A Next-Generation Design Framework for Platform-Based Design

Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto,
Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, Qi Zhu
{davare, densmore, tcm, apinto, alberto, guyang, zenghb, zhuqi}@eecs.berkeley.edu

***Abstract*— The platform-based design methodology [1] is based on the usage of formal modeling techniques, clearly defined abstraction levels and the separation of concerns to enable an effective design process. The METROPOLIS framework embodies the platform-based design methodology and has been applied to a number of case studies across multiple domains. Based on these experiences, we have identified three key features that need to be enhanced: heterogeneous IP import, orthogonalization of performance from behavior, and design space exploration. The next generation METRO II framework incorporates these advanced features. The main concepts underlying METRO II are described in this paper and illustrated with a small example.**

I. INTRODUCTION

The design of embedded systems is becoming more difficult as design complexity increases, time-to-market pressures continue, and development teams with diverse backgrounds are assembled. The platform-based design methodology (PBD) [1] is a technique to combat these challenges. This methodology advocates the separation of concerns between an architectural platform – a collection of architectural primitives configured to provide a set of services – and the functionality – a description of what the design does defined in terms of the same services. By taking these two portions of a design through a set of clearly defined abstraction/refinement steps which culminate in mapping, correct-by-construction design as well as structured design space exploration are enabled.

To support the design and analysis of heterogenous systems our group has developed the METROPOLIS Design Framework [2]. It is based on the ideas of PBD and orthogonalization of concerns [3] in terms of communication-computation, function-architecture, and behavior-performance. It features a flexible and formal semantics that supports a wide variety of models of computation.

In this paper, we will describe METROPOLIS in more detail and present some of the lessons learned from several case studies. These limitations will be used to drive the three main features we would like to enhance in the future: the import of heterogeneous IP, designer-friendly orthogonalization between performance and behavior, and design space exploration. In turn, these requirements motivate the development of the METRO II framework, whose semantics and building blocks are described using a small example.

II. RELATED WORK

There are multiple tools, environments, and languages related to METROPOLIS and the planned features of METRO II. A broad survey of this related work in the context of the PBD methodology is provided in [4]. Both academic and industrial approaches are summarized with respect to functional modeling, architectural modeling, and mapping. This section highlights a few of these approaches.

A. SystemC-Based Solutions

SystemC [5] is a free C++ library for modeling both hardware and software at various levels of abstraction. It is currently the most popular language for system-level design. For hardware design, it is based on the discrete-event MoC (Model of Computation). Because of this similarity, RTL designers can migrate to SystemC with little difficulty. The main synchronization mechanisms are events and global timing. For software design, C++ constructs can be used.

SystemC separates communication from computation by using port-interface calls. However, it lacks all other separations of concerns, such as behavior-performance and function-architecture. As a result, it is less efficient in modeling reusable system level designs. SystemC also has standard libraries for Transaction Level Modeling and Verification.

B. Ptolemy II

Ptolemy II [6] focuses on component-based heterogeneous modeling. It uses tokens as the underlying communication mechanism. Directors regulate how actors in the design fire and how tokens are used to communicate between them. This mechanism allows different MoCs to be constructed within Ptolemy II. Actors are specified using Java.

Ptolemy II uses hierarchical composition to handle heterogeneity. Each level in a hierarchy has a director that organizes the firing of the actors at that level. Ptolemy II has no intrinsic notion of mapping between actors or of using declarative specification in the design.

C. Rapide

Rapide [7] is an executable architecture definition language (EADL) which has advanced features for handling event patterns. Architectures in Rapide are interconnections of modules which can represent a system at any level of abstraction. Modules must conform to certain interfaces, which include facilities that are provided, facilities that are required, some definition of behavior, and constraints on the external environment. Compatible modules can be connected to each other to form an architecture.

When architectures execute, they generate timed events along with causality relationships. Event patterns can be used to recognize certain sets of events. Event patterns, in turn, can be used to specify relationships between different architectures. For instance, a specific event pattern in an abstract architecture can be used to trigger a larger event pattern in a more refined architecture. Rapide is not targeted specifically at embedded systems design, and as such there is no mention of different MoCs or function-architecture mapping.

