Metro II Execution Semantics for Mapping

Douglas Michael Densmore
Trevor Conrad Meyerowitz
Abhijit Davare
Qi Zhu
Guang Yang

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-16
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-16.html

February 18, 2008

Acknowledgement

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

Metro II Execution Semantics for Mapping

Douglas Densmore, Trevor Meyerowitz, Abhijit Davare, Qi Zhu, Guang Yang

February 18, 2008

Abstract

This document presents three proposals for the execution semantics of mapping in Metro II. Mapping is the relationship between what a system does (functionality) and how it does it (architecture). The main concern is whether the functionality and architecture models should execute concurrently or sequentially during simulation. Proposal #1 presents sequential execution with the functionality being executed before the architecture. Proposal #2 also presents sequential execution, but with the architecture executing before the functionality. Finally, Proposal #3 presents concurrent execution. Processes are present in the architecture to execute simultaneously with the events mapped to them in the functionality.

Each of these three proposals is demonstrated on a set of design scenarios with hand traces illustrating their execution. Additionally general assumptions, glossary terms, and proposal-specific assumptions made regarding the execution semantics are discussed. Finally, the proposals are compared and contrasted, especially regarding how they can properly implement the examples and the general semantic assumptions.

Contents

1 Introduction

This document details the ongoing meetings among the authors to discuss the Metro II execution semantics of mapping. The outcome of these meetings, held primarily in the Summer of 2007, is *a set of three proposals* for the execution semantics of mapping in Metro II. The purpose of this document is to clearly illustrate the pros and cons of these three proposals as well as provide concrete design scenarios by which these and future proposals will be judged. In order to illustrate the semantics, hand example traces are provided for each proposal. These hand examples are created at a level of granularity which provides enough insight to compare the proposals without overwhelming the reader.

More importantly however, the design scenarios will serve as "necessary, but not sufficient" benchmarks for any modifications or other proposed execution semantics. The design scenarios are intended to capture a wide variety of potential situations a designer may want to model. We propose that the tables which contain the hand traces in this document set the standard by which Metro II execution is presented.

This document describes four main pieces: the general execution semantics shared by all proposals, the execution semantics' assumptions, the individual design scenarios, and each proposal with its accompanying hand execution traces for the design scenarios. It is the authors' hope that this document will prevent ambiguity regarding the semantics and facilitate discussions on Metro II. This is achieved by clearly defined scenarios and a standardized way to present Metro II execution.

It should be noted as well that this document is not intended to be a comprehensive discussion of Metro II. We refer the reader to [2] for more information (which should be read before this document). Aspects of this document may later be used for a future journal submission (i.e. an IEEE Transactions on Computers special issue). Also the reader should inspect the MetroII code base, as this document may not reflect the latest implementation details.

2 Execution Semantics Overview

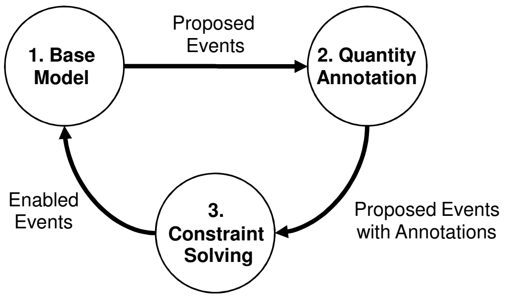


Figure 1: Metro II: Three Phase Execution

This section describes the common portion of the execution semantics for all of the proposals. Figure 1 illustrates the current 3-phase execution semantics consisting of:

1. **Base phase** - Where components execute concurrently and propose events.
2. **Quantity annotation phase** - Where proposed events are assigned physical quantities such as time or power.

3. **Constraint solving phase** - Where annotated events are enabled to execute or wait based on scheduling algorithms and constraints.

Metro II allows for “components” which encapsulate imperative code as well as “processes”. Processes are a single thread of execution which can propose events, perform computation, and access component interfaces. Processes are what make components “active”. Active components communicate with “required ports” (where service calls originate) and “provided ports” (where services are implemented). A component is “passive” if it has no process associated with it. A passive component only executes when it is invoked by a method call on one of its provided ports.

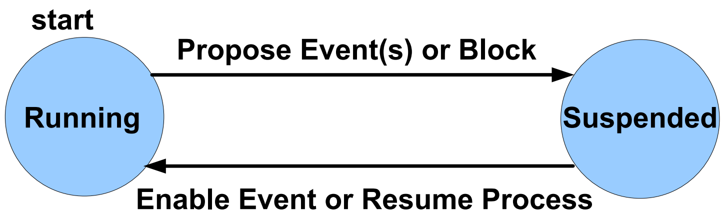


Figure 2: Metro II: Process States

Figure 2 illustrates the two states that processes can have in Metro II. Circles in the diagrams represent process states and arcs are labeled with the transition conditions for the states. Processes can either be running or suspended. All processes start as running, and execution switches from the base phase to the quantity annotation phase when all processes in the system are suspended. A process can be suspended either by proposing one or more events, or if it is is internally blocked (e.g. waiting to acquire a mutex). Blocked processes are outside of the realm of control of the Metro II framework, but are visible to it. For instance, a process may wait on a SystemC event (not directly exposed to the framework) and stop waiting when that SystemC event is notified. This may occur on a blocking read from an empty FIFO, for instance. In previous discussions, simulation would change phase only when all processes had proposed events. To allow for easier design import (where we make no assumptions on internal component behavior), this requirement has been relaxed to allow for phase switching when each process has proposed an event or is blocked (e.g. waiting on a SystemC event).

Figure 3 illustrates the state diagram for events. Events can be in one of three states: inactive, proposed, or annotated. Transitions between event states occur on phase changes. Figure 4 illustrates how each phase is associated with specific transitions. An inactive event is one which has not been proposed by a process. This is the initial state of all events. The self loop indicates that events can stay inactive over multiple phases (i.e. sometimes indefinitely). When an event is proposed by a process, it transitions from inactive to proposed. Next, the annotation phase runs on all proposed events and each event transitions from the proposed state to annotated state. After annotation, the constraint solver is run on all annotated events and may disable some of them. The self loop in the annotated state indicates that the disabled event keeps the same annotations, and will stay in the annotated state. An event may also transition back to the proposed state if it is disabled AND requires re-annotation. If the event is enabled in the constraint solving phase, it will then transition back to inactive and its associated process can then resume execution (i.e. enter the running state). Notice that we have removed an “enabled” event state (which was present in previous proposals) since it was effectively just a transition from the annotated to inactive states.

An example execution of a functional model is provided in Table 1 for the example design shown in Figure 5. This is a simple writer component with a FIFO. The writer is going to write one transaction

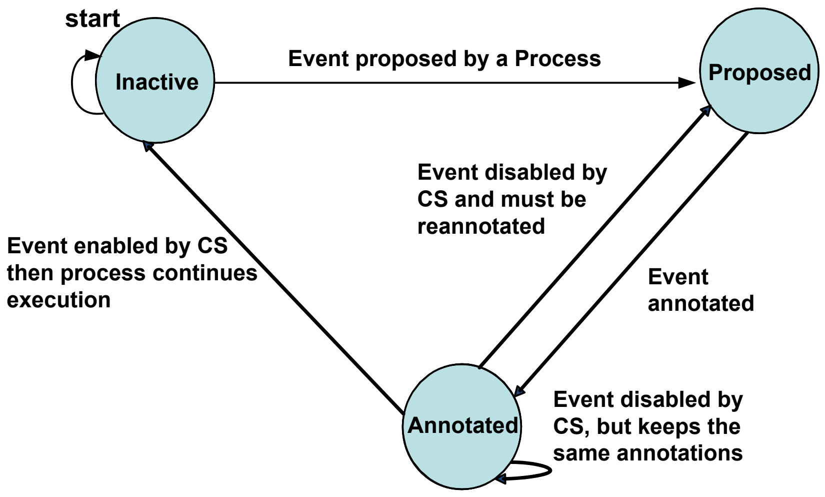

start
Inactive
Event proposed by a Process
Proposed
Event disabled by CS and must be reannotated
Event enabled by CS then process continues execution
Event annotated
Annotated
Event disabled by CS, but keeps the same annotations

