Metropolis Architecture Refinement Styles and Methodology

Douglas Densmore – University of California, Berkeley
densmore@eecs.berkeley.edu
Technical Memorandum UCB/ERL M04/36

November 5, 2004

Contents

Contents **2**

1 Abstract **5**
1.1 Keywords 5

2 Introduction **7**

3 Refinement Theory **9**

4 Refinement Methodology **13**
4.1 General Architecture Structure 14
4.2 Vertical Refinement 15
4.3 Horizontal Refinement 16
4.4 Depth Refinement 17

5 Refinement Properties **19**
5.1 Properties 19
5.2 Property Relationships 21

6 Initial Simple Architecture **23**

7 Specific Vertical Refinement Modifications **25**
7.1 Netlists 25
7.2 Services 28
7.3 Service Access Definitions 30
7.4 Tasks 30
7.5 Quantity Managers 31
7.6 Request Definitions 32

7.7 Support Structures 32
7.8 Vertical Additions to Simple Architecture 32

8 Proposed Horizontal Refinement Modifications 35
8.1 Horizontal Additions to Simple Architecture 35

9 Proposed Hybrid Refinement 37
9.1 Tasks transformed to Media 37

10 Xilinx Architecture Platform 39

11 More Information and Sponsors 41
11.1 Referencing This Document 41
11.2 Sponsors 41

Bibliography 45

One

Abstract

This document will detail both the ideology as well as the techniques involved with the development of architecture models in the Metropolis Design Environment at various abstraction levels. In particular, it is concerned with discussing how these models relate to each other when considered to be various refinements of one another. These issues have both theoretical as well as implementation issues when considering designing in the Metropolis Design Environment. The document with describe the concepts regarding the refinement of architectures and then go on to detail the development on an example architecture included with the distribution of Metropolis.

1.1 Keywords

Refinement, Vertical, Horizontal, Depth, Hybrid, simple architecture model, MicroProperties, MacroProperties, functional refinement, structural refinement, abstraction

Two

Introduction

This document serves several purposes. First of all, it introduces the reader to the initial Metropolis ideas and concepts regarding refinement development and verification of architecture descriptions in Metropolis. While this is in the context of the Metropolis Meta Model (MMM), many ideas extend outside of this framework and can be examined by the reader separately. Secondly, it serves as a guide to how the simple architecture models were refined (resulting in one completed model example and two proposed models). This description serves as a ”step-by-step” guide in which the model design process is documented and the design decisions explained. This version of this document is intended to be a living document that evolves as the Metropolis Team gathers feedback and the investigation into these topics change. Many of the ideas contained here are preliminary and have not been fully investigated. However, this is intended to provide not only a guide to how the example architecture was created but to also set forth interesting questions for the developer to explore with Metropolis.

The primary goals for architecture modeling and subsequent refinement that the Metropolis Team are trying to reach are to (1) determine how we can characterize the refinement of Metropolis architecture models (2) how we can carry out this refinement automatically (or have a very exact standardized methodology) (3) how to formally verify that one model is a refinement of another model (4) determine how to relate refinement across various architecture types and properties at various levels of abstraction.

Point #1 is important so that we have a vocabulary in which to describe not only what a refinement is but also to understand what properties a refinement should preserve (and hence what the design objective is). The second point naturally is something that would be ideal from a tool and efficiency standpoint. Efficiency meaning that it is computationally tractable and expressible in the MMM. The third point really indicates that we must understand what properties of a model we are looking at and specifically how to check those properties efficiently. The final point underscores the need to explore refinement inside of Metropolis.

The experimentation to be described is regarding how one might refine the current simple architecture developed by the Metropolis team and located in *examples/pip/architecture/simple*. This was initially constructed for the PiP exercise (an initial design proposed and implemented in the Metropolis environment). This has resulted in one example model that will be described in detail and two that are theoretically described. What is in progress is how to (1) characterize how we ″prove″ this is a refinement (2) how we can quantify that this refinement is different from the original models. Hopefully the discussion that follows can be useful in trying to get at these issues.

*Note on the terminology: Simple architecture refers to the simple architecture provided with the release.

Three

Refinement Theory

Refinement from a "System Level Design" standpoint has many different definitions and each can dramatically change the interpretation of what it means to refine an architecture model. The basic definition for the purposes of this document and Metropolis development is:

Definition: Refined Architecture Model - a model which can be substituted into a system (read: Metropolis Netlist) in place of another model while maintaining the correct overall functionality of the system while differentiating itself from the original via a communication, computation, or coordination property.