D. SPIRIT

SPIRIT [8] is an IP-integration consortium that aims to provide a common specification mechanism for describing and handling IPs. It includes: an XML-based IP meta-data schema that leverages industry standards (such as VSIA, XSLT, and XPath), configuration and generation interfaces, and the IP-XACT methodology which uses the former two. This is currently mainly at the RTL level, but an IP-XACT methodology with ESL extensions is currently under development. The ESL requirements for the XML schema include module hierarchy support, ad-hoc connection support, multiple views of different levels for one component (e.g., TLM PV, TLM CA, etc.), supporting mixed IP modeling abstraction levels.

E. Others

1) Academic: The MILAN project [9] employs a model-based solution for hardware/software co-design and co-simulation. Different simulators can be integrated once different simulation models are interpreted into the common model supported in MILAN. It is built on top of the Generic Modeling Environment (GME) [10], a framework creating domain-specific modeling languages, and DESERT, a collection of tools which are used for navigating and pruning large design spaces in GME. Artemis, Compaan and Laura, Sesame, and Spade [11] are provided together as a workbench to model applications and SoC based architectures. Finally, MESCAL provides programming infrastructure for application specific programmable platforms [12].

2) Industrial: Cofluent Studio by CoFluent Design enables design space exploration at the transaction level using a Y-chart modeling approach. MLDesign Technologies offers MLDesigner which offers support for discrete event, dynamic dataflow, and synchronous dataflow modeling of architectures and functionality. Mirabilis Design provides the VisualSim product family which also models continuous time and FSM based systems. Finally, Synopsys offers System Studio which performs algorithm capture and performance evaluation in SystemC [5].

III. First Generation Metropolis Framework

This section describes the first generation METROPOLIS framework [13][2] in more detail. METROPOLIS features a flexible and formal semantics based upon the tagged-signal model [14] that allows it to represent a wide variety of models of computation. Furthermore, it supports: platform-based design, behavior-architecture mapping, and orthogonalization of concerns at the levels of communication-computation-coordination, architecture-function-mapping, and behavior-performance. First, the execution semantics and the specification language are covered. Next, the set of tools developed within this framework is described. Finally, the limitations observed after carrying out a number of case studies are listed.

A. MetaModel Language and Execution Semantics

The METROPOLIS MetaModel specification language is used for the specification of both the functionality and architecture. It allows for imperative as well as declarative specification.

The four main types of objects in the MetaModel are: processes, media, quantity managers, and netlists. Each *process* contains its own thread of control and executes concurrently with all other processes in the system. The execution of a process is represented by a sequence of events, where events are actions executed by processes. *Media* are passive objects that are used for inter-process communication. Each medium implements a set of interface methods. Media are connected to processes and other media by means of ports that are type compatible with their interfaces. *Quantity managers* control access to shared media or assign physical quantities such as time or power to events. Netlists are objects where the other objects are instantiated and connected. Netlists can contain other netlists.

1) Two-phase Execution: When designing with the MetaModel, a system is captured by two netlists of objects: a scheduled netlist and a scheduling netlist. The scheduled netlist consists of a number of processes and media, which form the skeleton of the system behavior. The scheduling netlist contains a collection of quantity managers, each of which can model execution costs or scheduling policies. The execution semantics of the entire system is simply the alternation between the scheduled netlist and the scheduling netlist. The interaction between the two netlists is carried out by quantity annotation requests associated with events.

For example, if two processes in the scheduled netlists require arbitration to access a common resource, each of them will generate a representative event, and send an (arbitration) quantity annotation request for that event to an arbiter (a particular quantity manager). This occurs in the scheduled netlist phase. In the following scheduling netlist phase, those quantity annotation requests will be resolved by the arbiter quantity manager. When the execution is switched back to the scheduled netlist, based on the quantity resolution results, the processes can either proceed to access the common resource or wait until the resource becomes available.

2) Declarative Specification: Processes, media, and quantity managers are described with purely imperative specification. In addition, the METROPOLIS MetaModel also supports declarative constraints. The mixture of imperative and declarative specification gives the designer additional flexibility. Two

kinds of formal constraint logics are supported: Linear Temporal Logic (LTL) [15] and Logic of Constraints (LOC) [16]. LTL is well studied in the formal verification field. It is quite expressive to specify properties along a time line. Therefore, it can be used to specify coordination among processes. LOC is particularly suited for specification of performance constraints over system behaviors.

