METROII: A Design Environment for Cyber-Physical Systems

ABHIJIT DAVARE, Intel Labs
DOUGLAS DENSMORE, Boston University
LIANGPENG GUO, University of California
ROBERTO PASSERONE, University of Trento
ALBERTO L. SANGIOVANNI-VINCENTELLI, University of California, Berkeley
ALENA SIMALATSAR, École Polythechnique Fédérale de Lausanne
QI ZHU, University of California, Riverside

Cyber-Physical Systems are integrations of computation and physical processes and as such, will be increasingly relevant to industry and people. The complexity of designing CPS resides in their heterogeneity. Heterogeneity manifest itself in modeling their functionality as well as in the implementation platforms that include a multiplicity of components such as microprocessors, signal processors, peripherals, memories, sensors and actuators often integrated on a single chip or on a small package such as a multi-chip module. We need a methodology, tools and environments where heterogeneity can be dealt with at all levels of abstraction and where different tools can be integrated. We present here Platform-Based Design as the CPS methodology of choice and METROII, a design environment that supports it. We present the metamodeling approach followed in METROII, how to couple the functionality and implementation platforms of CPS, and the simulation technology that supports the analysis of CPS and of their implementation. We also present examples of use and the integration of METROII with another popular design environment developed at Verimag, BIP.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; F.1.2 [**Computation by Abstract Devices**]: Models of Computation

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Platform-Based Design, Heterogeneous Embedded Systems, Cyber-Physical Systems, Modeling, Multiprocessor, System-on-Chip

ACM Reference Format:
Davare, A., Densmore, D., Guo, L., Passerone, R., Sangiovanni-Vincentelli, A. L., Simalatsar, A., and Zhu, Q., 2013. METROII: A design environment for cyber-physical systems. ACM Trans. Embedd. Comput. Syst. 12, 1, Article 49 (March 2013), 31 pages.
DOI = 10.1145/2435227.2435245 http://doi.acm.org/10.1145/2435227.2435245

1. INTRODUCTION

Cyber-physical systems (CPSs) are integrations of computation and physical processes. Embedded computers and networks monitor and control the physical processes, usually

This work is supported by the FCRP GSRC and MUSYC Centers, Intel, General Motors, and UTC, and the EU projects COMBEST (n. 215543) and ArtistDesign (n. 214373).
Authors' addresses: A. Davare, Intel Labs, USA; email: abhijit.davare@intel.com; D. Densmore, Boston University; email: dougd@bu.edu; L. Guo, University of California, Berkeley, CA; email: glp@eecs.berkeley.edu; R. Passerone, University of Trento, Italy; email: roberto.passerone@unitn.it; A. L. Sangiovanni-Vincentelli, University of California, Berkeley, CA; email: alberto@eecs.berkeley.edu; A. Simalatsar, École Polythechnique Fédérale de Lausanne, Switzerland; email: alena.simalatsar@epfl.ch; Q. Zhu, University of California, Riverside, CA; email: qzhu@ee.ucr.edu.

DOI 10.1145/2435227.2435245 http://doi.acm.org/10.1145/2435227.2435245

with feedback loops where physical processes affect computations and vice versa. When studying CPS, certain key problems emerge that are rare in so-called general-purpose computing. For example, in general-purpose software, the time it takes to perform a task is an issue of performance, not correctness. In CPS, the time it takes to perform a task may be critical to correct functioning of the system. Moreover, many of the technical challenges in designing and analyzing embedded software stem from the need to bridge an inherently sequential semantics with an intrinsically concurrent physical world.

In the CPS world, it is important to model both functional and architectural aspects of the design that impact timing and other physical properties of the design such as power consumption, cost, and weight. And it is essential to model the environment as it determines the correctness and the robustness of the design. Given the increasing demands on the functionality of the design, its safety, cost and security, the choice of the implementation must be done by comparing different solutions with an efficient design-space exploration process. The space of available solutions has mutated in recent years to include heterogeneous components where multiple copies of programmable components of different kinds (FPGAs, Digital Signal Processors, microprocessors, ad hoc processing elements) are paired with memories, application specific components and peripherals, yielding implementation platforms called Multi-Processor Systems on Chip (MPSOC). Hence, to design Cyber-Physical Systems effectively, we need a design methodology, a design environment, a set of tools and algorithms that deal with heterogeneity both at the functional level and at the implementation level where different mathematical formalisms and multiple tools possibly with different semantics must be coordinated and effective design space exploration must be carried out.

Obtaining a full view of the design and its environment is a major challenge since it involves the interaction and coexistence of multiple heterogeneous formalisms. One solution is to work at a low abstraction level, close to the final implementation, where an expressive model can be used to describe the complex interactions in detail. By giving up on abstraction, however, this approach sacrifices the ability to apply domains-pecific techniques. Another solution is hybrid modeling, which has been used especially for domains where safety and formal analysis are of primary importance [Carloni et al. 2006; Passerone et al. 2007]. Hybrid techniques have evolved into metamodeling frameworks, which are quickly becoming the backbone of modern system level design methodologies [Passerone et al. 2009; Sangiovanni-Vincentelli et al. 2009; Balarin et al. 2005; Basu et al. 2006; Karsai et al. 2003; Brooks et al. 2005; Alexander 2006]. In a metamodeling framework, the distinction between the individual heterogeneous models is retained throughout the design flow. Designers can then focus on the distinguishing aspects of the subsystems and apply techniques that rely on specific assumptions about the models in use. A typical metamodeling approach constructs a model of computation (MoC) that is general and highly expressive. The MoC is then tailored to the specific needs of each particular domain of interest by specializing its operators or by ignoring those portions of the formalism that are not needed at the level of abstraction under consideration. These specialized MoCs are seen simply as different customizations of a common underlying semantic structure. As the design is refined, the different domains are brought together through relations that arise from the way each of the specialized MoCs is mapped onto the common structure, thus relieving the designer from explicitly formulating the interactions.

On top of metamodeling techniques, the Platform-Based Design (PBD) [Pinto et al. 2006; Sangiovanni-Vincentelli 2002; Carloni et al. 2002] methodology was introduced to address integration and design reuse challenges with a refinement/abstraction-based approach. PBD is based on the identification and selection of different layers which represent different levels of the design abstraction called *platforms*. Each platform is a well

separated library of computational and communication components. The platforms at higher level of abstraction hide unnecessary details with respect to platforms at lower levels of abstraction. The design process in PBD is a sequence of refinement steps, from specification to implementation, where the functional representation at each level is mapped onto the architectural representation at the lower level. Performance metrics are evaluated and compared to the design requirements at each step. The abstraction levels must be carefully chosen to strike a balance between the desired level of analysis and implementation efficiency. Different design domains and application areas require distinct libraries of components and a different range of abstraction levels. Therefore, each domain requires the development of specific architectural, performance and functional models.

In this article, we describe the Metropolis II (METROII) [Davare et al. 2007] framework, which has been designed to support the PBD methodology that was developed with the intent of addressing the challenges posed by Cyber-Physical systems and among others, MPSoC implementation platforms. METROII builds on the experience of the METROPOLIS [Balarin et al. 2002] design tool-flow, and focuses on model integration at both functional and non-functional levels. METROII introduces new techniques for defining component interaction, representing non-functional quantities, and coordinating different models for performance evaluation. The article is organized as follows. Section 2 covers related system-level design frameworks, especially those that focus on metamodeling and heterogeneous MoCs and architectures. Section 3 introduces METROII, while Sections 4 and 5 describe the building blocks of the environment and semantics respectively. The basics of modeling with METROII is introduced in Section 6, while the results of modeling and analyzing a complex MPSoC system are presented in Section 7. Finally, to demonstrate the capability of the environment of integrating different tools with different semantics, we present the integration with another popular metamodeling framework developed at Verimag, BIP, in Section 8 before concluding in Section 9.

2. RELATED WORK

We have previously presented a broad survey of system level design tools and methodologies in the context of PBD [Densmore et al. 2006b]. There, several general aspects are identified, which are important for comparing tools and flows. In this section, we will focus primarily on metamodeling frameworks that are most related to METROII. In particular, we are interested in their modeling capabilities and especially in their handling of different interaction paradigms to support heterogeneity. We will also compare the tools in terms of their ability to effectively decouple functionality and architecture, and on the mechanisms available for their mapping.

A host of industrial tools have their roots in *Model Driven Architecture* (MDA) developed by OMG [MDA 2003]. At its basis is the separation between system behavior vs. usage of platform capabilities. Development starts with a *computation independent model* (CIM), which captures detailed requirements but no functionality. This model is refined into a *platform independent model* (PIM), used to specify the functionality of the system without committing to any particular platform. A PIM is transformed into a *platform specific model* (PSM) through a mapping that consists of model transformations, i.e., rules or algorithms that take objects in the PIM model language and generate (one or more) objects in the PSM model language. Annotations and attributes can be used to enrich the PSM model with non-functional properties [Cancila et al. 2010]. There are many similarities betweeen MDA and our methodology, starting with the shared goals of achieving model portability, interoperability and reusability. The main difference relates to our focus on architecture exploration. We therefore employ a mapping that is more generic, and intended to provide performance metrics rather

than a detailed implementation. Our notion of mapping makes it easier to adapt to different platforms which facilitates the exploration of a large design space.

An approach similar to MDA is *Model Integrated Computing* (MIC) [Karsai et al. 2003]. MIC also uses models for the design representation and generators to synthesize and integrate the system. Unlike MDA, which uses UML for all levels, MIC uses domain-specific modeling languages. Thus, different modeling languages are used to express the functionality, the architecture and their relation (the mapping). The MIC methodology is supported by a set of tools that can create and manage such languages. For instance, the *Generic Modeling Environment* (GME) has been designed to facilitate the construction and the manipulation of a domain-specific modeling language, by providing a way to specify an abstract as well as a concrete syntax (textual or graphical), including well-formedness constraints and static semantics. The language design activity is again based on UML and on OCL constraints [OCL 2006], which are used as *meta-languages*. The resulting language need not be related to UML at all. *MILAN* [Bakshi et al. 2001; Ledeczi et al. 2003] is a verification tool which supports simulator integration using model interpreters, and integrates the design space exploration tool *DESERT* [Neema et al. 2003]. DESERT allows the designer to express platform flexibility by specifying structural constraints in OCL. Symbolic pruning of the design space based on these constraints can greatly reduce the number of points to be evaluated with lower level simulators. Unlike GME, our work is not concerned with the design of the modeling language. Instead, we construct adaptors to coordinate components from different MoCs. Accordingly, we are mainly focused on the combined semantics of different models, instead of the relationships between their syntactic elements. Our objective is to determine semantic relationships using refinements into a common semantic domain, and then abstract the results into a mixed domain that supports the development of adaptors. In this sense, our approach is closer to that used by MILAN, where model interpreters can be seen as adaptors. However, MILAN is focused on simulator integration, not a formalism for MoC integration. In addition, our exploration paradigm differs substantially from DESERT's. In particular, we employ functional and scheduling constraints, instead of structural constraints, and thus are able to relate functionality and architecture without resorting to low level simulators. However, the combined use of structural and scheduling constraints for fast generation and exploration of architectures is a promising avenue of future research.