Figure 3: Metro II: Event States

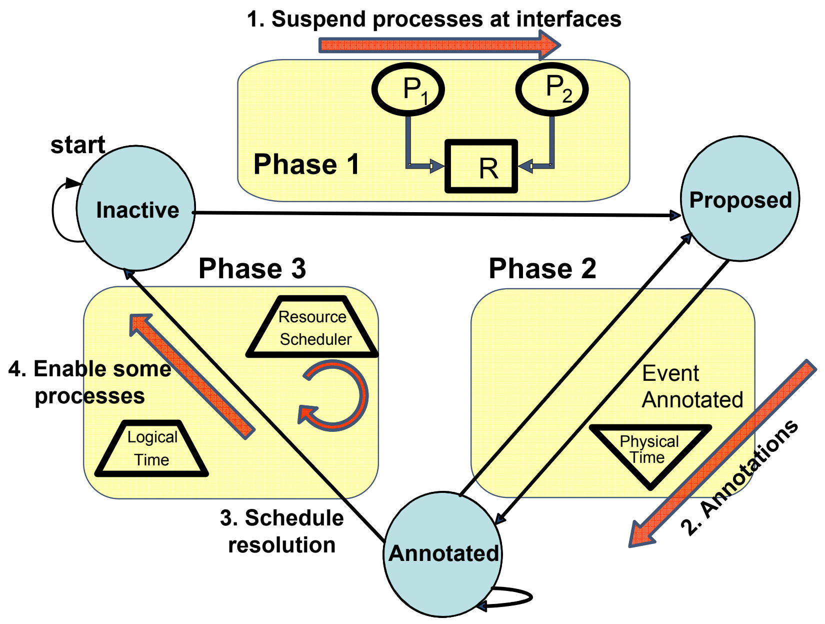


Figure 4: Metro II: Event States and Phase Correspondence

to the FIFO. The first row of the table is the first set of three phases in which a “write begin” event is proposed on the required port and eventually enabled. Notice two events are enabled as denoted by → in the third phase. The first is the required port event. The second is the provided port event which may or may not have been explicitly proposed in the first phase. The second row is the final set of three phases when the “write end” is proposed and eventually enabled. In this case the provided port

Round	**Process**	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
1	P1	Writer	W	B	Proposed	Annotated	Enabled → FIFO W B
2	P1	FIFO	W	E	Proposed	Annotated	Enabled → Writer W E

Table 1: Metro II: Basic Execution Example

end event is explicitly proposed and the required port end event is enabled by implication.

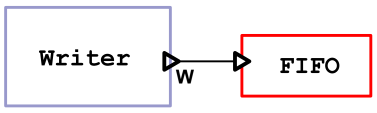


Figure 5: Metro II: Basic Functional Model

3 Assumptions

A few assumptions are shared between the multiple proposals to be introduced in Section 5. One way in which additional proposals can be seen as "improvements" would be if they remove or relax these assumptions. Some of these assumptions are inherent in Metro II and will be built into the simulation framework (3 phase execution for example) others are part of whatever the underlying design framework is (wait and notify based blocking in SystemC for example). The assumptions fall into the following categories:

- Suspension - these assumptions deal with how processes can be prevented from executing by other processes. This usually occurs because of shared resources or the process waiting on some condition to become true. In particular, we are concerned with "internal" suspension, which is also known as blocking. "External" suspension is handled in the constraint solving phase and is detailed in the scheduling assumptions.
- Scheduling - these assumptions deal with what is expected of the the third phase of the execution semantics. The constraint solving procedures in the third phase can implement any number of policies.
- Mapping - these assumptions deal with the way in which the functional and architectural models are related. These may have to do with the relationship between events in each model or the allowance of multiple functional components being assigned to single architectural components.

3.1 Suspension Assumptions

Suspension occurs when a process proposes one or more events, or if it becomes blocked due to a dependence on some other process or resource (e.g. a mutex). Suspension at the event level is handled quite nicely by the framework, so these assumptions primarily concern themselves with suspensions of processes between event proposals (e.g. blocking). For all three semantic proposals, we assume that suspension may occur within functional methods and the architectural services that they are mapped to. Each proposal will determine individually what should occur in a model if suspension in the other model occurs.

3.2 Scheduling Assumptions

Resource (and general) scheduling must be properly handled (in such a way that the functional and architectural model make progress) by the semantics. We assume that the design is created in such a way that it can be scheduled such that the application completes execution without deadlocking.

3.3 Mapping Assumptions

Mapping is defined between ports. Method calls on ports generating events are the impetus behind the progression of the simulation. When a mapped functional method call executes, then the architectural service that it is mapped to will also execute. The events of the mapped method call and the service call that it is mapped to have a "correspondence". What this correspondence is varies between the proposals and is described in each proposal's section.

In addition, mapping allows for many functional methods to be mapped onto one architectural service. For example many functional components (DCT, Quant, etc) could be mapped to one architectural service (i.e. a CPU). However, a functional component cannot be mapped onto multiple architectural components. For example, the DCT component functionality cannot be mapped between both CPU1 and CPU2.

4 Design Scenarios

In this document we evaluate the three different proposals for the execution semantics for mapping phase by phase on different design scenarios. The first scenario is one which features Metro II components with single readers, single writers, and separate explicit execution components. An application which has this structure is a segment of an MJPEG encoder. This can be designed as a simple producer-consumer with a FIFO between the producer and the consumer. The second scenario involves N producers writing to a shared FIFO and illustrates shared resource concerns. In each of these scenarios the functionality will be mapped to an architecture featuring processors connected to a shared bus (and memory). Additionally, we assume that only one event is proposed per phase from each component and that there is only one process per component.

4.1 Dataflow Example

The first design scenario used to illustrate the execution semantics of mapping examines a MJPEG encoder system. The functional model consists of 4 major components. These are "Source", "DCT", "Quant", and "Huffman". These represent the major computational components of MJPEG. Each of these components have a number of "required ports". These required ports either interact with a "FIFO" component, or an "ExX" component (which encapsulates the specific execution of that component). The FIFO is a communication component which will supply "provided ports" for READ and WRITE. The ExX component has a provided port for execution. The execution component (ExX) is created to expose the execution events so that they can be mapped to the architectural model (mapping is assigned between ports in the functional and architectural models).

The architecture model consists of four processor components, "Arch1, Arch2, Arch3, Arch4". Each of these has three provided ports (Read(R), Write(W), Execute(Ex)). In addition there is a bus component. It has a provided port for each of the previous 4 components. Each ArchX component will also have a required port to access the bus.

These models are shown in Figure 6. The top half of the illustration is the functional model and the bottom half is the architecture model. The ports are connected as shown. For the purposes of brevity and clarity, only a subset of this design scenario will actually be examined in the hand traces. The subset which is used in the hand example traces is boxed in on the figure. This subset simply consists of the "Source" and "DCT" aspects of the functional model and their architectural counterparts. This captures the key producer and consumer relationship which is the heart of this design scenario.

This design scenario illustrates: shared resources, blocking read and write operations, and mapping individual communication elements (FIFOs) to a shared communication element (Bus).