Both LTL and LOC constraints can be interpreted either as part of the specification or as assertions. Assertions are checked by viewing simulation traces or by formal reasoning. Similarly, constraints that are part of the specification can either be used to restrict the simulation or provide input to synthesis tools.

B. Tool Support

METROPOLIS features a frontend that parses the input MetaModel language and creates an abstract syntax tree. Then, the abstract syntax tree can be passed to different backend tools for analysis. One of the most important backend tools is the simulator [17], which preserves the MetaModel semantics while translating a MetaModel specification into the executable SystemC [5] language. LTL and a set of built-in LOC constraints can be enforced during simulation [16]. For verification there are backends for: checking LOC properties [18], interfacing to the SPIN model checker [19] to verify LTL constraints, and a refinement verification tool [20]. There is also an interface to the UCLA's xPilot [21] synthesis system that works on a synthesizable subset of the MetaModel. All of the frontend and backend tools can be invoked interactively by using the METROPOLIS Interactive Shell.

C. Limitations

We have performed multiple case studies [20] [22] [23] [24] using this framework and the PBD methodology. While validating the core ideas of our approach, these case studies also revealed some limitations of the tool framework that impede wide-spread adoption.

The generality of the MetaModel language [13] creates difficulties for both users and framework developers. Expecting users to learn a new language, which lacks many of the niceties of their favorite language(s), is a burden, as is creating import and export support for a variety of languages. Furthermore, a rich new language requires extensive infrastructure support for compilation, simulation, and debugging.

Secondly, interactions with quantities must be explicitly represented, and simplifying assumptions made in domain-specific languages cannot be made in the MetaModel. Although there are conceptual and implementation distinctions between modeling costs and modeling scheduling policies with quantity managers, METROPOLIS does not make them clear in the execution semantics. The end result is that specifying quantity managers – especially the interaction between them – is a difficult task for designers

Finally, the case studies reveal that design space exploration is one of the main benefits of the PBD methodology. Carrying out design space exploration requires relating together events and analyzing the associated annotations. The MetaModel language provides support for these features, but the ease-of-use can be significantly enhanced.

By focusing on the key value-added features of METROPOLIS, and addressing these limitations, we plan to make METRO II an IP-integration framework with enhanced support for PBD activities.

IV. METRO II GOALS

Based on the experience gained from the development and usage of the METROPOLIS framework, we have identified three main features to enhance. These features form the basis of the second-generation METRO II framework. The three features are:

1) *The ability to import pre-designed IP.* IP providers develop their models using languages and tools that are domain specific. Requiring a singular form of design entry in a system-level environment results in significant effort to translate the original specification into the new language while making sure that semantics are preserved. If different designs can have different semantics, heterogeneity has to be supported by the new environment.
2) *The ability to separate cost from behavior when carrying out design.* In a system-level framework that supports multiple abstraction levels, many implementations of the same basic functionality will have the same behavioral representation at higher levels of abstraction. For instance, different processors will be abstracted into the same programmable component. What distinguishes them is the performance vs. cost trade-off. Moreover, not all metrics are optimized at the same time. It should be possible to introduce performance metrics during the design process from specification to implementation.
3) *The ability to explore the design space in a structured manner.* This requirement is divided into two main parts: facilitating correct-by-construction abstraction/refinement and efficiently relating the functional and architectural portions of the design together. The first part is crucial to guarantee that the points explored in the design space are legal.

The remainder of this section describes these three requirements in more detail.

A. Heterogeneous IP Import

This feature exposes many implementation challenges. It shapes the nature of METRO II to be primarily an integration environment. There are two main challenges that have to be addressed.

First, IPs can be described in different languages and can have different semantics that can be tightly related to a particular simulator. Importing the IP would entail providing a way of exposing its interface. The user must have the necessary aids to define wrappers that mediate between the IP and the framework such that the behavior can be exposed in an unambiguous way.