Ptolemy II is a design environment for heterogeneous systems that consists of several executable domains of computation that can be mixed in a hierarchy controlled by a global scheduler [Brooks et al. 2005]. Each MoC is described operationally in terms of a common executable interface. For each model, a "director" determines the activation order of the components (or actors). Similarly, a "receiver" implements communication in terms of a common interface. A MoC, or *domain*, in Ptolemy II is a pair composed of a director and a receiver. Heterogeneity in Ptolemy II is strictly hierarchical. This implies that each node of the hierarchy contains exactly one domain, and that each component interacts with the rest of the system using the specific communication mechanism of the hierarchy node it belongs to. Domains only interact at the boundary between two different levels of the hierarchy. *SystemC-H* [Patel et al. 2007] is a heterogeneous extension to *SystemC* [Grötker et al. 2002]. While SystemC is based on a discrete event simulation kernel, SystemC-H extends it to provide additional MoCs such as dataflow and hierarchical FSMs, using similar techniques as Ptolemy II. The authors demonstrate an increase in simulation efficiency over SystemC with MoC-specific analysis such as static scheduling for dataflow. The hierarchical approach to heterogeneity of Ptolemy II and SystemC-H is nicely structured and is excellent for experimentation. The structure, however, also imposes limitations on the heterogeneity that can be achieved. More importantly, the relationship between different models is implicit in the

way the execution protocol schedules the activation of the directors and the transfer of information through the receivers. This makes it hard to predict the outcome of a hierarchical heterogeneous composition or to study its properties. In addition, the execution protocol is hard-wired in the framework, and therefore it cannot be changed without altering the core of the tools. As a result, the relationship between different models (i.e., abstraction and refinement) is fixed, unless boundary components are used to explicitly translate between different domains. For example, this technique is used in Ptolemy II to translate from the discrete to the continuous domain, and vice versa, through special transducers. This, however, appears to relax the requirement for strict hierarchical heterogeneity. Our approach to heterogeneity is instead based on establishing clear abstraction and refinement relationships between models, typically through the use of a common semantic domain. Adaptors which are consistent with the chosen abstractions and refinements can be constructed to coordinate the execution of different models. The approach is therefore not hierarchical, as in Ptolemy II, where the interaction between models is prescribed by the framework, but favors instead a more flexible horizontal adaptation of models, as discussed in Section 4.6.

ForSyDe [Sander and Jantsch 2004] initially specifies the system as a deterministic network of fully synchronous processes that communicate over sequences of events. Haskell has been chosen as the concrete language for expressing the model, since it natively supports higher order constructs. This specification, which lacks detailed timing, is then refined into an implementation by applying a series of network *transformations*, that may or may not preserve the semantics. These transformations can, for example, partition the system into sub-domains that run at different speeds (the model is therefore no longer fully synchronous), interfaced through up- and down-converters. When the desired structure has been obtained, processes can be converted into hardware or software. The basic ForSyDe model was then extended to cover a larger array of MoCs [Jantsch 2003], and has been implemented in Standard ML in the *SML-Sys* project [Mathaikutty et al. 2004b], as well as in C++ [Mathaikutty et al. 2006]. There, the initial assumption of a fully synchronous system is dropped in favor of an untimed model similar to Kahn process networks [Kahn 1974]. In addition, synchronous, clocked and timed models can be used for refinement. However, SML-Sys appears to be more focused on heterogeneous design, rather than on transformational refinement. For this reason, SML-Sys relies on ForSyDe for network transformations, while more complex interfaces have been introduced to bridge the gap between different sub-domains. More recently, the same group has developed a front-end to both SML-Sys and ForSyDe, called *EWD* [Mathaikutty et al. 2004a], which captures their common structure into a GME-based metamodel, and provides some code generation facilities. GME uses the static semantics to catch certain classes of errors early in the design process. Unlike ForSyDe, METROII supports non-deterministic systems, a choice that renders analysis more complex, but greatly simplifies the description of the system's environment. As the design is refined, deterministic components are replaced for non-deterministic ones in the system. The transformation-based refinement in ForSyDe has clear advantages in terms of the ability to prove correctness and maintain consistency with the original specification. However, the distinction between functionality and architecture is lost, and a change of mapping may require substantial restructuring of the system. Our approach to mapping, instead, makes this task simpler, since only the mapping function must be changed. Heterogeneity is addressed in SML-Sys, using domain interfaces, or adaptors, that add or remove events from the event sequences. However, SML-Sys does not advocate building these adaptors based on an underlying common semantic domain substrate, as we do in METROII.

In the heterogeneous system-level design language *Rosetta* [Alexander 2006; Kong and Alexander 2003], a MoC is described declaratively as a set of assertions in a

higher order logic. Different MoCs can be obtained by extending a definition in a way similar to the sub-classing relation of a type system. MoCs obtained in this way are automatically related by an abstraction/refinement relationship. Unrelated MoCs can still be compared by constructing functions that (sometimes partially) express the consequences of the properties and quantities of one domain onto another. This process is particularly useful for expressing and keeping track of constraints during the refinement of the design. In contrast to Rosetta, the relationship between the function and the architecture in METROII is not described explicitly as a function, but rather as a mapping and annotation process at the event level. Annotations at this level are simpler to express, and can be used directly for simulation. Conversely, domain interactions are harder to manipulate, since they are defined at the level of the domain, but may in principle be used to derive stronger results via formal reasoning. Tools that take full advantage of the Rosetta representation are, however, still in development.

The separation between computation and coordination is central to the Behavior-Interaction-Priority (BIP) framework [Basu et al. 2006]. In *BIP*, a system specification is divided into three layers. The lowest layer describes a set of independent components, while the second layer controls their activation and interaction via *connectors*. The top layer overlays a set of priorities to govern component interaction which reduce non-determinism. One of the strengths of the BIP framework is the ability to check certain properties, such as deadlock-freedom, compositionally. However, this may require a complex coordination scheme between a large set of connectors. In contrast, we utilize a centralized scheduler to accomplish similar objectives. Another difference is our utilization of an imperative description for the scheduler instead of BIP's declarative specification of connectors. More information is provided in Section 8, where integration between BIP and METROII is explored.

3. FROM METROPOLIS TO METROII

The METROPOLIS [Balarin et al. 2003] design framework was the precursor of METROII and it was developed to embody the principles of PBD. METROPOLIS was the first system to leverage the concept of a semantic metamodel (abstract semantics) to manage the integration of heterogeneous components, to allow declarative and operational design entry, and to model architecture and functionality in a unified way. The concepts of separation of concerns, mapping of functionality to architectural components as a way of refining a design from specification to implementation, and communication as a first class citizen were all instrumental to building the framework. These concepts originated from work over years of research and development. The roots of METROPOLIS can be found in Polis [Balarin et al. 1997] that was the first framework to be based on the separation of functionality and architecture.

METROPOLIS has its own specification language, the METROPOLIS Meta-Model (MMM), that is used to capture functionality, architecture, mapping, and a variety of design activities. It is based on formal semantics and remains general enough to support existing models of computation [Lee and Sangiovanni-Vincentelli 1998] and accommodate new ones. MMM also includes a logic language to capture nonfunctional and declarative constraints. Because MMM has a precise semantics, METROPOLIS is able to perform synthesis and formal analysis in addition to simulation.

METROPOLIS offers syntactic and semantic mechanisms to compactly store and communicate all relevant design information. Designers can plug in algorithms and tools for all possible design activities that operate on the design information. The capability of introducing external algorithms and tools is important, because these needs vary significantly for different application domains. To support this capability, METROPOLIS provides a parser that reads MMM designs and a standard API that lets developers browse, analyze, modify, and add additional information within those designs. For each

tool integrated into METROPOLIS, a back-end uses the API to generate required input by the tool from the relevant portion of the design.

Based on experience gained from the development and usage of METROPOLIS, we identified three main features to enhance. These features form the basis of the second-generation METROII framework.

(1) *Import of pre-designed heterogeneous IPs.* 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 not only for syntactic translation, but also to ensure the semantics are preserved. Instead, heterogeneity has to be supported by the new environment to import IPs with different semantics and languages.
(2) *Cost/behavior separation.* In a system-level framework that supports multiple abstraction levels, different implementations will have the same behavioral representation at higher abstraction levels. For instance, different processors will be abstracted into the same programmable component. What distinguishes them are the performance vs. cost trade-offs, which may not be optimized at the same time. It should be possible to introduce various performance metrics during the design process from specification to implementation. Separating cost and behavior is important to support such design activities.
(3) *Structured exploration of the design space.* 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, while the second part enables fast exploration to achieve optimal design.

In this article we show how these activities, which are basic tenets of the PBD methodology, are implemented in the METROII framework.

4. METROII BUILDING BLOCKS

To realize the features we envision for METROII, various types of objects are defined as building blocks for modeling. *Events* are the fundamental elements that constitute the semantics of the system and provide the foundation for the definitions of other objects. *Components* are the primary entities used for specification, and *ports* define the communication between components. A set of specialized objects operating on events are defined to capture different aspects of the system and govern its execution, including *annotators*, *schedulers*, *constraint solvers* and *adaptors*.

To better explain these objects, we utilize a simple producer-consumer example, as shown in Figure 1. The figure shows both the functional model at the top and the architectural model at the bottom. The functional model consists of a producer component communicating over a blocking FIFO to a consumer component. The architectural model includes *write* and *read* tasks that implement the producer and consumer functionality, executing on a processing element, and communicating through a shared memory. Additional architectural components regulate the platform execution. Various snippets of code are shown in Figure 2. In the rest of this section we will provide details on the building blocks of METROII and refer back to this example and figures for concrete explanations.

4.1. Events

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 tags (T) and values (V). Tags correspond to physical quantities in the design, such as time or power. Values come from the variables that are in the scope of

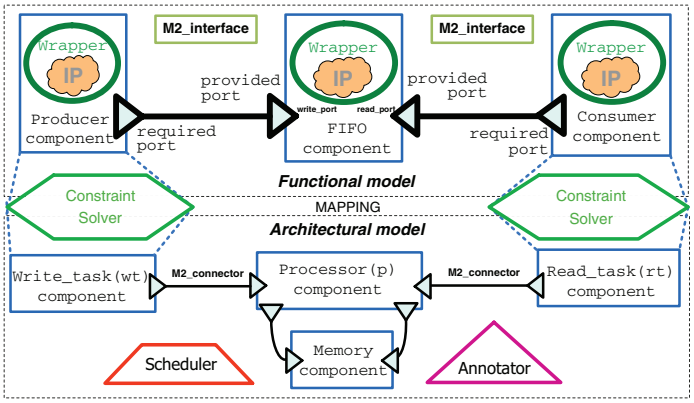

Fig. 1. Producer and consumer communicating through shared FIFO example. The top shows the functional model, while the bottom shows the architectural model. In the middle the mapping process is modeled using constraint solvers.

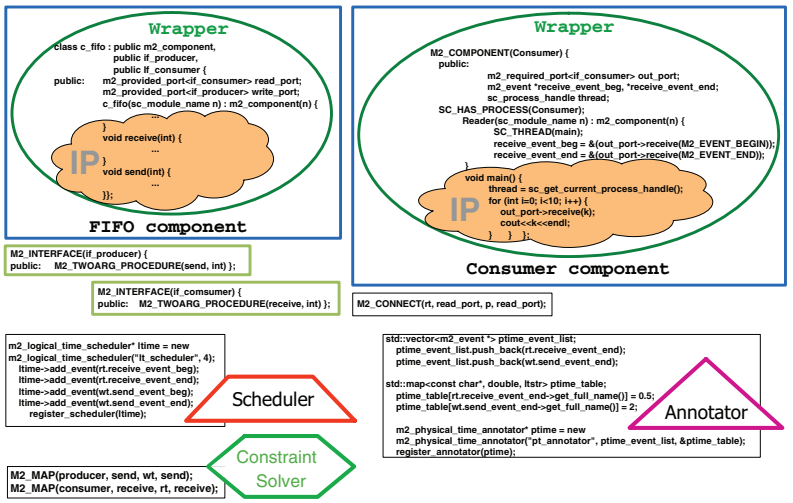

Fig. 2. Code snippets of METROII elements: communication, components, interfaces, scheduler, constraint solver, and annotator.

an event. An event also has state, which will be explained more in Section 5. The tags, values and states of events may be modified by other objects, as shown later in this section.

4.2. Processes and Components

A *component* is an object that encapsulates imperative code, used for either functional or architectural modeling, which may contain zero or more *processes*. Processes contain their own thread of control and execute concurrently with other processes in the system. Process execution is represented by a sequence of events.

There are two types of components: *atomic components* and *composite components*. An atomic component is made of two parts: the *internal behavior* and its *interface*. The internal behavior of an atomic component is specified in some language and defines the functionality or the service that the component offers to the system. However, a component is seen by the system as a black box with no visibility of the internal behavior.