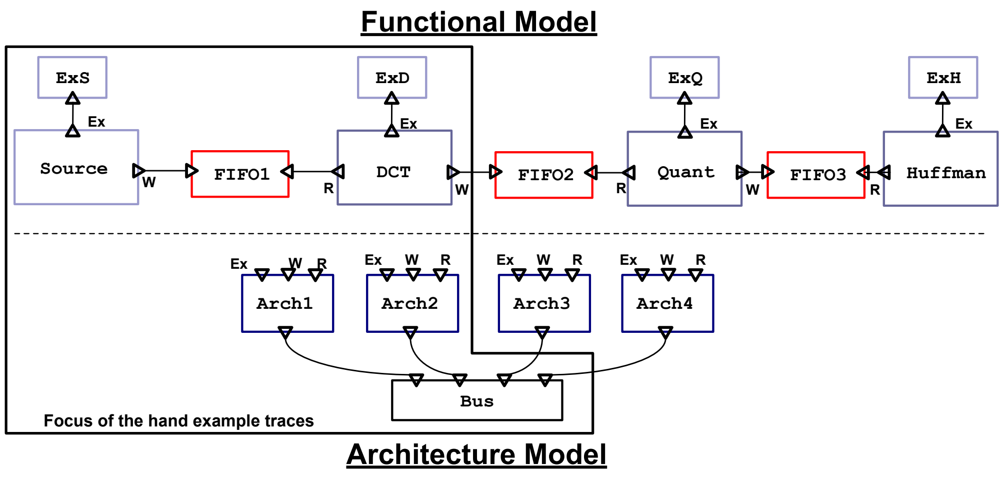


Figure 6: Dataflow Design Scenario: MJPEG Encoding

4.2 Shared FIFO Example

The second design scenario is shown in Figure 7. N producers, "Prod1" to "ProdN", will all attempt to write to the FIFO. The FIFO has N provided ports, one for each producer, all of which will provide a WRITE function. Each producer has one required port used to invoke the WRITE request. The architecture simply consists of "Arch1" to "ArchN" all with provided ports for the WRITE service. These N components are connected to a single bus and memory component.

This design scenario is intended to illustrate how the proposals deal with functional non-determinism, mutual exclusion, and atomicity. In this example, the order in which the producers write to the FIFO is nondeterministically determined in the functional model. After mapping, the architecture should enforce an ordering (e.g. based on resource scheduling). This ordering may depend on the bus scheduler employed in the architecture, for instance. The hand traces to follow in future sections will only use two producers to make the example manageable.

5 The Proposals

This section will present the three proposals for the execution semantics of mapping in Metro II. Each section will contain three components. The first is a table which summaries the proposal. This includes the correspondence between events in the architectural and functional model as well as any assumptions that the proposal makes. Secondly each section is an illustration of the mapping structure which shows how a required port call in the functional model is related to both architecture execution as well as the

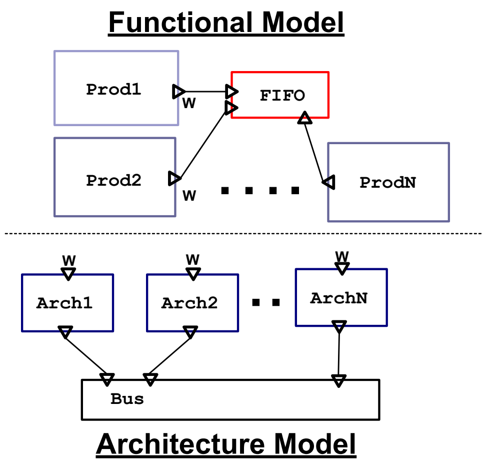


Figure 7: Shared FIFO Design Scenario

Proposal	**Execution Order in Simulation**	**Mapping Structure** (Func $\leftrightarrow$ Arch) Port	**Event Correspondence**	**Requires Blocking**
1	Functionality then Architecture	Required $\leftrightarrow$ Provided	FR_b, FP_b FP_e, AP_b AP_e, FR_e	Yes
Goals The initial proposal to get a functioning framework. Prevent unnecessary architecture execution by ensuring functional correctness/progress first.				
Unique Assumptions Architectural model can not directly influence functional model execution. Functional model must resolve non-determinism in the absence of an architecture. Architecture components which are mapped are passive.				

Table 2: Proposal 1 Overview

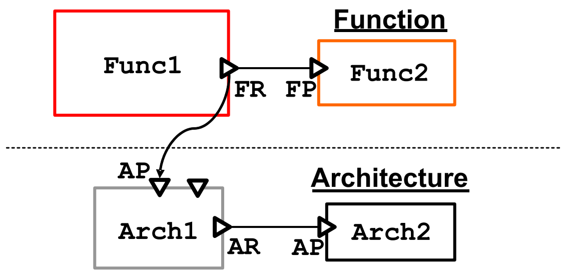
provided port events on the functional side. Finally a small description of the proposals operation is provided as well.

5.1 Proposal 1: Execute Functionality then Architecture (FTA)

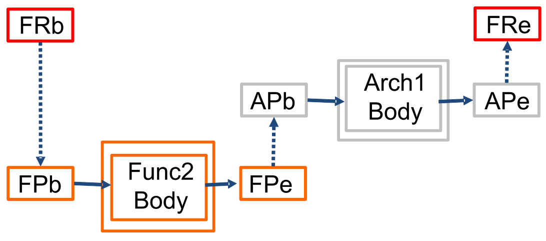
The first proposal is a sequential proposal in which the functional model begins execution before the architectural model. Some of the highlights of this proposal are capture in Table 2.

Figure 8 shows both a structural and "call graph" view of mapping in the first proposal. The ports in these and future diagrams are specified with an "F" or an "A" if they are in the functional or architectural model respectively. Also ports are designated as "R" or "P" if they are required or provided. "b" and "e" designate the begin and end events respectively. These designations can be combined. For example "FPe" would indicate the end event of a functional, provided port.

Figure 8(a) shows the mapping structure of a system using this proposal. The functional model contains a method call to Func2 from Func1. The mapping of this method call occurs by assigning events proposed by Func1.FR to events proposed by Arch1.AP. This is considered a required port to



(a) Mapping Structure



(b) Call Graph

Figure 8: Mapping Semantics in Proposal 1

provided port mapping structure.

Figure 8(b) shows the "call graph" of the system. Boxes with single line borders are events. Boxes that have two line borders are code blocks that may or may not contain events. The arrows indicate program flow (from left to right). If an arrow is dashed it means that two events connected to it are treated as a single event by the framework (this is the "correspondence" discussed in Section 3.3). The functional component "Func1" calls a method from component "Func2". This is mapped to the architectural component "Arch1", which further uses architectural component "Arch2" when providing the service.

Execution in this proposal occurs as follows: component Func1 contains a process. This process is responsible for proposing event "FRb". "FRb" corresponds to "FPb" (in Func2). Once these events are enabled, "Func2 body" (the code body of the function call to Func2) can now execute. Upon completion, "FPe" (Func2) will be proposed. This event corresponds to "APb" in the architecture. The architecture body "Arch1 body" can now execute and culminate with the proposal of "APe". As shown, "APe" corresponds to "FRe" which completes the execution.

As shown, mapping of methods is carried out by invoking the mapped architectural service in the process of the caller AFTER the corresponding functional method has completed execution.

5.2 Proposal 2: Execute Architecture then Functionality (ATF)

This proposal is in some regards the opposite of the first proposal. It is summarized in Table 3.