Secondly, wrapped components have to be interconnected. Even if the interfaces are exposed in a unified way, interconnecting them is not usually a straightforward process. For instance, the type of data produced by one IP can be incompatible with the type of data that the receiving IP is expecting. Type conversion is the simplest case; more challenging communication problems can arise. Consider the case of a software model of an engine controller unit, represented as a finite state machine, that interacts with a continuous time model of a car engine. The composite system is known as a hybrid system. One model (finite state machine) is untimed while the other model (continuous time) is defined as functions over time that belongs to the reals. It is completely arbitrary to introduce a sample-and-hold interface between the two. Even if this is a commonly used interface, we believe that the designer is the only one that has the knowledge of how such heterogeneous models should interact. Therefore, the design environment should provide a formal way of defining these adaptors between different MoCs rather than imposing predefined ones.

B. Behavior-Performance Orthogonalization

The specification of what a component does should be independent of *how long it takes* or *how much power it consumes* to carry out a task. This is the reason why we introduce dedicated components, called *annotators* to annotate *quantities* to events.

A distinction has to be made between quantities used just to track the value of a specific metric of interest and quantities whose value is used for synchronization. For instance, time is used to synchronize actions and it is not merely a number that is computed based on the state evolution of the system. For quantities that influence the evolution of the system, special components, called *schedulers* are provided by the glue language. Schedulers are used to arbitrate shared resources.

The separation of schedulers from annotators allows for simpler specification and provides a cleaner separation between behavior and performance. As a result, instead of two-phase execution as in METROPOLIS, the execution semantics become three-phase.

C. Design Space Exploration

Following the platform-based design approach, we want to keep functionality and architecture separate. The implementation of the functionality on the architecture is achieved in the mapping step. In order to explore several different implementations with minimal effort, the design environment needs to provide a fast and efficient way of mapping without touching the functional or the architectural models.

The main problems to tackle are related to the specification of mapping itself and to the synchronization of the two models. The behavior of a mapped model (and therefore of the implementation) is essentially defined as the intersection of the behaviors of the functional model and the behaviors of the architectural model. The intersection is obtained through synchronization of events between the two models. Such constraints introduce potential deadlocks whose causes are usually difficult to identify.

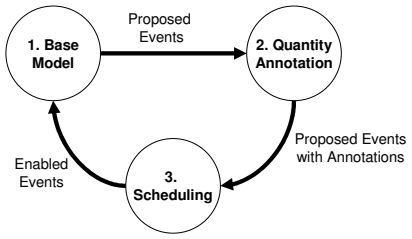


Fig. 1. Three Phase Execution in METRO II

A set of tools must be provided to help the design space exploration phase. A new set of theoretical results are needed to understand when automatic mapping can be carried out in order to guarantee optimal and correct-by-construction solutions. When manual mapping is the only viable solution, the number of constraints required may be very large. A flat graphical representation of the mapped system would not help the designer. New tools to handle hierarchical mapping and new languages to express mapping in a structured way must be developed.

The result of the mapping step is another model that refines both the function and the architecture. Checking that this function obeys the refinement relationship with the original specification is a task that should be automated by providing new verification tools that leverage the semantics of the glue language.

V. METRO II EXECUTION SEMANTICS

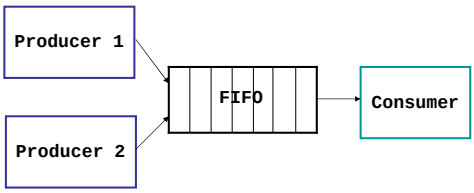
We are designing METRO II to address the design challenges in Section IV, and to overcome some of the limitations of METROPOLIS. Like METROPOLIS, the semantics of the METRO II framework will be centered around the connection and coordination of components.

The key concept underlying METRO II is an *event*. An event is a tuple $< p, T, V >$ where p is a process, T is a tag set, and V is a set of associated values. An event denotes an action taken by a process (p). Events may be associated with annotations (T) and state (V). Annotations correspond to quantities in the design, such as time or power. State includes variables that are in the scope of an event.

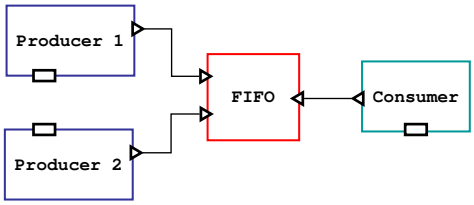
Based on the treatment of events, the design is partitioned into three phases of execution. In the first phase, processes propose possible events, the second phase associates tags with the proposed events, and the third phase allows a subset of the proposed events to execute. Figure 1 summarizes these execution semantics.