This is necessary to guarantee language independence. Instead, components communicate with other components through their interface, which is composed of zero or more *ports*. Within a component, ports must be able to interact with the internal behavior by means of *wrapper* code, which translates between the two levels of specification (internal behavior and interface). The wrapper is the only language dependent part of a METROII specification, and can usually be instantiated by following simple templates. Conversely, the semantics of component interaction in METROII is defined exclusively on ports, and on the events that occur at the ports. This way, the same abstract semantics definitions are reused across different specification styles and can be used to achieve integration of heterogeneous models. More details on ports will be introduced in the next section.

Composite components are used to hierarchically structure the system. A composite component contains one or more internal components connected through ports, and in turn exposes an interface of ports, so that it can be instantiated into other composite components that sit higher in the hierarchy.

In the producer-consumer example shown in Figure 1 there are seven components, shown as rectangles. They can be categorized into functional or architectural components. On the functional side we have a producer, a consumer and a FIFO components. The producer sends data that are then received by the consumer. The communication takes place through a FIFO which provides blocking read and blocking write services. On the architectural side we have a writer and a reader task components, which are synchronized to the functional producer and consumer components through *mapping constraints*. More details of mapping constraints will be explained in Section 5.4. The processor and memory components capture the structure and behavior of the architectural platform.

The code snippets of the consumer and the FIFO component are shown in Figure 2. The basic behavior of the components is initially specified in SystemC, then encapsulated by wrapper code (shown enclosed in a circle inside components) to provide METROII interfaces. As a simple example, the *main* function inside the reader component reads the incoming data using a typical SystemC procedure $out\_port \rightarrow receive()$, where *out_port* is viewed by the internal code as a SystemC port. The wrapper then wraps the SystemC procedure as a METROII method by implementing the *out_port* as a METROII port and associating a pair of events with the begin and the end of the $receive()$ procedure respectively. The states, tags and values of these events coordinate the execution of the procedure with other actions in the system and pass the information between the internal *main* function and the outside world.

4.3. Ports

METROII *ports* are used for communication between components. Each port is associated with a set of *methods* defined by the port interface. A method, which corresponds to the call to a function, is abstracted at the component interface by being associated with a pair of begin and end events. For instance, a METROII method corresponding to the $receive()$ function is shown above in the producer-consumer example.

Activities at the ports take place through the operations on the method begin/end events. Specifically, the execution of the methods/functions are based on the event states (to be described in Section 5), while the quantity information of the methods, such as the time taken to execute them, are reflected in the annotation of the end event tags. A sequence of such events describes the activity at the port (see the semantics in Section 5 below). Methods can refer to variables which are used in the computation within a component.

There are two types of ports: *required* ports and *provided* ports. A required port of a component specifies a set of methods that the component requires from other

components to support its execution, while a provided port specifies methods that the component implements and provides to other components for them to utilize. One-to-one port connections are allowed between a required port and a provided port with the same interface, i.e., the same set of specified methods. Figure 1 shows how producer and consumer communicate with the FIFO by means of two pairs of provided and required ports, represented as triangles at the border of the components.

4.4. Annotators and Schedulers

In METROPOLIS, both the performance annotation and the scheduling of events are carried out by a type of special component called *quantity manager*. In METROII, to have a more clear separation of design concerns, performance and scheduling are handled separately by *annotators* and *schedulers*, through operations on events.

As explained above, end events are associated with tags to represent certain quantities of interest (e.g., physical time, power). Each tag is determined based on the parameters supplied to the annotator, the state of the event, and the values of the event. Parameters are set by designers according to the characterization of the architecture platforms. Only static parameters are permitted for annotators, which may not have their own state. An annotator base class is provided in METROII, and subclasses can be derived for various quantities or quantities in different systems. As an example, a physical time annotator is provided in the METROII library. The code snippet for instantiating this annotator (triangle) is shown in Figure 2. The events *rt.read_event_end* and *wt.write_event_end* are associated with the methods in the architectural component *read_task* and *write_task*, respectively. They are added to a list (*time_event_list*) as events to be annotated. In addition, a table (*ptime_table*) stores the physical time units annotated for the events. In the design process, this table is typically constructed from platform specification or profiling. The event list and table are used to instantiate the annotator, which will update the tags of the events during the appropriate execution phase.

Events can be *enabled* or *disabled* according to whether they are allowed to occur at a certain point in the execution. Schedulers (the trapezium in Figures 1 and 2) coordinate the execution of the components by enabling/disabling the events, based on the local state of the scheduler, the current states of the events, as well as their values and tags. Enabling/disabling events corresponds to changes of events states, which will be explained more in Section 5. A base class scheduler is provided in METROII for designers to derive various schedulers. A logical time scheduler, for instance, schedules the events based on the mapping constrains, the physical time tags and the scheduling policy. This is a basic scheduler of METROII that determines the sequence of executed processes. A round-robin and first-come first-served schedulers are provided as library schedulers to regulate the access to shared resources. The logical time scheduler of the producer-consumer example is shown in Figure 2. It orders events *rt.read_event_begin*, *rt.read_event_end*, *wt.write_event_begin* and *wt.write_event_end*, based on the physical times annotated to them. A round-robin scheduler orders the events accessing the memory component (not shown in the figure to avoid clutter).

4.5. Constraint Solvers

Constraints are a form of declarative specification, as opposed to the imperative specification contained in components. Specifically, they are expressions built on the states, tags and values of the events that are associated with the ports in the system.

Constraint solvers are objects that enforce the validity of constraints during runtime by enabling/disabling events. Fundamentally they serve a similar purpose as the schedulers. The main difference is that schedulers are typically used for resource

scheduling and described in imperative code, while constraints solvers enforce the synchronization between events through declarative expressions.

Designers can derive various constraint solvers from the base class solver provided by the METROII infrastructure. In addition, a mapping constraint solver is provided as a library solver. Mapping constraints synchronize the execution of functional components with corresponding architectural components through events. Two events that are specified in a mapping constraint must occur simultaneously during simulation. Figure 1 shows two constraint solvers at the interface between the functional and the architectural model to establish mapping constraints, while Figure 2 shows an example of usage of the mapping constraint solver. The semantics of mapping will be further explained in Section 5.4.

4.6. Adaptors

Heterogeneity is one principal feature of CPS, so that tool must be able to precisely mix models of computation. In our experience, there is a strong need to interconnect heterogeneous models at the same level. For instance, the designer 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).

To bridge the different semantics of heterogeneous components, we define *adaptors* as first class objects in our specification. Adaptors are connected with other components through ports. When two heterogeneous components communicate, the adaptor between them translates the events at one end of the connection to the events at the other end. Denotationally, an adaptor is a relation that maps sets of events from one component to sets of events in another component. The difference between an adaptor and a component is that adaptors do not own specific begin/end events at their ports, but instead rely on those generated by the connected components. This way, each port transparently inherits the annotations and the scheduling of the connected components, making the adaptor a passive translator. Adaptors can also be used to build *transactors*, components that can translate communication at different granularity levels. Techniques for the automatic synthesis of adaptors and transactors have been reported in our previous work [Passerone et al. 2002; Balarin and Passerone 2007].

In the producer-consumer example, if the producer contained a dataflow processing element and obtained its data from a continuous time component, an adaptor would be required to bridge the two components. In continuous time, an event could be associated with a pair of a function and a right open interval of the reals (i.e., a time interval), $(f(t), [t_-, t_+))$, through its tags and values. One possible adaptor between continuous time and data flow would relate the continuous time event with a set of data flow events that sample the function at specific times:

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

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 delivers them to the dataflow model.

Control and data-dominated subsystems are common in CPS, and can be connected by adaptors that translate between a finite state machine (FSM) model to a process network (KPN) model. The FSM is a synchronous model that generates and consumes data tokens at regular intervals, while the KPN model is asynchronous and is triggered by the presence of the tokens. The FSM models introduces an undefined value, denoted by the symbol $\perp$, when no information needs to be exchanged. The KPN models does not require the $\perp$ value, since it can intrinsically distinguish when data is available.

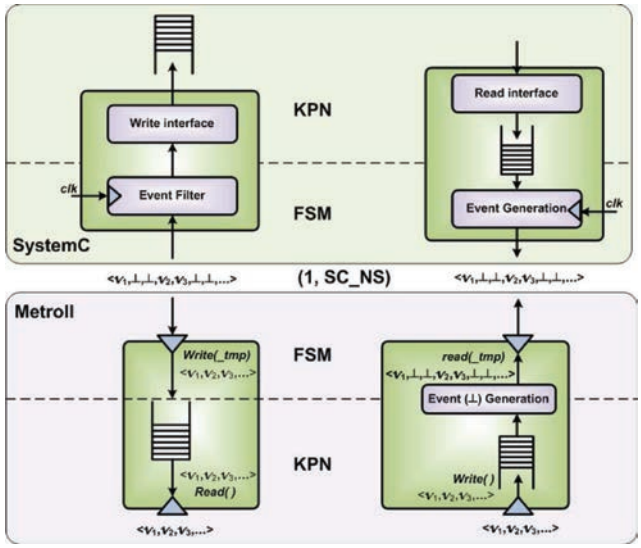

Fig. 3. A bidirectional adaptor between an FSM and a KPN model of computation in both SystemC and METROII.

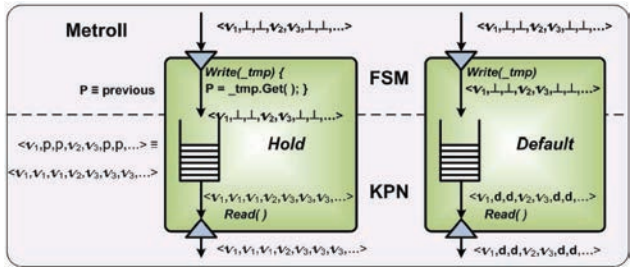

Fig. 4. Different FSM to KPN adaptors in METROII: one holds the last value, the second introduces default values.

Interfacing thus consists in adding and removing $\perp$ events from the stream of data, according to some desired logic of operation.

Adaptors of this kind can be built in several frameworks. Figure 3 shows an example of a bidirectional adaptor in both SystemC and METROII. The SystemC event filter is responsible for removing the $\perp$ values from the FSM stream, when interfacing towards KPN. Likewise, the event generator of the KPN to FSM adaptor will introduce such events. These components of the interface are modeled as threads or methods that must be synchronized with the state machine clock in order to function properly.

Although this METROII implementation is also based on SystemC, it takes advantage of the possibility of calling the adaptor methods directly from the components. Thus, all the adaptor methods are *provided* methods, and are driven by the respective domains. This way, the clock synchronization is no longer required, making the adaptor passive and self contained.

Unlike other frameworks which operate more rigidly at the interface between models (such as the hierarchical heterogeneity approach of Ptolemy II), the METROII designer can easily switch between different adaptor implementations. Figure 4 shows further examples of different implementations of the FSM to KPN adaptor. In the first case, on the left, the $\perp$ values are not completely discarded, but the last available valid value is

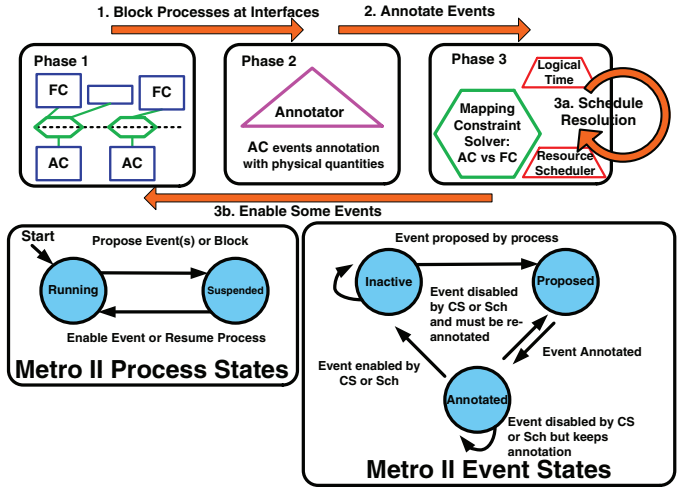

Fig. 5. Three Phase Execution in METROII (top) and process and event state transition diagrams (bottom).

repeated. We call this the *Hold* adaptor. A different implementation, useful when one needs to distinguish effective data duplication, is given by the *Default* adaptor, which introduces a default value d in place of the undefined value. This implementation tends to preserve the most information when moving from one domain to the other.

