, synthesis and simulation. After reviewing the basic concepts behind Metropolis, we will describe a realistic use-case scenario that showcases our current research efforts. We then describe architectural modeling with an example, and discuss our approaches to refinement. Next, we describe how the Metropolis framework allows for efficient design space exploration. Finally, we point out some of the main features of the Metropolis simulator.

I. INTRODUCTION

Due to the tremendous increase in design complexity during the past few years, the electronic design automation community has been forced to design at higher levels of abstraction. Other challenges include continued scaling, higher power consumption, IP integration, shorter design cycles and heterogeneous designs. Unfortunately, an ad-hoc adaptation of existing design flows will not tackle these problems successfully. Even though new languages and interchange formats are important, an overall design methodology – a design science – is key. Through the Metropolis [1] project, we aim to provide a concrete example of the platform-based design methodology [2] we advocate.

In the platform-based design approach, the separation of concerns between different aspects of the design allows for component reuse and easier integration, while enabling faster design space exploration. The three main types of orthogonal aspects are:

1) Functionality and Architecture: These two facets of the design are usually handled by separate groups within a company. Typically, a single architectural platform may serve many different functional models. A functional model contains both computation and communication elements.

2) Computation and Communication: Computation is usually design-specific while communication is standardized. Having a clearly defined separation between these two aspects of a functional description allows different communication schemes to be utilized easily. This separation also enables design space exploration and encourages component reuse.

3) Behavior and Performance: Performance indices are usually back-annotated from estimated system performance whereas behavior is dictated by system specifications.

Since different design groups usually work on these two portions of a design, this separation is useful. Also, changes in physical implementation platforms usually manifest themselves as performance changes, so this separation also provides for more implementation flexibility.

These aspects, along with a clear semantic foundation, are the main features of the Metropolis development environment. Before describing the details of Metropolis in Section III, we will compare and contrast the Metropolis approach with other well known tools and languages in Section II.

II. PRIOR WORK

Metropolis is similar to popular system-level design languages and design environments. In this section, we compare and contrast our approach to that of SystemC, Ptolemy, and Mescal.

A. SystemC: A Design Language

SystemC [3] is a popular system level design language. Typically, a SystemC design consists of a network of processes and channels. A process can communicate through ports with another process by calling interface functions (IF) defined in the shared channel. Besides this high level (more abstract) communication that uses interface function calls, SystemC processes can also communicate with other processes by sending and receiving signals. This signal-based communication is a legacy from existing hardware description languages (HDLs) such as VHDL and Verilog. Metropolis, as a system-level design language, uses port-IF for all communication purposes. Using this communication mechanism, it can easily implement signals in SystemC.

Compared to SystemC, Metropolis has several unique features. First and foremost, the Metropolis specification language – the Metropolis Metamodel – has a clearly defined semantics, unlike SystemC. This allows us to easily integrate formal tools and perform analysis on designs. For instance, we can formally check for model refinement. Second, in Metropolis, we explicitly represent the architecture platform using a separate model. This allows us to efficiently explore the design space and to refine both the functional and architectural models independently. Third, unlike SystemC, Metropolis allows constraints to be embedded as part of the design specification. This allows complex properties to be specified easily with clear semantics. Finally, to model architecture performance, an abstract notion known as
a quantity is used. A quantity has the ability to model any type of performance number, such as time, power, etc. We also allow the designer to specify quantity resolution, i.e. scheduling, in a concise manner.

B. Ptolemy & Mescal: Design environments

Several other research efforts in the CAD group in EECS at UC Berkeley are focused on system-level design. Ptolemy [4] is a research project that primarily aims to study the mathematical properties of the interaction between different models of computation. Ptolemy is more focused on modeling and analysis than on implementation and optimization. Mescal [5] is another research project that aims to develop customized hardware for software applications. Mescal focuses more on architectural modeling, where hardware is synthesized based on software requirements. In Metropolis, we advocate a meet-in-the-middle design methodology where both the architecture platform and the functional description are specified independently.