A. First phase: Base Model Execution

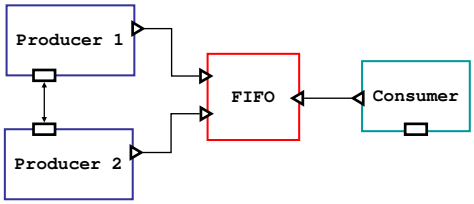
The base model consists of concurrently executing processes that block only after proposing events. A process may atomically propose multiple events – this represents nondeterminism in the system. After all processes in the base model have proposed at least one event each, the design shifts to the second phase.



(a) General View



(b) In METRO II



(c) In METRO II with Synchronization

Fig. 2. Producers Consumer Example

B. Second phase: Quantity Annotation

In the second phase, each of the proposed events is annotated with various quantities of interest. For instance, a proposed event may be annotated with local and global time tags. New events may not be proposed during this phase of execution.

C. Third phase: Scheduling

In scheduling, a subset of the proposed events are enabled and permitted to execute, while the remainder are blocked. At most one event per process is permitted to execute. Once again, new events may not be proposed during this stage.

VI. METRO II BUILDING BLOCKS

To simplify the designer's task of specifying models that conform to the three-phase semantics described in Section V, different types of objects are defined in METRO II. First, we describe the component, the primary block used for specification, and then introduce the different types of ports and connections in METRO II. After this the specialized METRO II objects are described, these are: constraints, adaptors, mappers, annotators, and schedulers.

To illustrate the function of some of these objects we use the same example as in [2], but we implement it in METRO II to highlight some of the differences. Figure 2(a) shows this example, which consists of two producers communicating with a consumer over a shared FIFO.

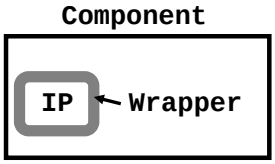


Fig. 3. Atomic Component

A. Components

A *component* is a possibly concurrent block which may contain zero or more processes. Components interface with other components via zero or more ports. There are two types of components: *atomic components* and *composite components*. An atomic component is a block specified in some language and is viewed by the framework as a black box with only its interface information exposed. A composite component is a group of one or more objects as well as any connections between them.

An atomic component with zero ports is shown in Figure 3. The IP encapsulated by the component is interfaced by means of a *wrapper*, which translates and exposes the appropriate events and interfaces from the IP.

◇ *Components in the example*

The three components in the figure are the producer, the consumer, and the FIFO. Their basic behavior is specified in SystemC. Each producer keeps writing integers to the FIFO. The consumer continually tries to read data from the FIFO, and the FIFO provides buffering between the reads and writes. Additional constraints can be added, for instance, in Figure 2(c), the writing of the two producers are synchronized to maintain the same rate.

B. Ports

There are two types of ports that components may have: coordination and view ports. Coordination ports are used for two-way interaction with other components by using events. View ports, on the other hand, may only expose internal events to the outside.

A *coordination* port is used to interact with other components. Each coordination port is associated with a set of methods. A *method* is a sequence of events, with a unique begin/end event pair. Variables in the scope of the begin event are method arguments. Variables in the scope of the end event are return values.

By setting constraints between events associated with coordination ports of different components, the execution of these components can be coordinated. Coordination ports are divided into three types based on the type of interaction: rendezvous ports, required ports, and provided ports.

1) Rendezvous Ports: Rendezvous ports can only be connected to other rendezvous ports. They are used to synchronize methods from different components. A connection between two rendezvous ports implies that the begin events of all methods in the first port occur simultaneously (same valuations for all tags) with all the begin events from the corresponding

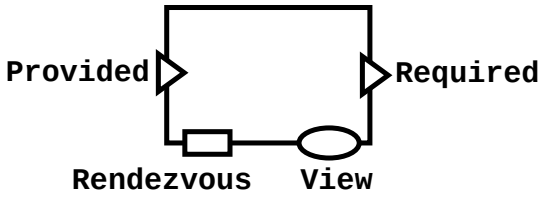


Fig. 4. Component with 4 ports

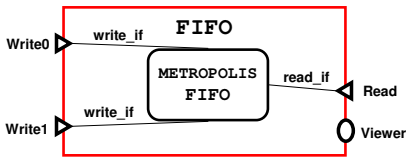


Fig. 5. Wrapped FIFO

methods in the second port. The end events of corresponding methods occur simultaneously as well.