5. EXECUTION SEMANTICS

The two key concepts underlying the METROII semantics are the basic notions of *event* and *process* that were introduced in Section 4. METROII has a three-phase execution semantics, during which processes execute and then block to generate events, events are annotated with quantities, and finally scheduled to resume the process execution. The top of Figure 5 shows these three phases and the METROII elements involved in each phase (here FC is a functional component and AC an architectural component). During simulation, processes and events transition through a number of states which determine, together with annotators, schedulers and constraints, the execution order. The bottom of Figure 5 shows these states and their transitions. In the rest of this section we provide details on each of the phases, and show how to use the infrastructure for architecture exploration.

5.1. Process and Event States

Events can be in one of three states: *inactive*, *proposed* or *annotated*. During simulation, all events begin in the inactive state, where they can remain indefinitely. An event becomes proposed when a method call on a required port generates it. The begin event will be proposed before the end event associated with the same method call. The event can then be annotated with quantities by the annotators and enter the annotated state. From there, it will transition back to the inactive state if enabled by a constraint solver (CS) or a scheduler (Sch) (via a variety of scheduling decisions).

Processes can be in two states: *running* or *suspended*. Processes start in the running state. They execute concurrently until an event is *proposed* on a required port of the component containing the process, or until they are blocked on a provided port. At this point, they transition to the suspended state. Once the event is enabled or the internal blocking is resolved, the process will return to the running state.

5.2. Three-Phase Execution

The execution semantics is partitioned in three phases, following the state changes of events. Events are i) proposed by processes in the first phase, ii) associated with tags in the second phase, and iii) enabled or disabled in the third phase. This three-phase semantics is used to clearly separate the processes of functional simulation, annotation and scheduling. Figure 5 summarizes such execution semantics.

First phase: Base Model Execution. The base model consists of concurrently executing processes, which may be suspended only after proposing events or by waiting (blocking) for other processes. A process may propose multiple events—this represents non-determinism in the system. After all processes in the base model are blocked, the design shifts to the second phase. The execution of processes between blocking points is beyond the control of the framework.

Second phase: Quantity Annotation. In the second phase, proposed events are annotated with various quantities by annotators. For instance, a proposed event may be annotated with a physical time quantity to represent the time cost of executing the corresponding method. One event may be annotated by multiple annotators that model different quantities. New events may not be proposed during this phase of execution.

Third phase: Scheduling or Constraint Solving. In this phase, a subset of the proposed events are enabled and permitted to execute, while the remainder remain suspended. The enabled events become inactive again while simultaneously allowing their associated processes to resume. Events are enabled/disabled by constraint solvers and schedulers, based on the resolution of declarative constraints or by imperative code. At most one event per process is permitted to execute. Once again, new events may not be proposed during this stage.

A collection of three completed phases is referred to as a *round*. After the constraint solving phase, the states of some processes are switched to running while the others remain suspended. The execution will then shift to the first phase and start a new round. Those processes that are in the running state will resume their execution.

5.3. Execution Semantics of Required and Provided Ports

We explained in Section 4 that required and provided ports with the same set of methods can be connected in METROII. We call the method associated with the required port "required method," and the one associated with the provided port "provided method." During simulation, first the begin event of the required method is proposed. Once it is enabled, the control transfers to the component at the other end of the connection that owns the provided port, and the begin event of the provided method is proposed. When this begin event is enabled, the content of the provided method is executed. After its execution is over, the provided method proposes the end event, waits until it is enabled, then transfers the control back to the required method, which has been waiting at the other end. Note that the provided method does not have a separate process to carry out the execution. Instead, it inherits the process from the caller component and the method is executed in that context.

5.4. Mapping Semantics

As a key step in the platform-based design process, mapping explores the design space when bridging the functionality and the architecture. In order to explore multiple implementations with minimal effort, the design framework needs to provide a fast and efficient way of mapping without significantly changing the functional or the architectural models. In METROPOLIS, this was achieved by event level synchronization. While providing a powerful way to link the models, this approach breaks the encapsulation of

the models by allowing constraints to be established between arbitrary pairs of events, and allowing any local variables in the scope of the events to be accessed. Furthermore, since there are no special declarative constructs for mapping, the process of finding events and setting up constraints is not easy for designers to manipulate and debug. In METROII, we restrict mapping to occur at the service level, i.e., the only accessible events for synchronization are the begin/end events of interface methods in the functional and architectural models. In addition, the only accessible values are parameters and return values of the interface methods. This coarser granularity and more restrictive mapping approach retains IP encapsulation, and makes mapping easier and more robust.

In the METROII framework, event synchronization is implemented with rendezvous constraints. These rendezvous constraints are handled in the same way as any other event constraint during phase 3 of the execution semantics. Because mapping uses the same infrastructure as the rest of the system, the semantics of simulation is not burdened with additional dedicated steps for mapping.

During execution, the functional and architectural methods independently propose their begin events. Because of the rendezvous constraint enforced on the two begin events, they may only be enabled when both are proposed. Then, after the begin events are enabled, the functional and architectural methods are executed. Upon completion of the methods, two end events will be proposed, one from the functional method and one from the architectural method. Again, these two end events may be enabled only once both of them are proposed. The enabling of the two end events marks the end of execution of a mapped functional and architectural method pair. Parameters and return values can be passed between functional and architectural methods through the value field of the synchronized begin/end events.

5.5. Execution Semantics of the Producer-Consumer Example

In the producer-consumer example presented in Section 4, the *receive_event_beg* and *send_event_beg* events of the functional model and the *rt.receive_event_beg* and *wt.send_event_beg* events of the architectural tasks will be proposed in the first phase of the first round of the execution at the same time. The begin and end events of the functional components mapped to the architectural tasks of this example will be proposed synchronously and thus mapping constrains will be satisfied. Thus, when describing the execution steps of the producer-consumer example we will refer only to the events of the architectural tasks. The *rt.receive_event_beg* and *wt.send_event_beg* events will be annotated with "zero" time units in the second phase. The *wt.send_event_beg* event, however, will be disabled in the third phase of the execution semantics, since it is listed later than the *rt.receive_event_beg* event in the logical time scheduler. The process of the consumer component will then be suspended. In the second round both the *wt.send_event_beg* event and the *rt.receive_event_end* will be proposed. The *rt.receive_event_end* event will be annotated with 0.5 time units. The *wt.send_event_beg* event will be disabled once more, since both tasks are executed on the same processing element and the round-robin scheduling policy used in the producer-consumer example does not allow method execution preemption. In the third execution round the global time will be equal to 0.5 time units. The value received by the consumer component will be equal to "zero", since nothing was sent yet by the producer component. The execution of the *send* method will be performed in the next execution rounds following the same execution scheme.

6. MODELING WITH METROII

Following the idea of Platform-Based Design, METROII supports the separate development of functional and architectural models of a system. A *functional* model is a structural representation of the system functionality described as a set of processes

that concurrently take actions while communicating with each other. An *architectural* model is an interconnection of elements, representing CPUs, memories, or physical level communication channels, which support Multi Processor System-on-Chip designs (MPSOC). Elements of the architectural model provide services which are annotated with *cost metrics* when used by a function to perform an operation. The architectural services complement the functional modeling effort. In order to offer a variety of mapping solutions, architectural services are modular and are able to support various configurations (association with different processing units) and parameterizations (service *costs* with respect to processing units). The ultimate goal is to create a model which has a firm grounding in the real world, and therefore provides meaningful simulation data when associated to a functional design during design space exploration.

The main building blocks of both functional models and architectural services are the METROII *components*, described in the previous sections, which may include both newly developed and pre-designed IPs encapsulated by wrappers. The appropriate events and interfaces are exposed by these wrappers for further components communication (functional to functional component connection) or mapping (functional to architectural components connection).

6.1. General Structure

To simplify the mapping process, models of architectural services are best described using certain specific patterns. In particular, as a service, the model must be able to support the functional model requirements and provide the associated costs when the service is requested. To do this, an architectural element typically contains a component that exposes ports at its interface to be associated, through appropriate mapping constraints, to the functional model. We call this component the *architectural task*, which is equipped with its own thread of execution. In turn, the thread generates events associated with the ports, which are synchronized with the corresponding events at the functional level. In order to compute costs, the events associated to the architectural task are also registered with the necessary annotators and schedulers. These are then able to accumulate the overall cost of the simulation.

Besides the architectural tasks, whose purpose is to serve as mapping targets for the functional component, the architectural model may also include *operating systems* (OS) and *processing elements*. An operating system is an explicit, imperative mechanism for scheduling and assigning tasks to processing elements. Processing elements, in turn, are the workhorse of the architectural model and are used to return the core cost of a service.

Figure 6 illustrates these three classes of architectural elements, and details, on the left, the various steps of architectural evaluation, as will be explained in the next sections. Selected components are exploded to see internal details. Tasks, shown at the top, compete for execution by proposing events. The OS, in the middle, gathers and orders the requests, and mediates with the physical platform components by assigning the tasks to the processing elements. The platform is shown at the bottom, as a collection of processor and memory resources (other platform components may also be present, and are omitted here for simplicity). The processing elements provide execution metrics through various styles of performance modeling (runtime and profiled, as discussed below). Processors communicate in this case through a shared memory. Below, we discuss each of the architectural elements in more details.

6.2. Architectural Tasks

Tasks in the architecture model are lightweight active METROII components. The thread they contain is created such that it is in a constant state of event proposal. Mapping creates a rendezvous constraint between the event generated by the task

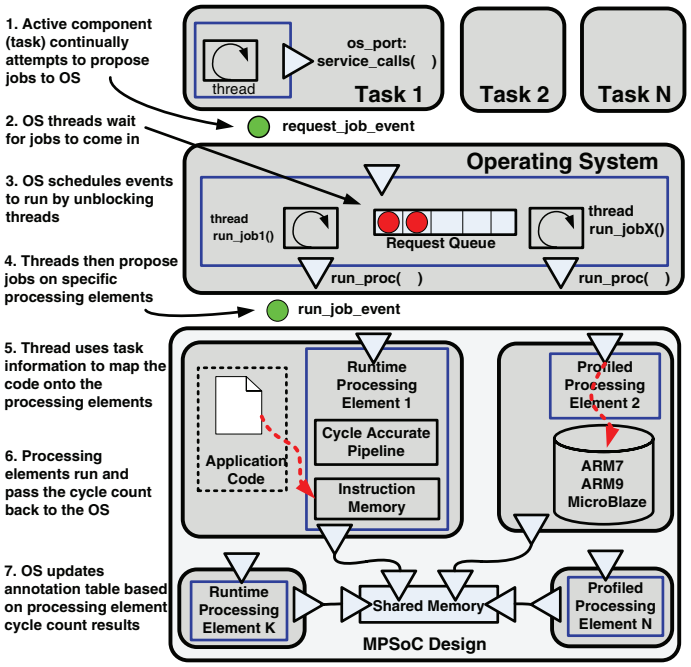

Fig. 6. Architectural Model: tasks (top), operating system (middle), and MPSoC architecture (bottom). The steps for architectural evaluation are shown schematically on the left.

thread (*request_job()* in Figure 6) and the functional event. Therefore there is a one-to-one mapping between these tasks and components which one desires to map in the functional model. The continually running architecture task is then blocked until the mapped event in the functional model is proposed. Step 1 in Figure 6 illustrates the task role in the architecture model execution.

6.3. Operating System

Using a METROII component, it is possible to define an operating system (OS) to assign tasks to processing elements in a many-to-one relationship. In addition, it can also perform scheduling during phase 1 to determine which events will be proposed prior to phases 2 and 3. As was described in Section 4.4, there exist two predefined METROII schedulers (i.e., RR and logical time). However, the user is free to add his/her own scheduling policy to the system. An operating system is a component with N threads (where N is the number of processing elements it controls), that maintains a queue of requested jobs from tasks and that queries the processing elements to decide if a task can be execute or not. Scheduling controls how events are added to this queue. Access to the queue is coordinated such that there is a limited amount of outstanding requests to any processing element. Steps 2-5 in Figure 6 illustrate the OS role in the architecture model execution.