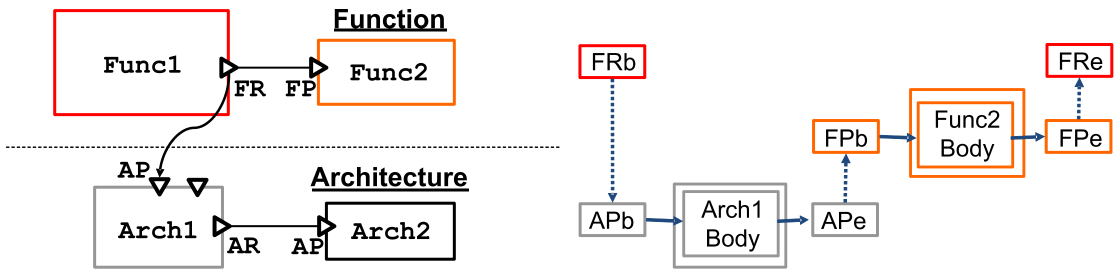
Figure 8 for the previous proposal shows the "call graph" for execution between the functional and architectural models. Basically, the functional methods need to be completed before corresponding architecture services start. However, in some cases this approach might not be able to reflect all the situations in the mapped system.

For instance, let's consider the shared FIFO example presented earlier. Proposal #1 cannot assure that the architectural ordering decision impacts the functional execution, since the function methods will finish before the architecture is invoked. Therefore, the shared FIFO example may not work as expected with proposal # 1. Essentially functional non-determinism can not be resolved by the architecture. Such operations may be desirable when the architecture is better able to perform given the opportunity to make decisions based on its state (free resources for example). This also removes some scheduling burden from other areas of the system.

This second proposal remedies this problem by requiring that architecture services should be completed before the corresponding function method starts. The new "call graph" is shown in Figure 9(b). This proposal shares the same mapping structure as proposal # 1.

Proposal	Execution Order in Simulation	Mapping Structure (Func $\leftrightarrow$ Arch) Port	Event Correspondence	Requires Blocking
2	Architecture before Functionality	Required $\leftrightarrow$ Provided	FR_b, AP_b AP_e, FP_b FP_e, FR_e	Yes
Goals Allow the architecture model to resolve non-determinism in the functional model. Provide a more intuitive semantics of how a architecture resource driven system would behave.				
Unique Assumptions Whenever the mapped functionality "internally" blocks, the corresponding architecture "internally" blocks. If the architectural model "internally" blocks, the functional model does not have to "internally" block (but can). Architecture components which are mapped are passive.				

Table 3: Proposal 2 Overview



(a) Mapping Structure (b) Call Graph

Figure 9: Mapping Semantics in Proposal 2

5.3 Proposal 3: Execute Functionality and Architecture Concurrently (FAC)

The third proposal is summarized in Table 4.

There is a consensus that the MetroII environment is rooted in the Platform-Based Design methodology [1], where the functional model and the architectural model meet in the middle with a set of well-defined services as the binding contract. To the architectural model, the middle point represents what services it can provide to implement certain functionalities, or to estimate the implementation cost. To the functional model, the middle point describes its need of services to achieve its entire function. If we look at the design scenarios, the services that are exposed at the middle point include *execute, read_fifo and write_fifo*. Therefore, the architecture model has to provide at least those services. As the three proposals exhibit, there are multiple possibilities in terms of which ports to map. In fact, the syntactic difference does not really matter.

What matters is the role of the mapped architectural component and its relationship to the components on the functional side. Imagine on the functional side, the source component calls *write_fifo* that is provided by FIFO1. No matter which part in the connection (the required port, the provided port, the connection) is mapped to the architecture, we expect the architectural service at some point to perform *write_fifo*. In that sense, the architectural counterpart corresponds to FIFO1, where both of them react to *write_fifo* request and do the job. If we can agree on this correspondence, then any

Proposal	**Execution Order in Simulation**	**Mapping Structure** (Func ↔ Arch) Port	**Event Correspondence**	**Requires Blocking**
3	Concurrent Functionality and Architecture	Provided ↔ Provided	FR_b, AP_b, FP_b FR_e, AP_e, FP_e	No
Goals More closely follow a platform-based design philosophy. Introduce concurrent architectural and functional execution. Remove any suspension assumptions between models				
Unique Assumptions Finer granularity of execution through resource (i.e. bus, cpu, etc) access protocols. Requires more opportunities for coordination between functional and architectural models. Architecture components which are mapped are active.				

Table 4: Proposal 3 Overview

mapping syntax will work. i.e. on the functional side, the required port, the provided port and the connection each represents a pair of events; on the architectural side, the service is also represented by a pair of events. Then mapping establishes another pair of correspondences between the two pairs of events. However, from the methodology point of view, where we emphasize the meeting point between functional and architectural models, mapping connections or provided ports from the functional side seem to be better choices.

When running the functional and the architecture models together[1], we would like the mapped services on both sides to finish simultaneously, because this will provide the most information about how an architectural model implements a functional model. However, there are concerns about the fact that suspension of processes on either side would prevent the entire mapped system from progressing. This is primarily caused by the semantics mismatch of the services from both sides. By carefully designing the consistent services, we should be able to make the mapped system work even with blocking behaviors on either or both sides. Sections 6 and 7 will examine how the design scenarios work under the 3rd proposal, which does not concern itself with blocking behavior at all and assumes the function and architecture run concurrently all the time.

The mapping structure and "call graph" for the 3rd proposal are shown in Figure 10. Notice that in proposal # 3, provided ports in the functional model are mapped to provided ports in the architectural model as well. This is different from the previous two proposals. Also in the "call graph" it is shown that "correspondence" points must be created in the form of "protocols" in order to create a more granular operation at the event level in each model. An example of such a "protocol" will be shown in more detail in the hand traces for proposal # 3.

5.4 Comparison of Proposals

Proposal 1 allows for simple functional models which do not necessarily need data driven or resource driven execution. Non-determinism will not be influenced by the architecture but only within the functional model itself. Since the architecture executes after functionality, the architecture does not need to capture all aspects of the functional method.

[1]Note that we can also run functional model first, recording the service demands, then drive the execution of the architectural model. But this eliminates behavior where the feedback from the architectural model would affect the execution of the behavior model.

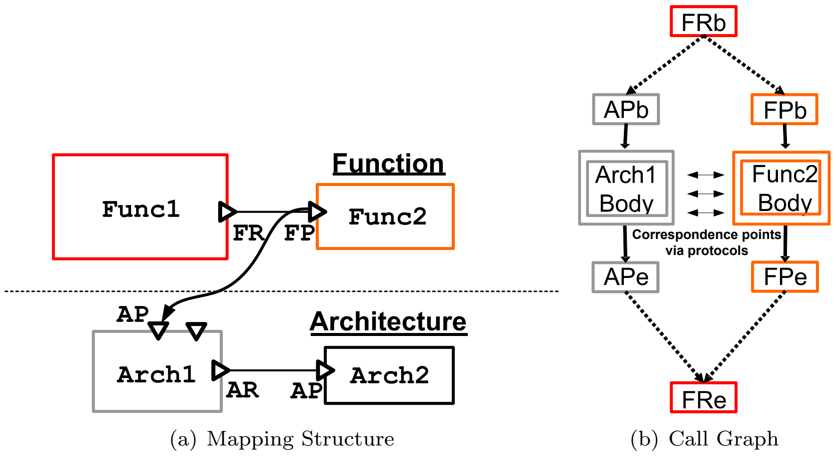


Figure 10: Mapping Semantics in Proposal 3

Proposal 2 elevates architecture to a more important position. The architecture does need to capture all aspects of the functional model (suspending behavior for example). However, it can also have more impact on the functional execution order. Functional models which want to be influenced by the architecture should use proposal 2.

Proposal 3 is the least constrained execution model that allows for the most parallelism in simulation. The architecture has the ability to influence the functional model. However it requires the creation of a more granular execution which provides explicit opportunities for the two models to synchronize with each other.