The execution semantics of rendezvous ports is as follows. All components with connected rendezvous ports independently propose their respective begin events. These proposed events are allowed to occur if and only if all other begin events have also been proposed, otherwise they are blocked. Similarly, after executing the methods, all components independently propose end events and wait for all other end events to be proposed. Depending on the specifics of the connection, values in the scope of the begin/end events may be checked for equality or transferred between the components.

2) Required Ports: Required ports are used by components to request methods that are implemented in other components. A required port can only be connected to a provided port that provides the required methods.

For required ports, a component proposes a begin event and associates values with the proposed event that represent the arguments of the method being requested. When the proposed event is executed, control transfers to the component at the other end of the connection, which owns the provided port. The component waits for the end event to be executed and obtains the return values from the method.

3) Provided Ports: Provided ports are used by components to provide methods to other components. As stated before, connections are permitted only between a required port and a provided port.

For provided ports, no separate process exists in the component to carry out the provided method. Instead, the component inherits the process from the caller component and executes the events in the provided method using that process. After the method has been executed, the component proposes the end event.

4) View Ports: A *view port* exposes some of a component's internal events to the outside world. These events are read-only, i.e., they cannot be blocked by outside world. View ports cannot be connected to other ports.

A component with required, provided, rendezvous, and view ports is shown in Figure 4.

◇ *The FIFO Component*

Figure 5 shows how a FIFO medium in METROPOLIS might be wrapped for use within METRO II. Each provides port connects to an interface implemented by the medium, with 2 write ports connecting to the write interface of the medium. The view port provides visibility of certain events. Examples of this could be: FIFO empty, FIFO full, read finished, and write finished.

C. Connections

Connections between coordination ports are the primary means of component interaction. One-to-one port connections are allowed between a required port and a provided port, and between a pair of rendezvous ports. Rendezvous and provided ports do not need to be connected, but each required port must be connected to a corresponding provided port.

◇ *Connections in the example*

This example has required-provided port connections from the producers to the FIFO and from the consumer to the FIFO. Also, there is a rendezvous connection in Figure 2(c) that synchronizes the events in each producer such that both producers proceed at the same rate.

D. Constraints and Assertions

Constraints are used to specify the design via declarative means (as opposed to imperative specification which is used in components). Assertions are used to check whether the rest of the design conforms to given requirements. Both constraints and assertions are described in terms of events, the values associated with them, and their tags. The events referenced by constraints or assertions must be exposed by means of coordination or view ports. Depending on the logic used to describe them, constraints can be enforced either by the base model or the scheduling phases of execution. Linear Temporal Logic (LTL) [15] and Logic of Constraints (LOC) [16] will be supported by METRO II.

E. Adaptors

There are many ways of handling heterogeneous MoCs in a design. One of the most common approaches is the hierarchical composition of heterogeneous models as done in Ptolemy II [6]. With hierarchical composition, a specific MoC exists at each level of the hierarchy. To allow models in two heterogeneous MoCs to communicate, a third MoC may need to be found within which the two will be embedded.

In our experience there is a strong need to interconnect heterogeneous models at the same level. For instance, the user may want to connect the output of a base-band processing component to the input of an RF component (i.e. a dataflow model interacting with a continuous time model). This way of handling complexity does not require changing the interface of a model in order to behave like another model. This is in line with one of the our main concerns: being able to re-use IPs in different contexts.

To bridge the different semantics of heterogeneous components, we define *adaptors* as first class citizens in our language. Adaptors are used to modify events as they pass from one MoC

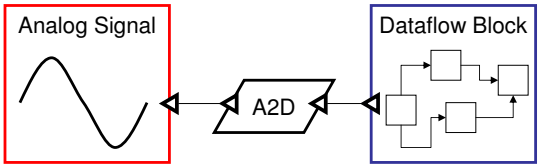


Fig. 6. An adaptor interfacing dataflow and continuous time models

to another. Denotationally, an adaptor is a relation that maps sets of events from one model to sets of events in another model.

Adaptors are connected with components through coordination ports. In the PBD methodology, they can be regarded as the bridge between heterogeneous functional components or between heterogeneous architectural components.