III. BACKGROUND

Metropolis offers a unified framework that embodies the concepts of platform-based design. The modularity of this framework allows developers the flexibility to add their own algorithms and tools to Metropolis. Figure 1 shows the details of the Metropolis framework. Metropolis uses the Metamodel for design capture. This modeling language has clear semantics and allows for representing different models of computation. The Metropolis compiler transforms the Metamodel into abstract syntax trees (ASTs). Developers can then use the information in these ASTs to create their own backends, which can interface with external tools.

A. Metamodel

The Metropolis Metamodel is a specification language used by Metropolis that has clearly defined semantics. It allows the designer to specify a netlist of concurrently executing objects, known as processes, that each take actions sequentially. These processes are connected using media, which represent the communication aspects of the design. To help in design capture for architectural platforms, quantity managers can specify algorithms. Also, constraints can explicitly represent properties to either be obeyed or checked.

B. Functional Model

A functional model in the Metropolis Metamodel is represented using a functional netlist. A functional netlist consists of a network of processes, linked using media. A process communicates using ports, which have associated interfaces. Media that implement these interfaces can be connected to ports. The behavior of the network is described as a set of executions, which themselves consist of a sequence of events. Events are atomic, and can be associated with the starting and ending points of actions.

The functional model utilizes services, which are just methods bundled into interfaces, to carry out computation and communication. These services can be thought of as system calls. The functional model also specifies in which order services are utilized. However, it does not represent the concurrency with which these services are carried out, nor does it specify the cost of utilizing these services. To obtain this additional information, we turn to the architectural model.

C. Architectural Model

An architectural model is realized using an architectural netlist in the Metropolis Metamodel. An architectural netlist is an interconnection of computational and communication components characterized by some services and costs, i.e. quantities. An architectural model provides services to the functional model by defining mapping processes. A mapping process is similar to a device driver, in that it sits between the hardware platform and the application. The services in an architecture are also annotated with costs by using quantity managers and their associated algorithms.

In Metropolis, an architecture model is in general divided into two netlists: a scheduled netlist and a scheduling netlist. The scheduled netlist contains all the architectural components, while the scheduling netlist contains the quantity managers [6]. Each process and medium in the scheduled netlist has a statemedia (e.g. CPU, bus, memory) associated with it. These statemedia belong to the scheduling netlist. The scheduling netlist also contains all the quantity managers.

The distinction between the scheduled netlist and scheduling netlist is based on the implementation of a scheduling interface. Each component in the architecture has some resources or physical quantities that characterize the services it can provide, i.e. what it can do, and how much it will cost. For instance, the CPU clock cycle is a quantity associated with the CPU component. Normally the scheduling netlist will implement two methods: resolve() and postcond(), which we use to model the scheduling of the resource or physical quantity the corresponding quantity manager will perform. We will describe how this is carried out in Section V-B.
D. Mapping

In Metropolis, the architecture provides certain services at a particular cost and concurrency while the functional model utilizes these services. Mapping adds explicit synchronization constraints to coordinate the interface functions that characterize these services. In this manner, it eliminates some of the non-determinism present in the functional and architectural models by intersecting their possible behaviors. After mapping, a mapped implementation is created.

Figure 2 illustrates an example of mapping in the Metropolis Metamodel. Mapping involves the creation of a new mapping netlist. This netlist instantiates both the functional and architectural netlists without modification. Then, synchronization constraints are added to the mapping netlist that coordinate the beginning and end events of the interface function calls. These constraints force the functional model to inherit the concurrency and latency defined by the architecture while forcing the architectural model to inherit the sequence of calls specified for each functional process. The synchronization constraints on the architectural side actually refer to events in the mapping processes (or tasks) which are shown in the right-hand side of Figure 2.

E. Constraints

Constraints allow different types of formal properties to be either specified or checked. Linear Temporal Logic (LTL) [7] constraints restrict the possible behavior of the system during execution. Behaviors which violate the constraint are not legal behaviors of the system. In most cases, to specify a set of properties is much easier than to implement a piece of code with those properties. So, LTL can help prototype a system rapidly. We will describe how LTL constraints are enforced in Section VIII. Synchronization constraints used in mapping are an example of LTL constraints.