The OS is also used to access the annotation tables, which relate event costs to architectural services. The annotation tables are used by the annotator in phase 2 where event tags can be written. The OS updates the appropriate entry in the table after a request is completed and the true cost known. In addition to the processing cost, the OS may also add cost related to overhead (e.g., context switching). Tables are updated dynamically at runtime and do not require to be statically created with the netlist. The advantage of this configuration is that tasks are independent of this process

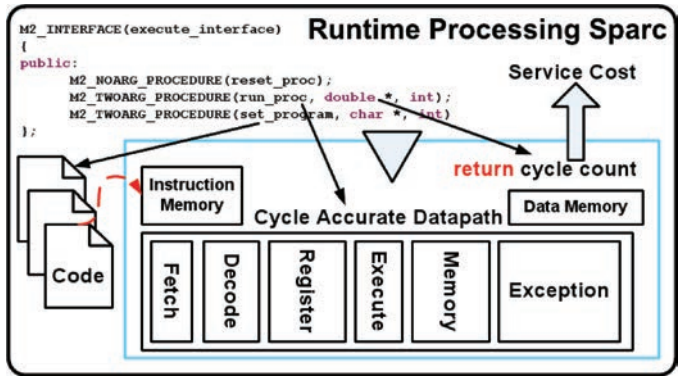

Fig. 7. Sparc Runtime Processing Element: instruction code, datapath microarchitecture, and cost computation.

and only need to indicate which service they require. This is essential to separate the computation behavior from its performance cost.

6.4. Processing Elements

The third element of the architecture platform are the actual processing elements themselves. Once the OS decides to run a request from a task, it calls the corresponding *run_proc()* function on one of its N required ports. There are two distinct types of processing elements, which compute the cost for its use in a different way. Steps 6-7 in Figure 6 illustrate the processing element role in the architecture execution.

6.4.1. Runtime Processing. In *runtime processing*, a component executes a routine at runtime to determine the cost and the annotation for a given event. An example of runtime processing is a cycle accurate model of a Leon 3 SPARC shown in Figure 7. It is a METROII composite component that offers three key functions used in three different steps of function cost evaluation. The first step in using this type of processing element is for the OS to provide information gathered from the task as to which operation is requested. This will be done via the *set_program()* function where the instruction memory of the processing element is loaded with pre-compiled code for the operation. The second step is then to execute that code at runtime with *run_proc()* which will return the cycle requirements for that code. The operating system will use that information to update the annotation table. Finally, the OS will call *reset_proc()* to prepare the processor for the next request.

While this style may result in slower simulation performance compared to the following approach, it only requires that the code for the function be available. It requires no other modeling work by the user and is as accurate as the level of detail in the microarchitectural model. svalues for vari

6.4.2. Profiled Processing. In *profiled processing* a set of precomputed values for various job requests are collected prior to simulation and stored for later look up. Again, the OS will indicate to the processing element which services are requested. In turn, the processing element will look up the costs for the given operations. These can be trivial table look-ups or more complex (but still static) calculations based on the current state of the processing element. Ways to characterize processing elements for this approach have been shown in previous work [Meyerowitz et al. 2008; Densmore et al. 2006a; Simalatsar et al. 2008]. One advantage of this approach is that the look-up is extremely fast compared to the runtime processing approach. The drawback is that the modeling of these elements is often more limited in its usage and requires that

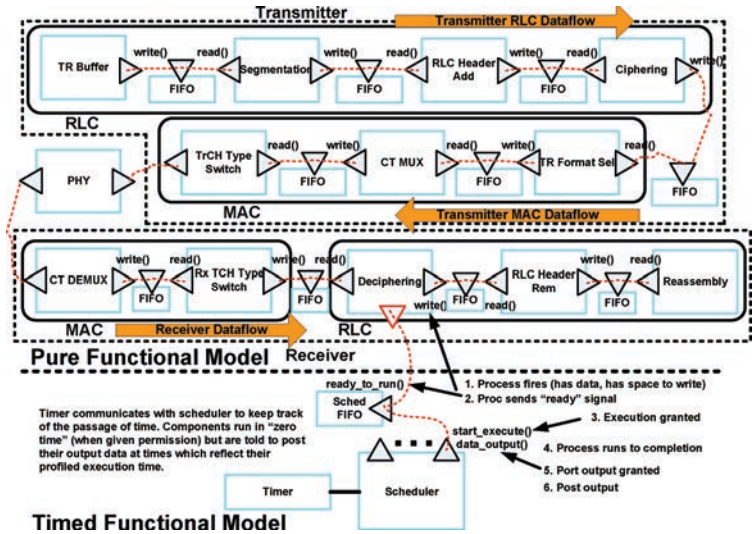

Fig. 8. UMTS METROII Functional Model.

characterization be carried out prior to simulation. This pre-characterization however only needs to be done once per computation routine. It does require a more complex set of transformations compared to simple compilation (runtime processing approach).

7. MODELING MPSOCS: THE UMTS CASE STUDY

In this section we present an example of MPSoC modeling with METROII. The example is based on the UMTS data link layer functional model mapped into various multiprocessing architectures. After describing the functional and architectural models of the UMTS data link layer, we enumerate and explore 48 different points in the design space. In Section 7.4, we detail the estimated execution times and processing element utilization, the design effort, the simulation cost breakdown, and an analysis of how events are processed during each simulation phase.

7.1. Functional Model

We focus on the Data Link (DLL) layer User Equipment Domain of the UMTS protocol [3^{rd} Generation Partnership Project 2004]. The DLL layer contains the Radio Link Control (RLC) and the Medium Access Control (MAC) sublayers and performs general packet forming. The RLC communicates with the MAC through different *logical* channels to distinguish between user, signaling, and control data. Depending on the required quality of service, the MAC layer maps the logical channels into a set of *transport* channels, which are then passed to the Physical Layer.

The UMTS application is separated into both receiver and transmitter portions and then further by the RLC and MAC functionality. Simulation consists of processing 100 packets, each packet being 70 bytes. The functional model is shown in Figure 8. The semantics follows the dataflow model, with blocking read and blocking write FIFOs. Both timed and untimed models were created to determine the advantages of functional/architectural separation—the timed model mixes both, while the untimed model relies on mapping with an architectural model to obtain performance metrics. Both models are presented in Figure 8.

The pure functional model allows processes to communicate in zero time through corresponding FIFO elements. The timed model, on the other hand, introduces a scheduler

and a timer. The scheduler is modeled as a finite state machine which controls the execution of the system. The activation of each process is controlled by the typical firing conditions of process networks, i.e., the availability of data at the input queues, and the availability of space at the output queues. These conditions are notified to the processes every time data is written to or read from the attached FIFOs. When a firing condition is satisfied, the process triggers the scheduler by sending a "Ready_to_Run" signal through the dedicated bi-directional scheduling channel and then waits for permission to start computation, which will be granted by the scheduler. For this example we have used a fixed-priority preemptive scheduler [Simalatsar et al. 2008]. Therefore, the permission to start computation will be granted to a processes when the resources are available and when no higher priority process is ready to run. The process runs to completion, and stops before the results are written to the output FIFO. Computation is carried out in logically zero time, however, the outputs are posted in the FIFOs only at the correct time calculated by the preemptive scheduler.

Other scheduling policies, such as round robin (RR), priority (PR) based, and first-come, first-serve (FCFS) are implemented in the architectural OS component, outside of the timed functional model. Therefore, mapping between functionality and MPSoC architecture uses the untimed model allowing for costs and scheduling to come purely from the architecture services.

7.2. Architecture Model

The architecture model assigns one task for each of 11 UMTS components (TR Buffer and PHY were not mapped as they represent the environment). The OS employs three different scheduling policies for each processing element: RR, PR and FCFS. The processing elements communicate through FIFO links or shared memories.

The runtime processing elements were fed C code reflecting the kernels of each UMTS component. The runtime processing element available for this case study is a cycle accurate datapath model of the Leon 3 Sparc processor. The pre-profiled processors use the same code but carry out offline characterization as detailed in [Meyerowitz et al. 2008] and [Densmore et al. 2006a]. The processors profiled were the ARM7, ARM9, and Xilinx's MicroBlaze, all of which are common in embedded and SoC applications and are widely documented.

7.3. Mapped System

Table I describes the 48 mappings that were investigated. These reflect systems with a high level of concurrency (11 PEs) all the way down to 1 PE. The mappings reflect system partitions broken down by Rx and Tx as well as RLC and MAC functionality. The partitions are enumerated at the bottom of the table. Each mapping is categorized into one of 9 separate *classes*, assigned to make analysis more manageable. The classes are based on the number of processing elements and the mix of pre-profiled and runtime processing elements. Mappings are further categorized as purely runtime processing based (RTP) elements, purely profiled processing (PP) elements, or a mix (MIX) of both.

7.4. Results

We analyze results presenting model estimated processing time as well as design effort, framework simulation time, and event processing. Five different models were utilized in the experiments: a timed SystemC UMTS model [Simalatsar et al. 2008], a timed METROII UMTS model, an untimed METROII UMTS functional model, a SystemC runtime processing model, and a METROII architectural model. We have used constraints to synchronize the execution of the components in the functional model as opposed to explicit synchronization primitives (for instance in SystemC), to make synchonization visible in the framework and because experiments have shown better

Table I. Mapping Scenarios for UMTS Case Study

#	Type	Partition	#	Type	Partition	#	Type	Partition
1	1: RTP	11 Sp	17	6: PP	2 μB (2), 2 A9 (3)	33	7: MIX	A7 (4), Sp (5), μB (6), A9 (7)
2	2: PP	11 μB	18	6: PP	2 A9 (2), 2 μB (3)	34	7: MIX	A7 (4), Sp (5), A9 (6), μB (7)
3	2: PP	11 A7	19	6: PP	2 A7 (2), 2 A9 (3)	35	7: MIX	A7 (4), μB (5), Sp (6), A9 (7)
4	2: PP	11 A9	20	6: PP	2 A9 (2), 2 A7 (3)	36	7: MIX	A7 (4), μB (5), A9 (6), Sp (7)
5	3: RTP	4 Sp (1)	21	7: MIX	Sp (4), μB (5), A7 (6), A9 (7)	37	7: MIX	A7 (4), A9 (5), μB (6), Sp (7)
6	4: PP	4 μB (1)	22	7: MIX	Sp (4), μB (5), A9 (6), A7 (7)	38	7: MIX	A7 (4), A9 (5), Sp (6), μB (7)
7	4: PP	4 A7 (1)	23	7: MIX	Sp (4), A7 (5), μB (6), A9 (7)	39	7: MIX	A9 (4), Sp (5), μB (6), A7 (7)
8	4: PP	4 A9 (1)	24	7: MIX	Sp (4), A7 (5), A9 (6), μB(7)	40	7: MIX	A9 (4), Sp (5), A7 (6), μB (7)
9	5: MIX	2 Sp (2), 2 μB (3)	25	7: MIX	Sp (4), A9 (5), A7 (6), μB (7)	41	7: MIX	A9 (4), μB (5), Sp (6), A7 (7)
10	5: MIX	2 μB (2), 2 Sp (3)	26	7: MIX	Sp (4), A9 (5), μB (6), A7 (7)	42	7: MIX	A9 (4), μB (5), A7 (6), Sp (7)
11	5: MIX	2 Sp (2), 2 A7 (3)	27	7: MIX	μB (4), Sp (5), A7 (6), A9 (7)	43	7: MIX	A9 (4), A7 (5), μB (6), Sp (7)
12	5: MIX	2 A7 (2), 2 Sp (3)	28	7: MIX	μB (4), Sp (5), A9 (6), A7 (7)	44	7: MIX	A9 (4), A7 (5), Sp (6), μB (7)
13	5: MIX	2 Sp (2), 2 A9 (3)	29	7: MIX	μB (4), A7 (5), Sp (6), A9 (7)	45	8: RTP	1 Sp
14	5: MIX	2 A9 (2), 2 Sp (3)	30	7: MIX	μB (4), A7 (5), A9 (6), Sp (7)	46	9: PP	1 μB
15	6: PP	2 μB (2), 2 A7 (3)	31	7: MIX	μB (4), A9 (5), A7 (6), Sp (7)	47	9: PP	1 A7
16	6: PP	2 A7 (2), 2 μB (3)	32	7: MIX	μB (4), A9 (5), Sp (6), A7 (7)	48	9: PP	1 A9