◇ *An Adaptor in the Example*

If a producer in the Producer-Consumer example contained a dataflow processing element and obtained its data from a continuous time component, an adaptor would be required to bridge the two MoCs. In continuous time, an event could be defined as a pair of a function and a right open interval of the reals (i.e. a time interval), $(f(t), [t_-, t_+))$. The adaptor between continuous time and data flow would relate the continuous time event with a set of samples of that function at specific times:

$$\begin{aligned}\{(f(t_i), t'_i) \quad &: \quad t_0 = t_-, t_{i+1} - t_i = T, \\ t_i < t_+ \quad &\wedge \quad t'_i < t'_j \iff t_i < t_j\}\end{aligned}$$

where T is the sampling period. This definition corresponds to sampling the continuous time signal at constant rate and storing the values in a FIFO which are then consumed by the dataflow model.

Figure 6 shows the example adaptor, called A2D, represented in METRO II. A2D is configured with the beginning times, ending times, and the period length. It transforms each request for data from the dataflow block of the producer into a request for the value of the analog signal at a particular time.

F. Mappers

When carrying out mapping, there is a many-to-one allocation of all functional components to architectural components. Similar to adaptors, mappers may be required to bridge the functional and architectural components.

The most common usage of mappers is to transform or add values in the scope of events. For instance, a functional component may have a required port whose begin event is associated with a data value. The architectural component to which the functional component is mapped has a corresponding required port. However, the expected values in the scope of the begin event of the architectural component's port may include both the data and the start address. In this case, it is the job of the mapper to specify this start address, since it is not relevant in the description of the functional component.

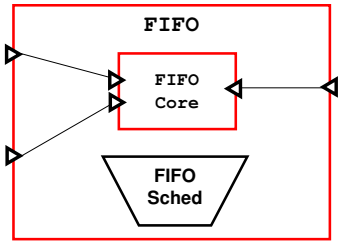


Fig. 7. FIFO with Scheduling Inside

G. Annotators and Schedulers

In METROPOLIS, both event scheduling and performance annotation is carried out with a special component called a quantity manager. It is hard to have a general mechanism to handle both scenarios since different design styles are used specify both. In METRO II, these two aspects will be separated by using *annotators* and *schedulers*.

Annotators are objects that write tags to events. Each tag is determined in terms of the event, the event's values, and any parameters supplied to the annotator. Only static parameters are permitted for annotators, which may not have their own state.

Schedulers are objects that can disable proposed events based on their scheduling policy. After the annotation phase has completed, the scheduling phase begins. Based on the scheduler's local state, the proposed events, and their values and tags, scheduling occurs which can lead to the disabling of some proposed events.

Like in METROPOLIS, annotators (schedulers) are instantiated in the netlist level. However, instead of explicit coordination or service connections, events are directly associated with individual annotators (schedulers). It is important to note that now these associations are specified at the netlist level instead of directly in the component as in METROPOLIS. The specification of these associations is static. When these are specified they are directly registered with the appropriate annotator or scheduler. Multiple annotators may be associated with a single event, as long as they write different tags. If multiple schedulers are associated with a single event, then they must agree on their decisions before switching back to the base model.

◇ *Example 1: Scheduling in the Example*

Figure 7 shows the FIFO refined to be a composite component that contains a scheduler. The scheduler connects to the events on the individual ports and ensures that race conditions do not occur between the two producers.

◇ *Example 2: Global Time*

Global time is a special quantity. Besides the performance annotation aspect of global time, the other concept of scheduling exists in almost all systems. For example, in synchronous languages, the logical time denotes the global execution ordering. This kind of global ordering is important to synchronize the entire systems and sometimes crucial to capture the correct behavior. In METROPOLIS, there is a global time quantity manager which provides both the scheduling and annotation

aspects.

As discussed before, we want to separate scheduling from annotation in METRO II. More specifically, the GlobalTime scheduler will take care of only the scheduling part; there is a separate performance annotator which will charge the time for a particular service. If a component gets a performance annotation from a performance annotator, it then sends to GlobalTime scheduler that number and its current global time. This seems like just a simple twist of what we are doing in METROPOLIS, but it makes the two natures of global time separate.