Logic of Constraints (LOC) is the second way in which constraints can be represented in the Metropolis Metamodel. LOC constraints are checked during an execution trace, they are not enforced. Metropolis currently has several backends that allow the checking of traces both online and offline. Since only the traces are checked, not the system itself, LOC checking is relatively cheap. Full-blown model checking makes much stronger claims about system properties, but its PSPACE complexity rules out evaluating industrial-size designs.

IV. Use Case Scenario

In this paper, we focus on the current research activities in Metropolis that will support the following use case scenario. The scenario will highlight the separation of concerns methodology that we feel is necessary to tackle complex systems design.

A. Functional Description

First, the designer specifies the functionality of the design as a network of processes and media using the Metropolis Metamodel. At this point, the target architecture and the implementation details of the inter-process communication schemes are not fixed. In this scenario, the communication channels can first be modeled as unbounded FIFOs. However, the designer is still able to debug the functional model using the formal verification backends and the Metropolis simulator.

B. Choosing an Architecture

Once the designer is satisfied that the functional description is correct, the target architecture can be chosen. The chosen architecture must provide all of the services that are utilized in the functional model. Initially, the chosen architecture can be an abstract model. Later, through architecture refinement, architectures that are better suited to the functional model or architectures that are closer to implementation can be utilized instead. Since the interface defined by the services remains the same, refined architectures can be substituted easily. An example of architecture modeling will be described in Section V. Architecture refinement will be described further in Section VI.

C. Choosing a Communication Scheme

Once an architecture has been chosen, an inter-process communication scheme (e.g. bounded FIFO, shared memory) can be assigned to each communication channel. The amount of memory allocated to each channel (or subset of channels) can also be decided. Depending on the interplay between the functional model and the architecture, these decisions can dramatically influence the speed and the memory usage of the mapped implementation.

Our approach in Metropolis is to leverage the separation of concerns between computation and communication in the Metamodel to create a library of commonly used communication schemes. By automatically specifying mapping constraints, the designer is able to quickly evaluate different points within the design space. Based on the performance of these mapped implementations with respect to speed, memory usage, power, or any other relevant quantity, the appropriate schemes and channel sizes can be chosen. This step will be described further in Section VII.

D. Realizing the Implementation

Once the designer is satisfied with the level of performance obtained with the mapped implementation, the design can be transformed into an appropriate standard representation for
V. ARCHITECTURAL MODELING

In this section we will describe an example of how an architecture can be described using the Metropolis Metamodel. First we define a simple abstract architecture which is composed of a CPU/RTOS component, a bus, and a memory. Taking this abstract architecture as an example, we then describe the execution model of the architecture with respect to quantity annotation.

A. Example: An abstract architecture

In our abstract architecture model, we have three types of statemedia - a CPU/RTOS combination, a bus and a memory. In addition, software tasks known as mapping processes are also added. The mapping processes are connected to the statemedia and use their services while also serving as the interface between the architectural components and the functional model. The CPU/RTOS component is connected to the bus. The bus is a multi-master multi-slave communication device that allows the CPU/RTOS to communicate with the memory. The memory is a slave device connected to the bus. A diagram of our abstract architecture is provided in Figure 3.

1) Quantity managers: The CPU/RTOS component in our architecture is shared among several tasks or mapping processes. Each of them may require a service. When more than one request is issued to the CPU, arbitration is needed. We can think of the CPU as a device that has some quantities associated with it. The CPU time is one such quantity that must be distributed among the tasks. As tasks request the CPU, the request is passed to the CPU time quantity manager which decides which task will be granted ownership of the CPU. The same thing happens when more than one master asks for ownership of the bus. In this case, of course, the bus time quantity manager will decide the owner. In this architecture model we also allow the possibility of requesting exclusive access to a specific address. This is modeled by another quantity manager: the memory access manager. Each time a specific address is requested by some task, the manager will decide if the task can access the memory location or if it is locked by another task. The manager will also arbitrate between multiple requests.