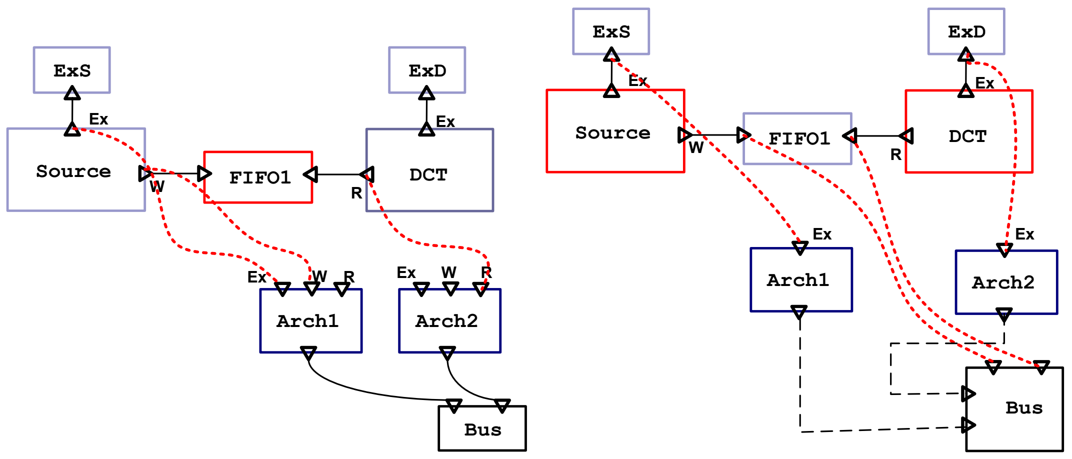
Proposals 1 and 2 require only passive components in the architectural model while proposal 3 requires active components. This has implications for the designers of architectural models and compatibility between proposals.

6 Execution Traces for the Dataflow Example

In this section, execution traces for all three proposals will be evaluated with the dataflow (MJPEG) design scenario from Section 4.1. In Figure 11, the mapping relationships are shown for all three proposals. Proposals # 1 and # 2 are between "required" and "provided" ports while proposal # 3 is between "provided" ports.

6.1 Proposal #1

This section will provide a phase-by-phase execution of the MJPEG example. The execution trace presented starts with the "Source" process executing and writing data to its FIFO to be read by the "DCT". The DCT will attempt to read data to begin execution but will have to wait until data is available. Each table row shows the process involved, the component name, the port generating the event, if the event is a beginning or end event, and its status in each phase of the round. The tables are broken up into rounds, each running through the different phases, where 1 round is equal to 3 phases (a cycle in the MetroII execution semantics).



(a) MJPEG Mapping in Proposals 1 and 2 (b) MJPEG Mapping in Proposal 3

Figure 11: MJPEG Port Mapping Structure

Before beginning the discussion of the phases, we present an alternate view of the tables. This view may be useful to the reader in the event that the tables presented later are unclear.

Process	**Event or Method Call**	**Phase 1**	**Phase 2**	**Phase 3**
P1	(Source/ExS/Arch1).Ex.B	Proposed	Annotated	Enabled
P2	(DCT/FIFO/Bus).R.B	Proposed	Annotated	Enabled

Whereas the tables to be presented have → and corresponding events shown in phase 3, this table presents information differently in several ways.

1. Instead of component/port/(begin/end) for the different columns, we instead have 'event or method call', where events are B, E, and method calls are the actual functions.

2. All corresponding events and method calls for each process are listed in the new field.

6.1.1 Round 1

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Source	Ex	B	Proposed	Annotated	Enabled → ExS Ex B
P2	DCT	R	B	Proposed	Annotated	Enabled → Fifo1 R B

In phase 1 all processes propose their begin events. Since both processes have proposed events, they will transition to a suspended state and the simulation phase can switch. The "source" component wants to execute and the other process wants to read from its FIFO.

In phase 2 all proposed events are annotated. The details of the annotations are left out for clarity. Typically annotations will detail the physical time required for the operation being proposed.

In phase 3 all annotated events are enabled. Notice that the annotated events have also enabled events on their corresponding provided ports (indicated by the "→"). The process of enabling multiple events("correspondence") was demonstrated in figures like Figure 8 by the dotted lines.

6.1.2 Round 2

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	ExS	Ex	E	Proposed	Annotated	Enabled → Arch1 Ex B
P2	Fifo1	n/a	n/a	Suspended	Suspended	Suspended

In phase 1 the “ExS” end event is proposed. This is due to the fact the ExS.Ex.B event was enabled in the previous round and computation takes zero time. The FIFO components are suspended awaiting data. This waiting is not part of the framework explicitly (i.e. a SystemC wait perhaps) but can be detected in order to switch phases.

In phase 2 all proposed events are annotated. The events associated with the suspended process naturally are not annotated.

In phase 3 the mapped architectural component “Arch1” enables its “Ex” begin event. Other processes remain suspended. This illustrates the architecture model following the functional model indicative of this proposal.

6.1.3 Round 3

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Arch1	Ex	E	Proposed	Annotated	Enabled → Source Ex E
P2	Fifo1	n/a	n/a	Suspended	Suspended	Suspended

In phase 1 “Arch1” proposes the end event of the “Ex” operation. Notice this comes in the round immediately after its begin event was enabled much like its functional counterpart.

In phase 2 all proposed events are annotated. Again events associated with suspended processes are not annotated.

In phase 3 “Arch1” and “Source” components enable their corresponding end events.

6.1.4 Round 4

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Source	W	B	Proposed	Annotated	Enabled → Fifo1 W B
P2	Fifo1	n/a	n/a	Suspended	Suspended	Suspended

In phase 1 now that the “source” has executed, it is ready to write its data to the FIFO. It does this by proposing a write begin event.

In phase 2 all proposed events are annotated as in other rounds.

In phase 3, “Source” and “Fifo1” write begin events are enabled. Fifo1.write executes and notifies process “DCT” via the FIFO1 read event.

6.1.5 Round 5

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Fifo1	W	E	Proposed	Annotated	Enabled → Arch1 W B
P2	Fifo1	R	E	Proposed	Annotated	Enabled → Arch2 R B

In phase 1 “Fifo1” write end is proposed. “Fifo1” read end (from the DCT’s initial enabled begin event) also is proposed now that the process is running again due to the presence of data in the FIFO.

In phase 2 all proposed events are annotated including the now proposed P2 events.

In phase 3 both fifo events are enabled. This in turn enables the corresponding events in "Arch1" and "Arch2" for read and write begin events.

6.1.6 Round 6

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Bus	W	B	Proposed	Annotated	Enabled
P2	Bus	R	B	Proposed	Annotated	Enabled

In phase 1 "Arch1" and "Arch2" call Bus.Write and Bus.Read respectively. The "Bus" begin events for these methods are now proposed.

In phase 2 all proposed events are annotated.

In phase 3 "Bus" begin events are enabled. The assumption here is that the bus can support multiple accesses. Notice that the architecture events do not correspond to any other events as they do not interface with other provided ports.

6.1.7 Round 7

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Bus	W	E	Proposed	Annotated	Enabled
P2	Bus	R	E	Proposed	Annotated	Enabled

In phase 1 "Bus" end events are proposed for both write and read.

In phase 2, all proposed events are annotated.

In phase 3 "Bus" end events are enabled. This sufficiently shows the typical operation for this application. This pattern of operation will repeat throughout the various components of MJPEG.

6.2 Proposal #2

Using the alternative execution semantics of proposal # 2, the phase-by-phase execution of the dataflow hand example is shown as follows. We assume the architecture services can provide data-driven communication which supports the dataflow models in the functional description. Otherwise, it is a mismatch between function semantics and architecture semantics, and we consider the mapping as ill-defined.