VII. CONCLUSIONS

The Platform-based design methodology imposes a number of requirements on system-level design frameworks. METROPOLIS represents the first attempt at such a framework. To address the limitations of METROPOLIS, in this paper we identified three main features that must be enhanced and described how the next generation METRO II framework will support them. The aim is to develop a framework that supports the import of heterogeneous IP, facilitates behavior-performance orthogonalization, and eases design space exploration. This is achieved by building an integration framework based on events with three separate phases of execution.

We are currently implementing the semantics of METRO II based on the SystemC kernel, and developing further case studies to exercise its capabilities.

VIII. ACKNOWLEDGEMENTS

This work is supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award #CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, National Instruments, and Toyota. This work is also supported by the MARCO-sponsored Gigascale Systems Research Center (GSRC).

We would like to thank Felice Balarin, Yaron Kashai, Luciano Lavagno, Claudio Pinello, Stavros Tripakis, and Yosinori Watanabe for helpful discussions about METRO II.

REFERENCES

[1] A. Sangiovanni-Vincentelli, "Defining platform-based design," *EE Design*, March 2002.

[2] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer Magazine*, pp. 45–52, April 2003.

[3] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: Orthogonolization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, December 2000.

[4] D. Densmore, R. Passerone, and A. Sangiovanni-Vincentelli, "A platform-based taxonomy for esl design," *IEEE Design and Test of Computers*, vol. 23, no. 5, pp. 359–374, 2006.

[5] "Open systemc initiative web site: http://www.systemc.org."

[6] C. Brooks, E. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Z. (eds.), "Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii)," University of California, Berkeley, Tech. Rep. UCB/ERL M05/21, July 2005.

[7] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using rapide," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336–355, Apr. 1995.

[8] "Spirit consortium website: http://www.spiritconsortium.org."

[9] A. Bakshi, V. Prasanna, and A. Ledeczi, "MILAN: A model based integrated simulation framework for design of embedded systems," in *Proceedings of Workshop on Languages, Compilers, and Tools for Embedded Systems*, June 2001.

[10] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," in *IEEE Workshop on Intelligent Signal Processing*, May 2001.

[11] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–112, 2006.

[12] A. Mihal, C. Kulkarni, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, C. Sauer, K. Vissers, and S. Malik, "Developing architectural platforms: A disciplined approach," *IEEE Des. Test*, vol. 19, no. 6, pp. 6–16, 2002.

[13] MetropolisDesignTeam, "The metropolis meta model version 0.4," in *Technical Memorandum UCB/ERL M04/38, University of California, Berkeley, CA 94720*, September 14, 2004.

[14] E. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, pp. 1217–29, December 1998.

[15] A. Pnueli, "The temporal semantics of concurrent programs," *Theoretical Computer Science*, vol. 13, pp. 45–60, 1981.

[16] G. Yang, H. Hsieh, X. Chen, F. Balarin, and A. Sangiovanni-Vincentelli, "Constraints assisted modeling and validation in metropolis framework," in *Proceedings of The Asilomar Conference on Signals, Systems, and Computers*, Nov. 2006.

[17] G. Yang and et al., "Separation of concerns: Overhead in modeling and efficient simulation techniques," in *Fourth ACM International Conference on Embedded Software*, September 2004.

[18] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Logic of constraints: A quantitative performance and functional constraint formalism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 23, no. 8, Aug. 2004.

[19] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.

[20] D. Densmore, S. Rekhi, and A. Sangiovanni-Vincentelli, "Microarchitecture development via metropolis successive platform refinement," in *Design Automation and Test in Europe (DATE)*, February 2004.

[21] "The xpilot system, http://cadlab.cs.ucla.edu/soc."

[22] H. Zeng, A. Davare, A. Sangiovanni-Vincentelli, S. Sonalkar, S. Kanajan, and C. Pinello, "Design space exploration of automotive platforms in metropolis," in *Society of Automotive Engineers Congress*, April 2006.

[23] A. Davare, Q. Zhu, J. Moondanos, and A. Sangiovanni-Vincentelli, "Jpeg encoding on the intel mxp5800: A platform-based design case study," in *IEEE 2005 3rd Workshop on Embedded Systems for Real-time Multimedia*, September 2005.

[24] D. Densmore, A. Donlin, and A. Sangiovanni-Vincentelli, "Fpga architecture characterization for system level performance analysis," in *Design Automation and Test Europe 2006*. DATE, March 2006.