2) Implementation: The CPU, bus, or memory media in the scheduled netlist can issue requests to the quantity managers in the scheduling netlist by calling the request() method of the corresponding statemedium. The user defines the statemedium and the implementation of the request() method. The statemedium request method will then call the request() method implemented in the quantity managers. This method will store all requests in order to make a decision on which one will be satisfied.

Normally, the quantity manager will implement two methods: resolve() and postcond(), which we use to model the scheduling of the resource or physical quantity that the corresponding quantity manager will perform. With these methods, the quantity manager can resolve all conflicts between the requests and decide the next valid event vector that will make the state transition in the scheduled netlist. In order to specify the event vector e, the quantity manager can call the methods SetMustDo(e) and SetMustNotDo(e) implemented by the statemedium connected to the tasks. In this way, the designer is able to specify the information flow between the media and schedulers in the architecture netlist.

B. Architecture Network Execution Model

In general, at each step of the execution of the architecture network, there are many event vectors that can happen next. Some of these events may be subject to quantity constraints. In this section we will explain the following two-step method to resolve quantities at each state of network execution.

1) Quantity requests: For each process, for each event e that it can take, find all quantity constraints on e. In the Metropolis Metamodel, this is done by explicitly requesting quantity annotations at the relevant events, i.e. Quantity.request(event, requested quantities).

2) Quantity resolution: Find a vector consisting of the candidate events and a set of quantities annotated with each of the events, such that the annotated quantities satisfy all quantity requests and all axioms of the quantity types. In the Metropolis Metamodel, this is done by letting each quantity type implement a resolve() method. The methods of relevant quantity types are called iteratively.

Semantically, a scheduler can be considered to resolve the quantity of execution index. Our two-step approach is similar to how schedulers work, such as OS schedulers, bus schedulers, or bus bridge controllers. Let’s consider the following scenario in our abstract architecture network as an example.

There are two software tasks T1 and T2 in the architecture. The functional netlist is composed of one producer and one consumer that share a FIFO, the producer is mapped on T1 while the consumer is mapped on T2. At some point the producer calls the service function release_data() provided by the CPU/RTOS, which is synchronized with the cpuWriteProtect() method in the CPU/RTOS. In this method, two labeled events
are defined: *cr* and *mr*. The first is used to make a request to the CPU time quantity manager. The request method of the statemedium connected to the CPU is called with the CPU request class parameter. This is a data structure used to store the related information, such as the begin event, the end event and how many CPU clock cycles this request may cost. The implementation of the request method of the statemedium simply calls the request method of the CPU quantity manager. At this point, the request is simply stored in the pending list. Control now passes to the scheduling netlist (for instance, assuming that the consumer is waiting a synchronization event). Now the scheduling netlist is executed and the resolve methods of the quantity managers are called cyclically until the solution stabilizes. The result, in our simple case, would be that the producer obtains ownership of the CPU in the next state. The request coordinated with the event *mr* is now executed and control again passes to the scheduling netlist.

A. Refinement Methodology

In order to systematically refine a model, there must be a methodology in place which demonstrates the procedure for a designer to follow in order to perform various refinements. In following such a procedure, ideally one can say *a priori* which properties will hold between the abstract and refined model. One can therefore follow various procedures depending on which properties are of interest.

The initial refinement procedures to be described are what we term *vertical* or *horizontal* refinement. *Horizontal* refers to the fact that the goal of the refinement is to move aspects of the model from the scheduling netlist into the scheduled netlist. This represents refinements geared toward physical implementation. *Vertical* refinement refers to the process of transforming items in the scheduled netlist. This typically is done by targeting one particular process or media element and decomposing it into multiple media and process elements and then replacing the decomposed structure back into the model. *Vertical* refinement is geared toward changing how a particular service provided by the architecture is carried out. Finally, there is a third “axis of refinement” which we term *depth* refinement. This is where a single process or media is changed internally (i.e. a scheduling algorithm changes or a media service is modified). Naturally a *hybrid* approach can be taken in which vertical, horizontal, and depth aspects of refinement can be combined. These areas will be touched on individually in the following sections.