(1 = Rx MAC, Tx MAC, Rx RLC, Tx RLC), (2 = Rx MAC, Rx RLC), (3 = Tx MAC, Tx RLC)
(4 = Rx MAC), (5)(Rx RLC), (6)(Tx MAC), (7 = Tx RLC) (Sp = Sparc, μB = Microblaze, A7 = ARM7, A9 = ARM9)

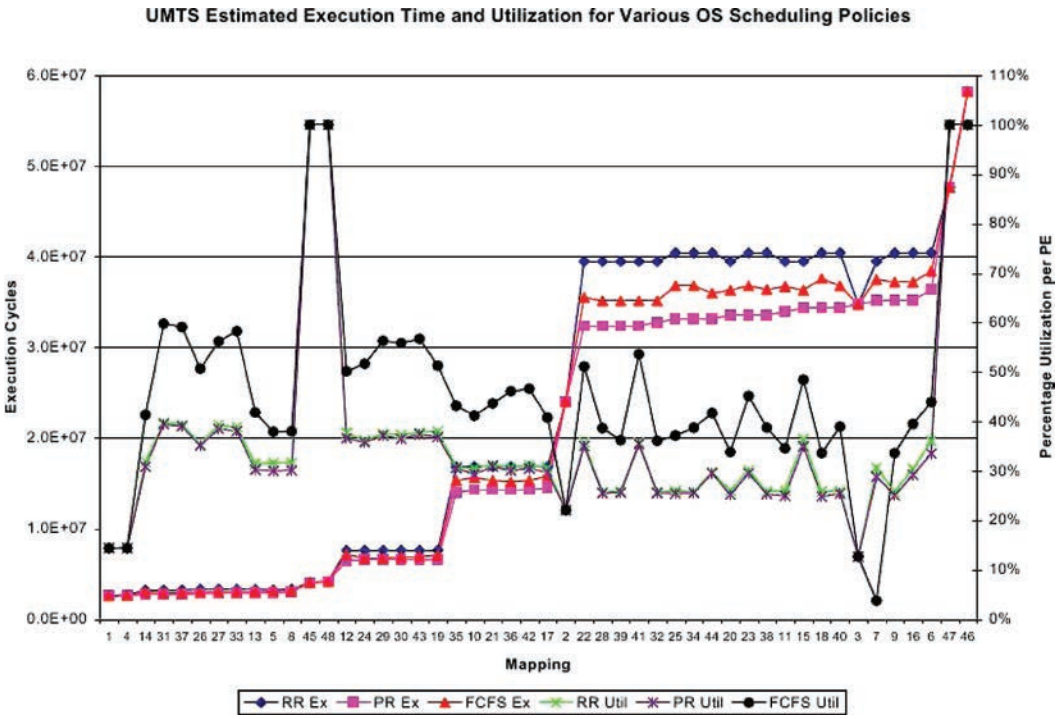

Fig. 9. UMTS Estimated Execution Time vs. Utilization.

simulation performance. The selection of constraints, functional model configuration, architectural model parameters, and mapping assignment is all achieved through small changes to the top level netlist. For example, to change the mapping of one method of a functional component into another method of an architectural task, only one line of code needs to be changed (see the code snippet of the constraint solver of Figure 2, lines of kind *MAP(FC, func_method, AC, arch_method)*, where FC is a functional component and AC an architectural component). To change parameters of the architectural model we only need to change the model annotation values. All results are gathered on a 1.8 GHz Pentium M laptop running Windows XP with 1GB of RAM.

Figure 9 shows the UMTS estimated execution times in cycles for the various mappings along with the average processing element utilization. The x-axis (mapping #) is ordered such that execution times increase moving toward the right.

Examining the RR scheduling first, as expected, the lowest and highest execution times are obtained with mapping #1 (11 Sparcs) and mapping #46 (1 μBlaze) respectively. Mapping #1 is 2167% faster than mapping #46. This shows a large range in potential performance across mappings. It is interesting to note that there are 23 different mappings which offer better performance than the 11 μBlaze or 11 ARM7 cores (mappings 2 and 3). This illustrates that communication becomes a bottleneck for some designs and despite having maximum concurrency those designs cannot keep pace with smaller, more heavily loaded mappings. Among all 4 processor systems, mapping #14 has the lowest execution time (two ARM9s used for receiver, two Sparcs used for transmitter). Mapping #31 has a similar execution time with 4 different processors (Rx MAC on μBlaze, Rx RLC on ARM9, Tx MAC on ARM7, and Tx RLC on the Sparc). Many of the execution times are similar and the graph shows that there are essentially 4 areas of interest. The lowest utilization values occur in the 11 processor setups (average of 15%). The highest is 100% for all single processor setups. The max utilization before 100% is 39%. This gap points to a deficiency in the RR scheduler used in the OS. One should also notice that for similar execution times, utilization can vary as much as 28% (mappings #41 and #32 for example).

The PR scheduling keeps the same relative ordering amongst the execution times but reduces them on average by 13%. The highest reduction is an 18% reduction (mapping #22 for example) and the smallest reduction is 9% (mapping #8 for example). The utilization numbers are actually reduced (made worse) as well by an average of 2%. The largest reduction was 7% (in mapping #6 for example) and the smallest was 1% (in mapping #31 for example). As expected there was no change to the utilization or execution times for mappings involving either 11 processing elements (fully concurrent) or those with 1 element (no scheduling options). The utilization drop results from high priority, data dependent jobs running before low priority, data independent jobs.

FCFS scheduling also does not change the relative ordering of execution times but is not as successful at reducing them. The average reduction is only 7%. The maximum reduction is 11% (in mapping #24 for example) and the minimum reduction is 4% (in mapping #5 for example). However, utilization is improved (increased) by 27%. The max improvement was 45% (in mapping #31 for example) and the minimum improvement was 20% (in mapping #5 for example). FCFS improves utilization by allowing low priority jobs to be processed in the same round as high priority jobs. A priority based scheme neglects their arrival time instead queuing large jobs in-lieu of processing waiting, small jobs. FCFS's round robin tie breaking scheme arbitrates between the small and large jobs.

The analysis of execution and utilization for UMTS shows that high utilization on multi PEs platform is difficult to obtain due to the data dependencies in the application. Also, some of the partitions explored do not balance computation well amongst the different processing elements in the architecture. Many of the more coarse mappings only make this problem worse. A solution is to further refine the functional model to extract more concurrency. From an execution time standpoint, scheduling can improve the overall execution time but not as much as is needed to make a large majority of these mappings desirable for an actual implementation. In fact only 11 (23%) of the mappings would actually be feasible for an actual UMTS system.

The size-utilization-performance trade-off (cost function) in relative units arranged in logarithmic scale is presented in Figure 10. This function is computed as $f = \text{util} \cdot \text{perf}/\text{size}$, hence an optimal solution for a real implementation is an architecture with the smallest physical size (because of the inverse proportion), and the highest performance and utilization values, in direct proportion to these values. All proportionality constants are set to 1. Low utilization numbers corresponds to the overcharge. To estimate the die area we have chosen the NXP LPC2119 as a microcontroller based on

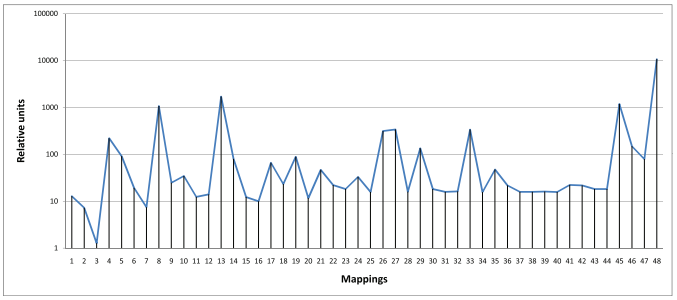

Fig. 10. Size/utilization/performance trade-off (optimization) function.

ARM7 processor core and NXP LPC3220 as a microprocessor based on ARM9. Their die areas are estimated to be 2600 mm^2 and 225 mm^2 respectively. For the Sparc we have chosen its ASIC implementation LEON3FT-RTAX with the die area equal to 191.52 mm^2. The die area of Xilinx Virtex-4 chip is estimated to be 1156 mm^2. A MicroBlaze on a Xilinx Virtex-4 (XC4VLX60) requires roughly .9% of the logic resources (based on a 1000 LUT design). To estimate the total circuit size we have simply summed the die areas of microcontrollers composing the hardware platforms of the evaluated mapping. For the FPGA implementation, one should carefully choose the size depending on what else can be mapped on the chosen FPGA, and considering that smaller FPGAs could also be used. For illustration purpose only, we will use the full die size here.

Mappings #3 has the worst parameters trade-off; its potential real implementation will have the largest physical size that cannot be justified by the performance or utilization. Mappings #8 (4 ARM9), #13(2 ARM9 and 2 Sparc), #45 (1 Sparc), and #48 (1 ARM9) shows the best size/utilization/performance trade-offs. Among them mapping #48 has the greatest value. It has one of the two smallest die areas, 100% utilization and one of the smallest execution time values (see Figure 9).

Design Effort. The untimed METROII UMTS functional model contains 12 processes while the architectural model may contain up to 26 processes depending on the configuration. The specification of the entire design is spread across 85 files and 8,300 lines of code. This is clearly a large design. The changing of a mapping is trivial however as it requires us to change a few macros and recompile 2 of the 85 files (2.3%) which takes less than 20 seconds. The 48 mappings could be done in less than 16 minutes.

Also METROII constraints for the read/write semantics of a FIFO only require 60 lines of code which is 1.4% of the total code cost. In fact the average difference of the entire conversion to METROII was only 1% per file. More than half of these lines (58%) have to do with registering the constraints with the solvers.

The conversion of a SystemC runtime processing model (for the Sparc processing element) to METROII only requires 92 additional lines for a direct conversion. This was a mere 3.4% increase (2773 lines to 2681). This includes adding support for loading new code at runtime, returning the cost of operation to the highest element in the hierachical netlist, and exposing events for mapping. Since METROII was created to easily import existing code, this result is encouraging.

Framework Simulation Time. Figure 11 illustrates the percentage of the actual simulation runtime spent in each of METROII's simulation phases for the 9 classes of mappings. The SystemC entry indicates the time spent in the SystemC simulation infrastructure upon which METROII is built.

As shown, on average 61% of the time is spent in Phase 1 (lowest section on the bar graph), 5% in Phase 2 (second section), and 17% in Phase 3 (third section). For

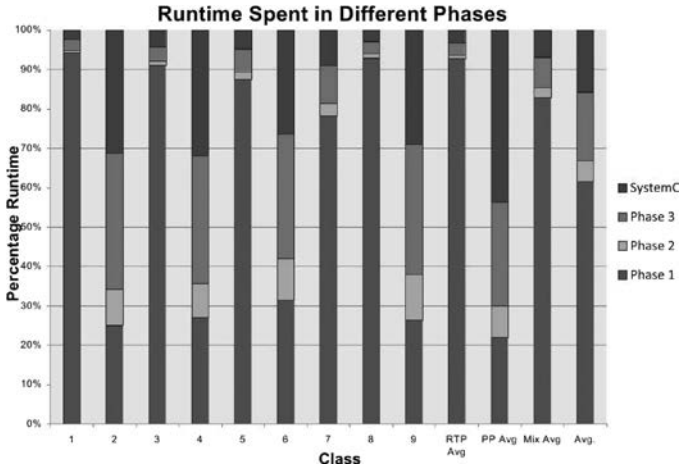

Fig. 11. METROII Phase Runtime Analysis.

models with only runtime processing elements (RTP) the averages are 93%, 0.9%, and 3% respectively. This indicates that in runtime processing, the METROII activities of annotation and scheduling are negligible in the runtime picture. For pure profiled (PP) mappings they are 21%, 7% and 26%. In this case one can see that METROII now accounts for a greater percentage of runtime than would normally occur (phase 1 alone is representative of other simulation environments). For mixed classes the numbers are 82%, 2.6%, and 7.6%. Again the runtime processing elements dominate. It should be noted that while PPs have higher averages, the average runtime to process 7000 bytes of data was 54 seconds. Phase 1 runtime (and SystemC overhead) are the main contributors to overall runtime.