6.2.1 Round 1

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Source	Ex	B	Proposed	Annotated	Enabled → Arch1 Ex B
P2	DCT	R	B	Proposed	Annotated	Enabled → Arch2 R B

In phase 1 all processes propose events. Since all processes have proposed events, the processes will transition to a suspended state and the simulation phase can switch. This is the same as in proposal # 1.

In phase 2 all proposed events are annotated. The details of the annotations are left out for clarity. Annotation in this proposal works just like in proposal # 1 as well.

In phase 3 all annotated events are enabled. The mapped architecture services now can start (their events are those which are "corresponding"). Notice that the architecture services, as opposed to the provided port functionality, now follows the required port function calls (this is different than proposal # 1).

6.2.2 Round 2

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Arch1	Ex	E	Proposed	Annotated	Enabled → ExS Ex B
P2	Bus	R	B	Proposed	Annotated	Enabled

In phase 1, Arch1's execution "end event" is proposed, which means the architecture service has finished executing (again in zero time). The other architecture components are trying to read from the bus and therefore propose a READ "begin event". This is a provided port in the architecture model.

In phase 2 all proposed events are annotated.

In phase 3, Arch1's execution "end event" is enabled, and the corresponding function method ExS starts. This is the transition from the architecture model back to the functional model. This is another aspect unique to this proposal.

6.2.3 Round 3

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	ExS	Ex	E	Proposed	Annotated	Enabled → Source Ex E
P2	Bus	n/a	n/a	Suspended	Suspended	Suspended

In phase 1, ExS proposes an "end event". And all other other processes are suspended when trying to read data (the data-driven bus architecture model assures this).

In phase 2 all proposed events are annotated. Naturally the event associated with the suspended process is not annotated.

In phase 3, ExS and Source components enable their end events. This process is the end of a proposed port event (ExS Ex) signaling the end of a required port event (Source Ex).

6.2.4 Round 4

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Source	W	B	Proposed	Annotated	Enabled → Arch1 W B
P2	Bus	n/a	n/a	Suspended	Suspended	Suspended

In phase 1, now that the source has performed its execution operation, the source is ready to write its data. It does this by proposing a write "begin event".

In phase 2 all proposed events are annotated. Again the events associated with suspended processes remain un-annotated.

In phase 3, the Source and Arch1 write "begin events" are enabled. This is a transfer from the functional model (required port) to the architectural model.

6.2.5 Round 5

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Bus	W	B	Proposed	Annotated	Enabled
P2	Bus	n/a	n/a	Suspended	Suspended	Suspended

In phase 1 a Bus write "begin event" is proposed. The second process, P2, remains suspended.

In phase 2 all proposed events are annotated. P2 remains suspended and its events un-annotated.

In phase 3 the Bus write "begin event" is enabled.

6.2.6 Round 6

Process	Component	Port	Begin/End	Phase 1	Phase 2	Phase 3
P1	Bus	W	E	Proposed	Annotated	Enabled
P2	Bus	R	E	Proposed	Annotated	Enabled

In phase 1 after P1(Arch1) writes data via its "end event", P2(Arch2) becomes un-suspended (running) and can propose to end reading the data.

In phase 2 all proposed events are annotated. This includes P2's event now that it is running.

In phase 3 both Bus events are enabled. **This implies that the bus has the ability to handle separate read and write requests during the same round.**

6.3 Proposal #3

Under the 3rd proposal, the mapping structure will look like figure 11(b). While this mapping differs from proposal # 1 and # 2, the functional model remains unchanged. The architectural model still contains two arch components and a bus component. All arch components support Ex services. The bus model has multiple provided ports, which provide either write or read services. Notice the new dotted line between the arch and the bus component. This represents that the arch component may communicate with the bus as well based on the execution service model on the architectural side. We assume that the bus model itself contains an arbitrator for concurrent requests. The mapping is now performed between the provided ports and the provided ports. This removes the explicit connection between Arch1 and Arch2 and the Bus in the first two proposals.

In order to make the bus model support FIFO read and write, we assume the following pseudo-steps in read/write service. However, on the functional side, the read/write services are not required to mimic the same protocol as in the architectural ones. This protocol is an example of the fact that proposal # 3 requires a level of service granularity not required of the other proposals in order to synchronize the models.

```
1. grab bus access
2. read fifo status
3. if it can proceed to read/write
     read/write; release bus
   else
     release bus; wait a random number of cycles; go to 1
```

The phase-by-phase execution of the hand example is shown as follows:

6.3.1 Round 1

Process	Component	Port	Begin/End	Phase 1	Phase 2	Phase 3
P1	Source	Ex	B	Proposed	Annotated	Enabled → ExS/Arch1 Ex B
P2	DCT	R	B	Proposed	Annotated	Enabled → FIFO/Bus R B

In phase 1 both processes propose events. Since all processes have proposed events the processes will transition to a suspended state and the simulation phase can switch. This would occur in the previous two proposals as well.

In phase 2 all proposed events are annotated. The details of the annotations are left out for clarity. Again, this is handled in the same way as the other proposals.

In phase 3 not only are the original required port functional side events enabled (Source and DCT) but also the corresponding provided port functions (ExS and FIFO) AND the architectural side services (Arch1 and Bus) are enabled as well. This concurrent enabling of functional and architectural events is a key feature of this proposal.

6.3.2 Round 2

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Arch1/ExS	Ex	E	Proposed	Annotated	Enabled → Source Ex E
P2	Bus	R	grab bus B	Proposed	Annotated	Enabled

In phase 1, Arch1 and ExS propose their “end events”. In addition, the bus begins the process of READing with the “grab bus” protocol proposal.

In phase 2, both processes’ events are annotated.

In phase 3, Arch1 and ExS enable their “end events” and concurrently this leads to the enabling of the functional required port “end event” request (Source Ex E). The bus proposed read event is enabled as well. The concurrent enabling here is unique to proposal # 3.

6.3.3 Round 3

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Source	W	B	Proposed	Annotated	Enabled → FIFO/Bus W B
P2	Bus	R	grab bus E	Proposed	Annotated	Enabled

In phase 1, Source proposes a WRITE “begin event”. P2 begins the second stage of the bus protocol with a “grab bus” end event proposal.

In phase 2, both P1 and P2 have their events annotated.

In phase 3, the enabling of the Source WRITE “begin event” leads to the enabling of both the FIFO and Bus WRITE “begin events”. This is another example of concurrent execution. In addition, the second phase of P2’s bus protocol “end event” is enabled.

6.3.4 Round 4

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	FIFO/Bus	W	E/grab bus B	Proposed	Annotated	Disabled → Source W E
P2	Bus	R	read fifo status B	Proposed	Annotated	Enabled

In phase 1, the FIFO and Bus components propose different types of events. The FIFO proposes the “end event” of WRITE. The Bus however begins to try to write with the “grab bus” begin event. P2 is still in the midst of its protocol which started rounds earlier.

In phase 2, both events related to the Bus operation are annotated. In P1, the FIFO and Source events will not be annotated.

In phase 3, all of P1's events are disabled since the bus is currently being used. P2 however can proceed with the enabling of its event.

6.3.5 Round 5

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	FIFO/Source/Bus	W	n/a	Suspended	Suspended	Suspended
P2	Bus	R	read fifo status E	Proposed	Annotated	Enabled

In phase 1, P1 is suspended awaiting P2 to finish using the bus. P2 proceeds with the proposal of events related to the bus protocol.

In phase 2, P2's event is annotated. The suspended process P1's events cannot be annotated.

In phase 3, P1 remains suspended while P2 has a protocol "end event" enabled.

6.3.6 Round 6

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	FIFO/Source/Bus	W	n/a	Suspended	Suspended	Suspended
P2	Bus	R	release bus B	Proposed	Annotated	Enabled

In phase 1, P2 continues proposing events in the protocol while P1 remains suspended.

In phase 2, events associated with running processes are annotated.