1) **Horizontal Refinement:** Horizontal refinement is the moving of the scheduler (quantity manager) functionality into the scheduled netlist. The spectrum of horizontal refinements really results from how many of the schedulers are moved and the portion of the schedulers that are moved. This is done in order to reduce the number of elements resolving quantities and focus that effort more globally which reflects a more implementation based view.

2) **Vertical Refinement:** Vertical refinement is the notion that changes are made *vertically* within the netlist. It is not swapping aspects between netlists but rather moving within a particular netlist. Naturally this contrasts to horizontal refinement. Vertical refinement of an architecture can be seen as a whole spectrum of refinements with the abstraction levels being defined according to which elements are passive (media) and which are active (processes). For example, one could change the number and types of processes or the number and type of media in the scheduled netlist. The primary method of vertical refinement is the addition of media. This ultimately is the addition of architecture services at a different level of granularity compared to the abstract services provided initially.

3) **Depth Refinement:** Depth refinement is where individual processes or services are themselves changed. The canonical example is a scheduling policy change from round robin to earliest deadline first (EDF). New objects are not added but rather the internal behavior of a process or media is modified. If this behavior can be expressed by the decomposition of the current object into multiple objects then this might be better expressed as a *vertical* refinement.

4) **Hybrid Refinement:** Hybrid refinement is a combination of other refinement methods. The goal of any of these refinement methods is to determine a set of properties that are held or not held depending on the refinement style. Ultimately this will determine which refinement methodology is employed. Figure 4 shows the various refinement styles.

B. Formal Refinement Verification

Once a refined architecture model has been created, we must verify that it can safely be substituted in place of the original model. Often this check is performed in an ad-hoc manner through simulation with various test cases, which hope to stress the typical and corner case usage for the model. However, due
to the clear semantics of the Metropolis Metamodel, this refinement check can be done formally through the creation of a backend tool which can be run on the model.

1) Refinement Theory: The notion of refinement verification for this paper stems from that of Hierarchical Verification in [9]. We will define the problem as follows:

A model is generically defined as an object which can generate a set of finite sequences of behaviors, B. One of these possible finite sequences is considered a trace, π. Given a model X and a model Y, X refines the model Y, denoted X ⊣ Y if, given a trace π of X, the projection π[ObsY] is a trace of Y. A trace, π is considered a sequenced set of observable values for a finite execution of the module. A projection of a trace, π[ObsY], is the trace produced on Module Y for the execution which created π over the observable variables of Y. The two modules X and Y are trace equivalent, X ≈ Y, if X ⊣ Y and Y ⊣ X.

The answer to the refinement problem (X, Y) is YES if X refines Y and otherwise NO.

2) Formal Refinement in Metropolis: As mentioned previously, a trace, π is considered a sequenced set of observable values for a finite execution of the module. In the case of Metropolis, the key observable values that we are concerned with are function calls to media. By checking these, we can check refinement of single threaded processes. This paper will refer to a trace consisting of function calls to media as a TraceM. According to the semantics of Metropolis, processes must communicate strictly via media. Ultimately, the behavior of a process can be characterized by the sequence by which it makes these calls. Figure 5 shows the interaction between a process and a medium. This shows what is observable to the rest of the system is the use of the media interface by this process.

In order to characterize the Metropolis trace, TraceM, the key structure to be obtained from the model will detail the ways in which these sets of observable events can occur. This structure is a Control Flow Automaton (CFA) representation of a Metropolis Model. Metropolis has an Action Automata specification underlying it [1] but this provides much more information than is required and its structure is not suitable for use in our refinement scenario.

A CFA is defined as a very much like [10]. It is a tuple <Q, q0, X, Op, →>. Q is a finite set of control locations. These will be determined by the Metropolis model structure. q0 is the initial control location, X is a set of variables, and Op are operations which denote either (1) function calls to media (2) basic blocks of instructions ending (3) basic block of instructions ending. This ending and beginning notion is taking from the Action Automata semantics. A basic block is defined in the traditional sense, meaning a section of code in which there is no conditional execution which could result in a different execution sequence. A basic block can simply be viewed abstractly as a function call. It is for this reason that the start and end are denoted. This way, the CFA can be augmented with the body of the function call, inserted between the beginning and end portions.