If we consider the SystemC timed functional model, METROII timed functional model, and the METROII untimed functional model mapped to an architecture, the METROII timed functional model had an average increase of 7.4% in runtime for the 9 classes while the mapped version had a 54.8% reduction. This reduction is due to the fact that METROII phases 2 and 3 have significantly less overhead than the timer-and-scheduler based system required by the SystemC timed functional model.

In a comparison of the METROII timed model running without constraints and one running with them, the average decrease for the 9 classes of mappings was 25%. There was little discernible difference between the runtimes for each class here. This is not surprising since constraints have no direct role in the scheduling of processes - only in how the phases switch (simulation scheduling).

8. HETEROGENEOUS MODELING: INTEGRATING BIP AND METROII

The ability to integrate different models is one of the strengths of the METROII infrastructure. In this section we discuss a structural semantic preserving integration of a subclass of BIP systems [Basu et al. 2006], which supports formal reasoning, and show how to take advantage of the METROII capabilities to analyze a BIP functional model in the context of a separately defined architecture platform.

8.1. BIP Framework Overview

BIP supports a component construction methodology based on the superposition of three layers: (1) *behavior*, (2) *interactions*, and (3) *priorities*. At the lower level of BIP, atomic components contain *behavior* described by automata or Petri Nets and extended

with arbitrary computation, expressed as C/C++ functions/methods. Automata transitions are triggered by *ports*, which are action names used to specify interactions. *Interactions* in BIP define the execution paradigm and model both the synchronization as well as the communication mechanism among the components. An interaction is specified using *connectors*, which declare the sets of ports of different components that can synchronize in an execution step. The communication within an interaction is represented in the connector by an arbitrary computation on the local variables associated to the ports in the components. When the interaction is executed, the local variables are read and written through the *ports* of the components associated with the connector. An interaction is executed when all involved components are ready, i.e., when every component reaches a control location enabling a transition labeled by the required port. Whenever enabled, the interaction code is executed as an atomic step, and all involved components execute (concurrently) the local computations of the interacting transitions. When several interactions are enabled for execution, the choice is restricted according to *priority rules*.

The BIP toolbox provides tools for implementation on distributed execution platforms. Send/Receive BIP (S/R-BIP) models are a subclass of BIP models where multiparty interactions are replaced by extra components and protocols, called *engines*, using only binary send/receive communication. Any BIP model can be automatically transformed into an observationally equivalent S/R model, that is, preserving all functional properties [Bonakdarpour et al. 2010]. The S/R BIP models are used for the connection with METROII, since they preserve the structure of the system which is essential for proceeding with architectural exploration.

8.2. BIP to MetroII Integration

From the actual code point of view, we take advantage of the ability of BIP to generate C++ code, which can be easily imported in METROII by defining wrappers that mediate between the component and the framework. Therefore, an atomic *behavior* of the BIP model can be easily embedded into a *metroII* component. Instead, we must pay greater attention in the correct handling of the interaction semantics, which includes both data communication and synchronization.

Communication. While communicating the value of variables within an interaction, BIP connectors can operate on them using two functions: up() and dn(). The up() function reads the values from the components and performs a specifically defined action, while the dn() function assigns new values to these variables to make them available to the corresponding components. METROII ports represent only communication interfaces and do not have associated variables. Therefore, the functionality of the BIP connectors must be implemented as separate METROII channels that provide the up() and dn() functions to the connected components. More importantly, the interactions in the connector specify synchronization primitives that determine whether the connected components can proceed, and if they have to do so in lock-step. These aspects, and conflicts between simultaneously enabled interactions, are normally handled by the BIP simulation engine. However, replicating the simulation engine in METROII would be error prone from a semantic point of view. For this reason, we rely on the S/R-BIP distributed model, where control is handled primarily by the engines, as our starting point.

The S/R-BIP model more closely follows the communication structure required by METROII, where components communicate through dedicated channels that incorporate the computation performed by the BIP connectors. As an example, we integrated the parallelized version of the bitonic sorting algorithm [Batcher 1968]. The algorithm splits the sequence into subsequences (*d0*, *d1*, etc.) which are handled in parallel by an equal number of components that sort them in ascending or descending manner.

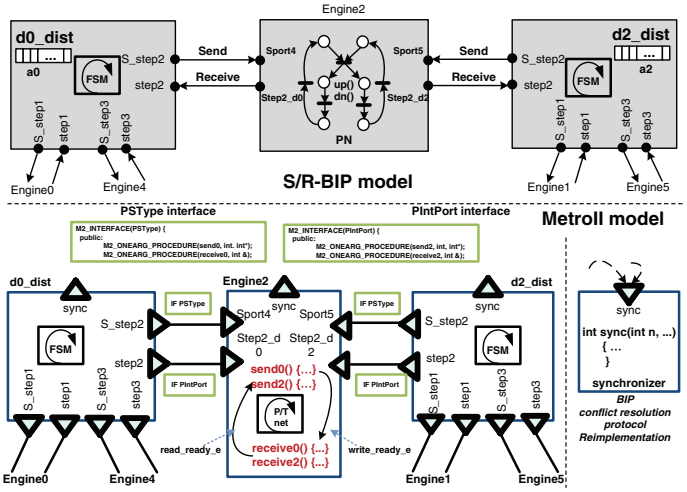

Fig. 12. S/R-BIP and METROII models of the bitonic sorting algorithm.

The bitonic sorting algorithm in BIP consists of four equal atomic components each one handling one part of the sequence.

The top of Figure 12 shows a fragment of the S/R-BIP model. The METROII translation of the same example is presented in the lower part of Figure 12. The figure shows the details of how the interaction between *Chiffre d0* and *Chiffre d2* (there labeled with the suffix *dist*, for distributed) has been realized with a dedicated communication component. Ports of the classical BIP model are replaced with a pair of ports (e.g., *S_step2* and *step2*) for *send* and *receive* communication, while the connector itself is replaced by a scheduler component called *Engine2*. Other ports and connectors of the model components are transformed in a similar way. The states of the components are split each into two states, one that sends the data to the scheduler, and the other that receives and synchronizes the updated values.

The *Engine2* scheduler is described as a Petri Net (PN) that manages explicitly the interaction of the BIP connector. The scheduler waits for the components to provide messages that denote they are ready to communicate, reads the values of their variables, and executes the up() and dn() functions. Then, it notifies back the components to resume their execution.

Synchronization. In the C code generated out of BIP models, including the S/R-BIP models, interactions between components are concretely realized by using a *synchronization* function call. This function call carries information about the current component state and the ports available for interaction. When the kernel receives all the requests from the components, it correlates the available ports with the list of registered interactions and sends the final decision back to the component, enabling or disabling their transitions to a new state. During code synthesis, this part of the model is not explicitly included into the generated C code of the BIP model. Therefore, in order to make the synthesized METROII models completely independent from the BIP core system, we have reimplemented this mechanism in a separate METROII component called *synchronizer*. This component plays the role of the BIP kernel where the decision about the next state transition of the component automata is taken. The *synchronizer* is shown in the bottom right part of Figure 12. The *synchronizer* component is present in any METROII model translated from BIP and remains identical for any of these models.

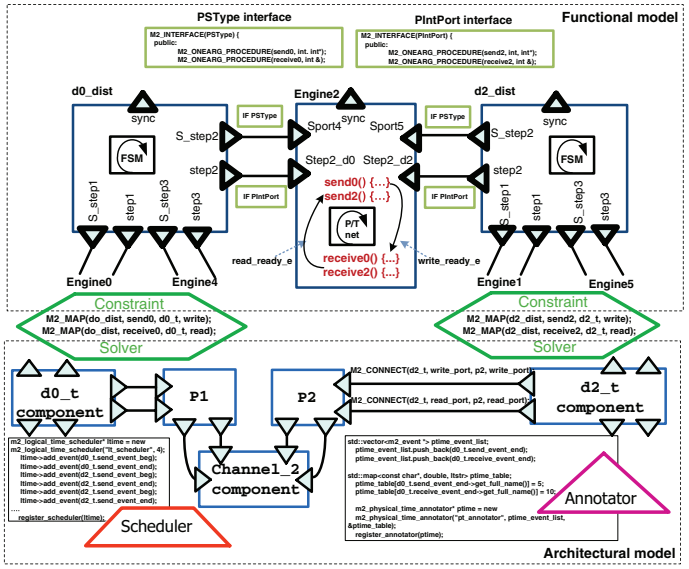

Fig. 13. Mapping of the Transformed BIP Model into METROII Architectures.

These model transformation rules have been implemented as an automatic BIP to METROII converter which takes as input the BIP functional model and produces a corresponding, structurally equivalent, METROII functional model.

8.3. Mappings and Results

Once the BIP model is imported in METROII, it is relatively easy to explore the performance of the system under different mappings. From the point of view of the BIP model, the architecture model and the METROII semantic can be seen as additional constraints on the execution of the BIP model. Thus the traces of the mapped model are a subset of all the possible traces of the original BIP model. Therefore the good properties in the BIP model are retained in the mapped model. For example, if the architecture can make sure that all the processes in the BIP model eventually have a chance to run, the mapped model will remain deadlock-free as long as the original BIP model is deadlock-free.

We show how this is done for the Bitonic Sorting Algorithm depicted in Figure 12. The METROII representation of this part of the functional model is repeated in the upper section of Figure 13. Components *d0_dist* and *d2_dist* include the threads that reuse the functionality of the BIP atoms, imported as C functions. Similar to the BIP model, each of these components has two ports to communicate with Engine2. These ports use two different interfaces, called PSType and PIntPort. The methods of these two interfaces are implemented inside the Engine2 channel. Engine2 also includes a thread that encapsulates the Petri Net implementation of the Engine2 atom of the S/R-BIP model.

METROII is very flexible in terms of changing the platform architecture and the performance metrics. In Figure 13, each *d(n)_dist* component, with $n \in \{0, 1, 2, 3\}$, is mapped using mapping constraints solvers to a dedicated task component *d(n)_t* of the architectural model. We have evaluated two different mappings of the tasks to the platform. In the first mapping (m1), each of the tasks is executed on a separate processing element which communicate through dedicated channels, realizing the functionality of the *Engine* components of the S/R-BIP model. In the second mapping (m2), pairs of components (e.g., *d0_dist* / *d1_dist* and *d2_dist* / *d3_dist*) share a processing element. Each task

Table II. Results of Mapping of the Integrated BIP Model

	m1RR	m1FCFS	m1mix	m2RR	m2FCFS	m2mix
set1	d0(de)sort = 100; write/read memory = 10 d1(de)sort = 200; write/read memory = 20 d2(de)sort = 300; write/read memory = 30 d3(de)sort = 400; write/read memory = 40					
Exe_t	1480	2050	1980	2590	2640	2670
p0:	20.27	14.63	15.15	34.75	34.09	33.70
Util p1:	40.54	29.27	30.30	81.08	79.55	78.65
% p2:	60.81	43.90	45.45			
p3:	81.08	58.54	60.60			
set2	d0(de)sort = d1(de)sort = d2(de)sort = d3(de)sort = 200 write/read memory = 20					
Exe_t	1040	1040	1040	1460	1460	1460
Util %	57.69	57.69	57.69	82.19	82.19	82.19

of the architectural model is annotated and scheduled using METROII annotators and schedulers. Performance numbers are selected arbitrarily to illustrate the mapping capabilities. However, METROII provides strong support for both off-line and run-time architectural components characterization as presented in Section 6.4.