In phase 3, P2's "release bus" begin event is enabled. P1 remains suspended.

6.3.7 Round 7

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	FIFO/Source/Bus	W	n/a	Suspended	Suspended	Suspended
P2	Bus	R	release bus E	Proposed	Annotated	Enabled

In phase 1, P2 proposes the release bus "end event". P1 remains suspended.

In phase 2, the events associated with running processes are annotated.

In phase 3, "release bus" end event is enabled.

6.3.8 Round 8

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	FIFO/Bus	W	E/grab bus B	Proposed	Annotated	Enabled $\rightarrow$ Source W E
P2	Bus	R	wait random cycles B	Proposed	Annotated	Enabled

In phase 1, P1 begins again the process of proposing events (after having been suspended) now that P2 has relinquished the bus.

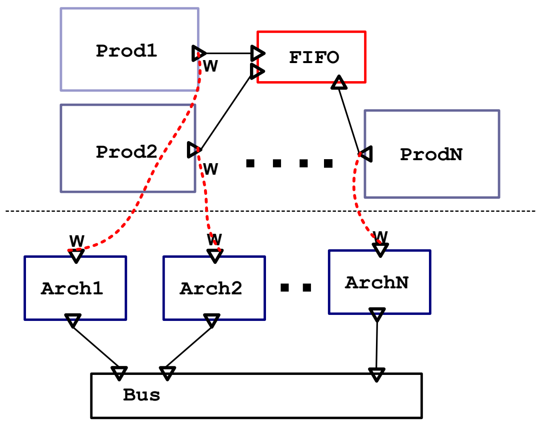
In phase 2, both process can have their events annotated.

In phase 3, all events are enabled. Notice that the enabling of the functional FIFO event and the architectural Bus event in P1 cause the enabling of the functional required port Source WRITE "end event" by implication.

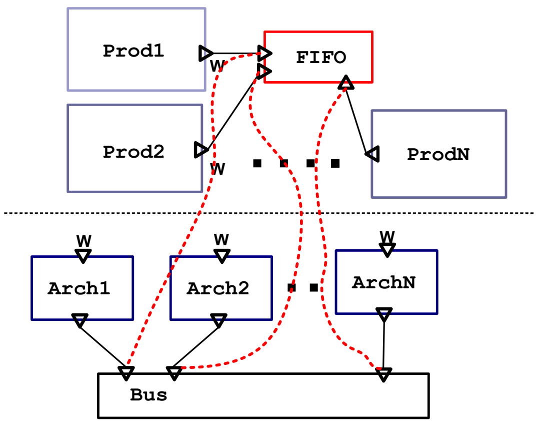
...

7 Execution Traces for the Shared FIFO Example

In this section, execution traces for all three proposals will be evaluated with the Shared FIFO design scenario from Section 4.2. In Figure 12 the required to provided port mapping of proposals # 1 and 2 is shown as well as the provided to provided port mapping of proposal # 3.



(a) Shared FIFO Mapping in Proposals 1 and 2



(b) Shared FIFO Mapping in Proposal 3

Figure 12: Shared FIFO Port Mapping Semantics

7.1 Proposal #1

This section will provide a phase-by-phase execution traces of the Shared FIFO example. To make this design scenario manageable for a hand trace, only two producers will be used. This application starts with both Prod1 and Prod2 processes attempting to write data to the shared FIFO. Each table row shows the process involved, the component name, port, if the associated event is a begin or end event, and its status, through 3 phases of the simulation. The tables are broken up into rounds of running through the different phases, where 1 round is equal to 3 phases.

7.1.1 Round 1

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Prod1	W	B	Proposed	Annotated	Enabled → FIFO
P2	Prod2	W	B	Proposed	Annotated	Disabled (remain annotated)

In phase 1, both processes propose events. Since both processes have proposed events, the processes will transition to a suspended state and the simulation phase can switch. At this point it is not clear which event will be enabled. This will be decided in the 3rd phase.

In phase 2, all proposed events are annotated. The details of the annotations are left out for clarity.

In phase 3, P1's events are enabled. Notice that the previously annotated event (Prod1.W.B) also has enabled an event on its component's corresponding provided port (indicated by the "→"). The process of enabling multiple events was demonstrated in Figure 8 by the dotted lines (these are the "corresponding" events as shown in other hand traces). P2 events remain annotated but are disabled (i.e. stay in the proposed state). The decision of which of the two producer events to enable is made by the constraint solver and would be specified by the designer.

7.1.2 Round 2

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	FIFO	W	E	Proposed	Annotated	Enabled → Arch1 W B
P2	Prod2	W	B	Proposed (old annotation)	Annotated?	Disabled

In phase 1, P1 FIFO.W.E is proposed and P2's events are proposed again as in Round 1.

In phase 2, all proposed events are annotated. P2's events may or may not be re-annotated as indicated by "?".

In phase 3, now that P1 is done with the "functional side" the architecture can execute. This ordering is a key aspect of this proposal. P2 remains disabled. The decision of which event to enable must be specified by the constraint solver.

7.1.3 Round 3

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Bus	W	B	Proposed	Annotated	Enabled
P2	Prod2	W	B	Proposed (old annotation)	Annotated?	Disabled

In phase 1, the Bus proposes a "begin event". This is a provided port service in the architecture model.

In phase 2 all proposed events are annotated. Again P2's events may or may not be re-annotated.

In phase 3, the Bus component enables its "begin event". P2 is disabled again since the first WRITE of P1 has not finished.

7.1.4 Round 4

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Bus	W	E	Proposed	Annotated	Enabled → Arch1/Prod1 W E
P2	Prod2	W	B	Proposed (old annotation)	Annotated?	Disabled

In phase 1, the Bus WRITE "end event" is proposed. Once enabled this will finalize the first write operation.

In phase 2, all proposed events are annotated. Re-annotation is still optional for P2's events.

In phase 3, the architecture bus execution "end event" is enabled. This is a "correspondence" event of multiple events by implication. In this case it also is the "end event" for Prod1 and Arch1 WRITE events.

7.1.5 Round 5

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Prod1	W	B	Proposed	Annotated	Disabled (remain annotated)
P2	Prod2	W	B	Proposed (old annotation)	Annotated?	Enabled $\rightarrow$ FIFO W B

In phase 1, Prod1 now again proposes a WRITE "begin event". This assumes that the application is created in such a way that each producer wants to produce indefinitely.

In phase 2, P1's proposed events are annotated for the first time, and once again P2's events may or may not be re-annotated.

In phase 3, finally P2 is now enabled and a symmetric trace could be given for P2 in future rounds as was shown in the previous rounds for P1. The constraint solver will ultimately determine the order in which events from P1 and P2 are enabled. Currently Prod1 and Prod2's write begin events must be followed by their corresponding end event before the other's begin event can be enabled. B$_{ProdX} \rightarrow B_Y \bigcup E_X, \forall(x = y)$.

7.2 Proposal #2

Using the alternative execution flow in proposal # 2, the phase-by-phase execution traces of the hand example are shown as follows. We assume the architecture service can provide a data-driven communication which supports the dataflow model in function. Otherwise, it is a mismatch between function semantics and architecture semantics, and we consider the mapping as ill-defined. This is the same assumption that is made in the first design scenario.

7.2.1 Round 1

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Prod1	W	B	Proposed	Annotated	Enabled $\rightarrow$ Bus W B
P2	Prod2	W	B	Proposed	Annotated	Disabled

In phase 1, all processes propose events. Both Prod1 and Prod2 both wish to write to the FIFO. Since all processes have proposed events the processes will transition to a suspended state and the simulation phase can switch. This is the same as in proposal # 1.

In phase 2 all proposed events are annotated. The details of the annotations are left out for clarity.