An edge (q, Op, q’) is a member of a finite set of edges and the transition relationship, →, is defined as (Q × Op × Q). A edge makes a transition based on the Op present, q →Op q’.

Ideally, a CFA is created which represents the model and corresponding automata are created which represent the state of variables in the automata. For example, a model may have a loop in which it checks the value of a particular variable. The CFA would have a variable, v ∈ X, which has an automata that can be queried as to the value of that variable to determine which edges can be transitioned. Figure 7 shows one possible representation that could be used to show the incrementing of an integer with a functional range of 0 to 2.

Figures 6 and 7 demonstrate a code snippet and the resulting Metropolis CFA as defined in this paper.

```c
void thread(){
  int x = 0
  while (x<2){
    port1.callRead();
    x++;
  }
  port2.callWrite();
}
```

Fig. 6. Metro Code Example

![Fig. 5. Function Calls to Media as Observable Variables](image-url)

![Fig. 4. Refinement Methodology Examples](image-url)
Once a CFA is defined, a Trace_M is nonempty word π₁...n over the alphabet of Q control locations such that a_i → a_{i+1} for all 1 ≤ i ≤ n. We can check for trace containment between models either manually (in the case of small CFAs) by examining cycles in the CFA or automatically with the use of tools which convert the CFA into languages (e.g. Reactive modules [11]) which can be automatically checked for refinement. For a full case study using this methodology we refer the reader to [12].

C. Netlist Refinement

Finally, once a model has been created and verified, netlist refinement is the process of defining the refinement relationship between an original object and a netlist of objects (its refinement) via the Metropolis refine statement. The netlist structure of the refinement takes as an input the object that it is considered a refinement of. It then creates a refinement netlist and establishes the relationship between the two objects, creates the internal objects necessary to constitute the netlist, and establishes the connections necessary between the existing objects as well as the new connections required for objects interacting with the object being refined. Figure 8 demonstrates the syntax of this operation.

Figure 9 demonstrates this refinement. What is shown is an abstract memory element which has infinite capacity. The refinement allows for a fixed capacity element with additional control to be substituted into the netlist automatically. The elements show represent the Yapi and TTL relationship from [13].

VII. MAPPING AND COMMUNICATION SYNTHESIS

Once architectures have been modeled and possible refinements created, the mapping stage can associate a particular architecture with the functional model. Based on the architecture, different communication schemes can also evaluated against the required design objectives.

A. Communication Schemes

The functional model consists of both processes and communication channels. The processes are highly design-dependent and require designer expertise to be modified. The communication channels, on the other hand, are highly standardized. Many functional models share a common set of communication primitives such as FIFO or shared memory. Also, due to the clear separation between computation and communication in the Metropolis Metamodel, communication channels can easily be replaced without changing the functional processes.

In the use case scenario described in Section IV, the functional model first utilizes unbounded FIFOs for the communication channels. Since no architecture can implement a FIFO of infinite length, a mapped design must necessarily have finite-length FIFOs. The size of these FIFOs needs to be chosen based on the trade-off between memory usage and the desired system design space exploration in an efficient, flexible, and reusable manner.
Fig. 10. Fine grain functions used by the TTL netlist

Fig. 11. Overview of simulation algorithm

performance. Choosing smaller FIFOs may force a particular interleaving of the reader and writer processes, adding latency to the system. Larger FIFOs loosen the interleaving constraints but may also consume excessive system memory.

Changing the communication scheme from unbounded to bounded FIFO also requires the addition of additional coordination. Figure 10 shows an example of the TTL [14] bounded FIFO protocol netlist which has replaced the existing Yapi [13] unbounded FIFO channel. The interface functions of the original unbounded FIFO: read() and write() must now be implemented as a series of finer granularity functions. This translation between the interfaces is implemented within the Yapi2TTL and TTL2Yapi blocks. Note that this communication refinement implies that the architecture used for mapping must support a larger set of services.