We run our experiments with two sets of annotation data (set1 and set2) while the tasks are scheduled either using a pure Round Robin (RR) or a First-come First-served (FCFS) scheduler, or their mix where RR is used for scheduling tasks on the processing elements and non-preemptive FCFS is used as an arbiter for communication channels. The information about the mappings, annotation, scheduling policies and simulation results are summarized in Table II. Each column name denotes the type of mapping and scheduling policy. For instance, *m1FCFS* denotes mapping (m1) where tasks and packets communication are scheduled using the FCFS scheduler. We have studied two main parameters of the system, the total execution time of the distributed algorithm (Exe_t) and the processors utilization (Util) for both sets of annotation data. We calculate the utilization as the percentage of time when the processing element is actually performing some job. The rest of the time it is idle either due to interprocessor communication or due to the data from other components being delayed.

For the annotation data we assume that the execution time for ascending and descending sorting as well as write/read to/from the memory running on the same processing element to be the same. The time for these operations is marked in the table as *d(n)_(de)sort* and *write/read memory* respectively. In the first annotation set, the execution and write/read memory time for different processing elements is different while in the second set they are all equal. Therefore, for the first set we have different utilization of processing elements while for the second set the utilization will be equal for all of the processing elements, so we show only one value in the table. Comparing the total execution time of the first annotation set we see that while using the FCFS scheduling policy the distributed algorithm is executed more slowly than while using the RR scheduling. This is due to the tight synchronization among the computational blocks that require bidirectional communication of data after each part of the computation. In such cases the smart RR scheduling shows better results. As for the second set of data, we can see that there is no difference while using one or another scheduling policy. This is again explained by the tight inter-component synchronization that does not allow any communication between pairs of components unless the computation inside both of them is finished.

9. CONCLUSIONS

We presented METROII, a design environment for Cyber-Physical Systems (CPS). The environment was developed to cope with the problems presented by the heterogeneity

and complexity of CPS. In particular, we presented the design methodology supported by METROII, its modeling approach, how to couple functionality and implementation platforms and its simulation technology. We have shown adaptors to mediate between different models of computation. We also presented how to use the environment to model and analyze a UMTS system. To demonstrate the capability of METROII to work as an integration framework for other environments and tools, we showed how it was integrated with another popular design environment developed at Verimag, BIP. We believe that much remains to be done to solve all the challenges presented by CPS especially at the foundational level but that METROII can be effectively used to support CPS research and development in the future. Our current work is focused on developing foundational theories oriented towards the use of contracts in a CPS design flow [Sangiovanni-Vincentelli et al. 2012; Raclet et al. 2011], and the integration of these concepts in the tools to adequately support the design and analysis phase.

ACKNOWLEDGMENTS

The authors would like to thank Felice Balarin, Luciano Lavagno, and Yosinori Watanabe for their contributions to the Metropolis project, Albert Benveniste, Werner Damm, Alberto Ferrari, Edward Lee, Jan Rabaey, and Joseph Sifakis for the many interactions on embedded systems over the years.

REFERENCES

3rd GENERATION PARTNERSHIP PROJECT. 2004. General universal mobile telecommunications system (UMTS) architecture (release 6). Technical Specification TS 23.101, 3GPP.

ALEXANDER, P. 2006. *System Level Design with Rosetta*. Elsevier.

BAKSHI, A., PRASANNA, V. K., LEDECZI, A., MATHUR, V., MOHANTY, S., RAGHAVENDRA, C. S., SINGH, M., AGRAWAL, A., DAVIS, J., EAMES, B., NEEMA, S., AND NORDSTROM, G. 2001. MILAN: A model based integrated simulation framework for design of embedded systems. In *Proceedings of the Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES '01)*.

BALARIN, F., CHIODO, M., GIUSTO, P., HSIEH, H., JURECSKA, A., LAVAGNO, L., PASSERONE, C., SANGIOVANNI-VINCENTELLI, A., SENTOVICH, E., SUZUKI, K., AND TABBARA, B. 1997. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press.

BALARIN, F., LAVAGNO, L., PASSERONE, C., SANGIOVANNI-VINCENTELLI, A. L., SGROI, M., AND WATANABE, Y. 2002. Modeling and designing heterogeneous systems. In *Advances in Concurrency and System Design*, J. Cortadella and A. Yakovlev, Eds., Springer-Verlag.

BALARIN, F. AND PASSERONE, R. 2007. Specification, synthesis and simulation of transactor processes. *IEEE Trans. Comput. Aided Des. Integrat. Circuits Syst. 26,* 10, 1749–1762.

BALARIN, F., PASSERONE, R., PINTO, A., AND SANGIOVANNI-VINCENTELLI, A. L. 2005. A formal approach to system level design: Metamodels and unified design environments. In *Proceedings of the 3rd ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '05)*. 155–163.

BALARIN, F., WATANABE, Y., HSIEH, H., LAVAGNO, L., PASSERONE, C., AND SANGIOVANNI-VINCENTELLI, A. 2003. Metropolis: An integrated electronic system design environment. *Computer Mag.* 45–52.

BASU, A., BOZGA, M., AND SIFAKIS, J. 2006. Modeling heterogeneous real-time components in BIP. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM06)*. 3–12.

BATCHER, K. E. 1968. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference* (*AFIPS '68*). ACM, New York, NY, 307–314.

BONAKDARPOUR, B., BOZGA, M., JABER, M., QUILBEUF, J., AND SIFAKIS, J. 2010. From high-level component-based models to distributed implementations. In *Proceedings of the Conference Embedded Software (EMSOFT'10)*. ACM.

BROOKS, C., LEE, E., LIU, X., NEUENDORFFER, S., ZHAO, Y., AND (EDS.), H. Z. 2005. Heterogeneous concurrent modeling and design in Java (Volume 1: Introduction to Ptolemy II). Tech. rep. UCB/ERL M05/21, University of California, Berkeley.

CANCILA, D., PASSERONE, R., VARDANEGA, T., AND PANUNZIO, M. 2010. Toward correctness in the specification and handling of non-functional attributes of high-integrity real-time embedded systems. *IEEE Trans. Industrial Infor. 6,* 2, 181–194.

Carloni, L. P., Bernardinis, F. D., Sangiovanni-Vincentelli, A. L., and Sgroi, M. 2002. The art and science of integrated systems design. In *Proceedings of the 28th European Solid-State Circuits Conference (ESSCIRC 02)*.

Carloni, L. P., Passerone, R., Pinto, A., and Sangiovanni-Vincentelli, A. L. 2006. *Languages and Tools for Hybrid Systems Design*. Foundations and Trends in Electronic Design Automation Series, vol. 1, Now Publishers.

Davare, A., Densmore, D., Meyerowitz, T., Pinto, A., Sangiovanni-Vincentelli, A., Yang, G., Zeng, H., and Zhu, Q. 2007. A next-generation design framework for platform-based design. In *Proceedings DVCon*. 239–245.

Densmore, D., Donlin, A., and Sangiovanni-Vincentelli, A. L. 2006a. FPGA architecture characterization for system level performance analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE06)*.

Densmore, D., Passerone, R., and Sangiovanni-Vincentelli, A. L. 2006b. A platform-based taxonomy for ESL design. *IEEE Des. Test Computers 23,* 5, 359–374.

Grötker, T., Liao, S., Martin, G., and Swan, S. 2002. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA.

Jantsch, A. 2003. *Modeling Embedded Systems and SOC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers.

Kahn, G. 1974. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*. J. L. Rosenfeld, Ed., 471–475.

Karsai, G., Sztipanovits, J., Ledeczi, A., and Bapty, T. 2003. Model-integrated development of embedded software. *Proc. IEEE 91,* 1.

Kong, C. and Alexander, P. 2003. The Rosetta meta-model framework. In *Proceedings of the IEEE Engineering of Computer-Based Systems Symposium and Workshop*.

Ledeczi, A., Davis, J., Neema, S., and Agrawal, A. 2003. Modeling methodology for integrated simulation of embedded systems. *ACM Trans. Model. Comput. Simul. 13,* 1, 82–103.

Lee, E. A. and Sangiovanni-Vincentelli, A. L. 1998. A framework for comparing models of computation. *IEEE Trans. Comput. Aided Des. Integrat. Circuits Syst. 17,* 12, 1217–1229.

Mathaikutty, D., Patel, H., and Shukla, S. 2004a. EWD: A metamodeling driven customizable multi-MoC system modeling environment. FERMAT Tech. rep. 2004-20, Virginia Tech.

Mathaikutty, D. A., Patel, H., and Shukla, S. 2004b. A functional programming framework of heterogeneous model of computation for system design. In *Proceedings of the Forum on Specification and Design Languages (FDL'04)*.

Mathaikutty, D. A., Patel, H. D., Shukla, S. K., and Jantsch, A. 2006. UMoC++: A C++-based multi-MoC modeling environment. In *Application of Specification and Design Languages for SoCs - Selected Paper from FDL 2005*, A. Vachoux, Ed., Springer, 115–130.

MDA 2003. MDA guide version 1.0.1. Tech. rep. omg/2003-06-01, OMG.

Meyerowitz, T., Sangiovanni-Vincentelli, A., Sauermann, M., and Langen, D. 2008. Source level timing annotation and simulation for a heterogeneous multiprocessor. In *Proceedings of DATE08*.

Neema, S., Sztipanovits, J., and Karsai, G. 2003. Constraint-based design-space exploration and model synthesis. In *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT03)*.

OCL 2006. Object constraint language, version 2.0. OMG Available Specification formal/06-05-01, Object Management Group.

Passerone, R., Alfaro, L. d., Henzinger, T. A., and Sangiovanni-Vincentelli, A. L. 2002. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the 20th IEEE/ACM International Conference on Computer-Aided Design (ICCAD02)*. 132–139.

Passerone, R., Burch, J. R., and Sangiovanni-Vincentelli, A. L. 2007. Refinement preserving approximations for the design and verification of heterogeneous systems. *Formal Methods Syst. Des. 31,* 1, 1–33.

Passerone, R., Hafaiedh, I. B., Graf, S., Benveniste, A., Cancila, D., Cuccuru, A., Gérard, S., Terrier, F., Damm, W., Ferrari, A., Mangeruca, L., Josko, B., Peikenkamp, T., and Sangiovanni-Vincentelli, A. 2009. Metamodels in europe: Languages, tools, and applications. *IEEE Des. Test Comput. 26,* 3, 38–53.

Patel, H. D., Shukla, S. K., and Bergamaschi, R. A. 2007. Heterogeneous behavioral hierarchy extensions for SystemC. *IEEE Trans. Comput. Aided Des. Integrat. Circuits Syst. 26,* 4, 765–780.

Pinto, A., Bonivento, A., Sangiovanni-Vincentelli, A. L., Passerone, R., and Sgroi, M. 2006. System level design paradigms: Platform-based design and communication synthesis. *ACM Trans. Des. Automation Electron. Syst. 11,* 3, 537–563.

Raclet, J.-B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., and Passerone, R. 2011. A modal interface theory for component-based design. *Fundamenta Informaticae 108,* 1–2, 119–149.

SANDER, I. AND JANTSCH, A. 2004. System modeling and transformational design refinement in ForSyDe. *IEEE Trans. Comput. Aided Des. Integrat. Circuits Syst. 23,* 1, 17–32.

SANGIOVANNI-VINCENTELLI, A., DAMM, W., AND PASSERONE, R. 2012. Taming Dr. Frankenstein: Contract-based design for cyber-physical systems. *European J. Control 18,* 3.

SANGIOVANNI-VINCENTELLI, A., YANG, G., SHUKLA, S. K., SZTIPANOVITS, J., AND MATHAIKUTTY, D. A. 2009. Metamodeling: An emerging representation paradigm for system-level design. *IEEE Des. Test Comput. 26,* 3, 54–69.

SANGIOVANNI-VINCENTELLI, A. L. 2002. Defining platform-based design. *EEdesign*.

SIMALATSAR, A., DENSMORE, D., AND PASSERONE, R. 2008. A methodology for architecture exploration and performance analysis using system level design languages and rapid architecture profiling. In *Proceedings of the 3rd International IEEE Symposium on Industrial Embedded Systems (SIES08)*.

Received December 2011; revised August 2012; accepted October 2012