In phase 3, Prod1's write "begin event" is enabled. The corresponding architecture services (Bus.W.B) start. Notice that the scheduler disables P2's Prod2.W.B.

7.2.2 Round 2

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Bus	W	E	Proposed	Annotated	Enabled $\rightarrow$ FIFO W B
P2	Prod2	W	B	Proposed (old annotation)	Annotated?	Disabled

In phase 1, the Bus WRITE "end event" is proposed, which means the architecture write service wishes to finish executing. Prod2 proposes its WRITE "begin event" again in an attempt to access the bus.

In phase 2, all proposed events are annotated. Prod2 may or may not re-annotate its events.

In phase 3, the Bus write "end event" is enabled, and the corresponding function method FIFO WRITE "begin event" is enabled. Again this illustrates the architecture then functionality ordering of this proposal.

7.2.3 Round 3

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	FIFO	W	E	Proposed	Annotated	Enabled → Prod1 W E
P2	Prod2	W	B	Proposed (old annotation)	Annotated?	Disabled

In phase 1, the FIFO using P1 proposes a WRITE "end event". Prod2 once again proposes a WRITE "begin event".

In phase 2, all proposed events are annotated if desired.

In phase 3, FIFO's WRITE "end event" is enabled which in turn leads to the enabling of Prod1's WRITE "end event" (effectively finishing the write). In the subsequent rounds, Prod2 will be enabled instead of Prod1 which will lead to an execution symmetric to the previous 3 rounds.

7.3 Proposal #3

Under the 3rd proposal, the provided ports are mapped. So, on the architecture side, as shown in figure 12(b), one BUS component implements *N* (in our case, two) provided write services. The writing to the architectural BUS corresponds to the writing into the functional FIFO. If there is a bus scheduler of any sort, the two write services will be coordinated upon their activation. The functional model on the other hand remains the same.

The phase-by-phase execution of the hand example is shown as follows. Again the tables are organized as they have been in the previous hand examples. Notice that certain processes are denoted with an apostrophe. These are dummy processes in architectural active components that are activated by triggering events.

7.3.1 Round 1

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Prod1/FIFO	W	B	Proposed	Annotated	Enabled
P2	Prod2/FIFO	W	B	Proposed	Annotated	Disabled
P1',P2'	BUS	W1,W2	B	Proposed	Annotated	Enabled/Disabled

7.3.2 Round 2

Process	**Component**	**Port**	**Begin/End**	**Phase 1**	**Phase 2**	**Phase 3**
P1	Prod1/FIFO	W	E	Proposed	Annotated	Enabled
P2	Prod2/FIFO	W	B	Proposed	Annotated	Disabled
P1'	BUS	W1	E	Proposed	Annotated	Enabled
P2'	BUS	W2	B	Proposed	Annotated	Disabled

7.3.3 Round 3

Process	Component	Port	Begin/End	Phase 1	Phase 2	Phase 3
P1	Prod1/FIFO	W	B	Proposed	Annotated	Disabled
P2	Prod2/FIFO	W	B	Proposed	Annotated	Enabled
P1'	BUS	W1	B	Proposed	Annotated	Disabled
P2'	BUS	W2	B	Proposed	Annotated	Enabled

...

8 Unresolved Issues and FAQ

This section details the unresolved issues brought up in the development of this document and the requirements which must be met by any proposed Metro II execution semantics for mapping in the future.

1. How is non-determinism in both the functional and architectural models handled? Which model influences which? - Functional non-determinism is resolved by the architecture model in proposal 2. Vice versa potentially in proposal 1. We need to distinguish between inter-processor non-determinism (e.g. multiple processes accessing an atomic resource) and intra-processor non-determinism (e.g. a process proposing multiple events to the framework, where at most one can be enabled at a time (aka. the await statement)).

2. Mutual-Exclusion? (e.g. What if there are multiple calls to a FIFO (or bus) read event and they are all waiting on the same event to be enabled?) - Constraints in the constraint solver will disallow events from proceeding.

3. How is state shared between function and architecture models? Should it be formalized? - Shared state may not exist, only compatible behaviors. Architecture and functional models are not expected to share state. This is particularly true in the case of IP import where the IP is a black box. Information can be passed to each model via mappers or in event values.

4. How are we mapping communication pieces? Do we still have the notion of "media" or "passive" components? - A structural vs. behavioral view is more appropriate to think about mapping. Communication is kept separate still but what is communication is implicit not explicit. Passive components still exist to provide a multitude of services without a thread of control. In proposals 1 and 2, the architecture components are actually passive.

5. Atomicity, Mutual Exclusion, Blocking. How are we going to define these? - Blocking and waiting are supported now. Suspending is an explicit process state caused by the proposal of events or waiting. Waiting is outside of the framework's control (i.e. systemC waiting) and is part of the coordination mechanism. Mutual exclusion can be accomplished both with waiting or a forced suspending procedure through the creation of resource access protocols.

6. Do all required ports which are mapped in a functional model need to be connected to provided ports also in the functional model? For example, do we need to connect the MJPEG design example's "Ex" required ports. - All ports need to be connected in the current MetroII implementation. This is largely a syntactic issue and could be relaxed in the future.

7. What is the role of an annotator? Should functional events be annotated? How should the annotation information be used? Can annotators perform dynamic annotation based on the

simulation at runtime? - Annotators are currently in their infancy and these issues are truly unresolved.

9 Glossary

This glossary attempts to define many of the terms used throughout this document in one centralized location. The terms themselves are often defined when they first appear as well.

1. **Architecture** - A collection of services provided to the functionality to implement the behavior specified.
2. **Active** - A Metro II component with a process and a set of required ports.
3. **Call graph** - A diagram which illustrates the timing and causality between method calls in the functional model and service calls in the architectural model.
4. **Connection** - An assignment of a required port to a provided port as specified in a Metro II netlist. This is between components in the same domain (architectural or functional).
5. **Consumer** - An active Metro II component which acts as a reader.
6. **Correspondence** - Specifies which events in the functional model are related to events in the architectural model. The visual illustration of this is in a "call graph".
7. **Execution semantics** - The process by which Metro II simulation proceeds and how Metro II components interact during simulation.
8. **Execution semantics for mapping** - The relationship between the functional and architectural models during simulation in Metro II.
9. **Functionality** - What a design does. This is ideally specified by models of computations with no indication of implementation level issues such as performance or cost.
10. **Hand traces** - This is a round by round written description of a Metro II simulation. Included are the process states and the event states associated with them at each of the three phases of Metro II simulation.
11. **Interface** - A collection of methods or services, accessed via a port.
12. **Mapping** - The assignment of functional behavior to architectural services. This is a many-to-one relationship between functional methods and architectural components.
13. **Mapping structure** - The relationship between ports in the functional model and those in the architecture model. This determines which functional methods will invoke which architectural services.
14. **Method** - Methods are how functionality is invoked in the functional model.
15. **Passive** - A Metro II component without a process and a set of provided ports.
16. **Process** - A single thread of execution which can propose events, perform computation, and access component interfaces.

17. **Producer** - An active Metro II component which acts as a writer.
18. **Read** - A required port method called by active Metro II components to access data; a provided port method implemented by passive Metro II components.
19. **Reader** - A Metro II component which receives data.
20. **Services** - Services are how architectures are invoked. Services are what the architectural model presents to the functional model via interfaces.
21. **Write** - A required port method called by active Metro II components to manipulate data; a provided port method implemented by passive Metro II components.
22. **Writer** - A Metro II component which provides data.

10 Acknowledgements

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

References

[1] Alberto Sangiovanni-Vincentelli. Defining Platform-Based Design. *EEDesign*, February 2002.

[2] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. A Next-Generation Design Framework for Platform-Based Design. In *Design and Verification Conference (DV-CON'07)*, February 2007.