Besides using smaller FIFOs, memory can be saved by combining independent FIFOs together into a shared memory block. Now, the same memory can be used by different channels at different times. If the pattern of writes and reads alternates between the two channels, then the amount of memory required can be reduced.

Finally, memory mapping is another means of modifying the performance of a mapped implementation. Memory mapping refers to a designer-specified allocation of FIFOs or shared memory blocks to physical memory locations in the target architecture. The rationale behind this method is that allocating FIFO locations in contiguous memory may lead to faster writes and reads.

Each of these communication schemes can be placed into a communication library. This will allow designers to quickly import existing schemes into their design.

B. Automatic constraint generation

Manually specifying the synchronization constraints needed for mapping is tedious and error-prone. Therefore, Metropolis supports a more designer-friendly interface to the mapping procedure. This interface automatically transforms the designer’s mapping decisions into synchronization constraints, allowing the designer to quickly and easily see the results of these decisions.

Mapping decisions relate to the association of functional processes with architectural tasks and the communication mechanisms used in communication channels. Functional processes and architectural processes can have a one-to-one or many-to-one association. A one-to-many association is not useful since each process contains only a single thread of control. This decision can be taken by the designer based either on intuition or an analytical model of the system. Also, any supported communication scheme can be chosen by the designer for use in the functional netlist. Based on the choice of the scheme, the designer may have to specify the channel buffer size, FIFO memory location, task priority, and other parameters for use in the synchronization constraints. Metropolis provides a simple interface to these mapping decisions by allowing the designer to fill in parameters for a mapping constructor. The parameters are transformed into synchronization constraints according to the method specified in the communications library. This also helps future automation of mapping decisions by concentrating design options into a concise representation.

VIII. METROPOLIS SIMULATOR

In Metropolis, a system is modeled with functional netlist, architecture netlist and mapping. In each netlist, LTL and LOC constraints can be used to confine and check the behavior. Simulating such a compound system is very challenging.

In the behavior and architecture netlists, each process generates a sequence of events. The execution of a process is to move from one event to the next. The system behavior is composed with all sequences of events of all processes, which we call an event vector. However, it is not the case that these sequences of events can be arbitrarily interleaved, because the semantics of some of the Metamodel constructs imply mutual exclusion of some events. For example, LTL constraints and quantity resolution result restrict the legal event sequences, a mapping netlist requires the simultaneity of pairs of events, etc. The task of the simulator is to enforce these restrictions and demonstrate only a legal event vector.

Conceptually, the Metropolis simulator has two alternating phases. Processes in a designer’s system run in one phase and the simulation manager runs in the other phase. Figure 11 summarizes this flow.

The simulation manager controls the simulation flow. It keeps track of all events that can legally take place at a given point of the execution, and uses a series of functions to determine the order of these events. This series of functions includes a mapped events check, quantity resolution results check, LTL constraint enforcement, Metamodel semantics check, and an LOC constraint check.
For mapped events, the simulator will check whether all events in the same mapped event group are ready to proceed. If any one of them is not, the whole group will be disabled. Sometimes, associated with a group of mapped events, there are conditions only under which the group can proceed together. These conditions are also checked in this step.

Quantity resolution algorithms are completely specified by designers when the quantities are defined. Designers should make sure that all resolution functions can together result in a fixed point. The simulator’s task in quantity resolution is to make sure that the quantity resolution functions are called at the right time. Thereafter, based on the quantity resolution result, some of the events may be disabled.

The most challenging part of simulation is the enforcement of LTL constraints, since these constraints are declarative and it is not obvious how to enforce them in the simulation code. If we can transform the declarative constraints into operational ones, then the simulator can handle them more easily.

Toward this direction, we use Büchi Automaton (BA) to represent LTL constraints. In fact, LTL-to-Büchi translation is a well-studied topic in the formal verification domain [15][16]. We have integrated an existing translation tool [15] into the Metropolis simulator. Suppose the BA is already constructed for LTL constraints. The simulator needs to keep track of two sets of states, those of all Metamodel processes and those of the BA. In a given system state, there may be several enabled event vectors. The simulator needs to choose one of them, and move the system to the next state. In addition, the simulator must make sure that the chosen event vector enables at least one transition in the BA, and it has to update the state of the BA. If the BA happens to be non-deterministic, the simulator needs to maintain a set of possible current states of the BA. This technique is known as on-the-fly subset construction.

The Metamodel semantics check is mainly for interface function calls and await statements. Internal data structures about the usage of IFs (being used and being prohibited by processes) are stored in communication media. By checking this information, some of the IF calls events might be delayed.

Unlike the above checks that can affect the event vector, LOC is used to monitor the event vector and verify whether the system satisfies a set of LOC properties. This monitoring is achieved through an executable LOC checker [17]. This checker can work in one of two modes. One is an offline mode, where the checker takes simulation traces generated by the simulator and checks the LOC properties. This mode allows checking different properties in one simulation run. The other mode is the online mode, where the checker is integrated into the simulator and reports LOC violations on-the-fly. This mode can speed up the design and debugging stages of a system by reporting property violations early.

IX. Conclusions and Future Work

In this paper, we have presented a vision of how the Metropolis Design Environment can support a realistic use case scenario. Starting from an architectural model, we show how we can refine this model and map it with the desired functional description. Through the multiple backends in Metropolis, we allow interfaces to formal verification tools and a clear path to physical implementation. By advocating a clear separation of concerns throughout the design process, we can quickly explore the design space and reuse components.

The current abstract architecture model is fairly simple and is used to test various mechanisms such as quantity resolution and mapping coordination. Future work is to develop an architecture component library with more generality and modularity. There are a few issues that still need to be considered. One is what kind of features an architecture component should specify, such as computation, communication, and coordination. Another is how we model the interfaces between architecture components, i.e. how to formally define the component interfaces such that we know two components can be integrated. The current interface functions do not include such characteristics.

Future work in the area of refinement focuses on expanding formal techniques to media objects and targeting implementation platforms (i.e. Xilinx Vertex II Pro). As mentioned, the current formal methods verify processes and their behavior as defined by function calls to media. By definition this will not be sufficient for media themselves. What is needed is an approach to characterize services provided by media and a method to ensure that substituted services do not violate a defined refinement relationship. Targeting implementation platforms will entail creating refinement relationships between abstract models and APIs which reflect services provided by existing hardware platforms so that there exists a direct path to implementation. This could be used to quickly estimate the performance of Metropolis architecture on off-the-shelf components.

In the area of design space exploration for communication schemes, we are exploring methods of automating the scheme and channel size assignment process. Exact algorithms are impractical due to their combinatorial complexity. We are looking at two types of heuristics, one using information derived from traces (in the spirit of LOC) and a second method based on an analytical model of the mapped implementation.

X. Acknowledgments

We would like to thank the members of the Metropolis team for providing the ideas, support, and infrastructure needed to carry out this research: Felice Balarin, Jerry Burch, Luca Carlsoni, Rong Chen, Xi Chen, Robert Clariso Viladrosa, Jordi Cortadella, Fernando De Bernardinis, Erwin de Kock, Alberto Ferrari, Daniele Gasperini, Gregor Gossler, Harry Hsieh, Timothy Kam, Arjan Kenter, Mike Khshinevsky, Alex Kondratyev, Wido Kruijtzer, Dunny Lam, Alberto La Rosa, Luciano Lavagno, Yanmei Li, Radu Marculescu, Grant Martin, Mark McKelvin, Trevor Meyerowitz, John Moondanos, Andy Ong, Claudio Passerone, Roberto Passerone, Claudio Pinello, Alessandro Pinto, Sanjay Rekhi, Ellen Sentovich, Marco Sgroi, Greg Spirakis, Laura Vanzago, Ted Vucurevich, Yosinori Watanabe, and Howard Wong-Toi.

REFERENCES