The main challenge with this definition is how do you measure the "communication, computation, or coordination" property? In addition there is the difficulty of saying under which environments (external stimulus) can the model be substituted? Are restrictions necessary to the environment or can it be in any environment? Naturally, this all assumes that minimally the correct behavior of the system is maintained upon the refinement.

Initially there are two basic divisions of refinement verification:

- Provide properties and check that those properties hold between two models.
  - The user would supply a set of models and a set of properties encapsulated either in the model descriptions themselves or as a separate specification.

- Check two models and tell which properties hold between them.
  - The user would supply a set of models and the result would be a set of properties which are satisfied (or not satisfied).

The first method requires that one know which properties one would like to check. This is problematic since you must not only specify these properties but also you must make sure not to ″miss″ any that would lead to incorrect information regarding the refinement relationship. In addition, you must be ″sure″ that these are relevant properties. What is relevant to one model might not be to the other and what is relevant in one application might not be in the other.

The second method requires that ″all″ properties be checked. This is difficult naturally from a feasibility standpoint. Not only is the number of properties large (presumably infinite) but also the definition of a property must be automatically decided.

Refinement checking should be a combination of these two approaches. This means that the verification process can accommodate both the specification of properties as well as the checking of implicit properties relevant to that model/application. This could be achieved by defining a two stage process.

Two Stage Process of Refinement

- Group Properties together in such a way that the establishment of one property ensures the adherence of the others.
  - The initial property that is checked is referred to as the **MacroProperty**. The other properties implied are **MicroProperties**.
  - The relationship between a MacroProperty and its MicroProperties is a *Partial Order*. It is reflexive, antisymmetric, and transitive.
- Develop refinement methodologies such that if a model is created in a certain manner then the MacroProperty is preserved.
  - Depending on which MacroProperty you want to preserve you do refinement in a specific manner. This leads to the need for a specific refinement methodology!
  - This is *correct by construction* and provides insight into the relationship between various MacroProperties.

Methods of refinement as defined by [4] are trace based, execution based, and property based. The level of property specification can be LTL, CTL, Timed LTL, LOC, etc (these are all logics that are used in formal verification). What is important is that the Macro and Micro properties can be expressed in the same syntax.

With MacroProperties that cannot be preserved by construction methods, checking this property is still much smaller than checking all its associated MicroProperties. Having relatively easy to verify MacroProperties is key to this methodology.

Another issue is that of Structural vs. Functional refinement. This means refinements that hold particular functional properties (data consistency) and refinements that hold particular structural properties (FIFO size). Depending on your goals, one or the other (both) might be important. Naturally, there are many obvious cases where you want functional refinement such as in protocol development. The key is to determine what aspects of your architecture are concerned with functional refinement. It is not a given that all functional aspects of a system must be preserved in refinement. Structural refinement likewise may be important in such areas where a design is known to need certain structures (memory requirements). Again, structural refinement does not necessarily have to hold across the entire design.

Four

Refinement Methodology

In order to systematically refine a model there must be a methodology in place that demonstrates the procedure for a designer to follow in order to perform various refinements. In following such a procedure, ideally one can say *a priori* which of the properties will hold between the abstract and refined model. Naturally, this leads to the fact that one can follow various procedures depending on which properties are of interest. In addition, knowing which properties should be checked prunes the space of all properties making the automatic checking of such properties feasible.

The initial refinement procedures to be described are termed *vertical* or *horizontal* refinement. Horizontal refinement refers to the fact that the goal of the refinement is to move architectural aspects of the model from the ”scheduling netlist” into the ”scheduled netlist”. A key will be to **identify which properties can be preserved by horizontal refinement**. Vertical refinement refers to the process of transforming items in the ”scheduled netlist”. This typically is done by targeting one particular process or media element and decomposing it into multiple media and process elements and then replacing that decomposed structure back into the model. Again, a key will be to **identify which properties can be preserved by vertical refinement**. Naturally, a *hybrid* approach can be taken in which ”vertical” and ”horizontal” aspects of refinement can be combined. Finally, there is a third ”axis of refinement” termed *depth* refinement. This is where a single process or media is changed internally (i.e. a scheduling algorithm changes or a media service is modified).

An interesting area is to identify how the mapping process in Metropo-

lis plays into refinement. Does the method by which you refine make for easier or harder mappings? More efficient mappings? Better performance mappings? Does it make a difference? For example, if one were to perform a refinement by which they change the scheduling algorithm for a task using a resource (*depth refinement*), an intelligent mapping may now need to take into account that a "better" mapping may occur by mapping certain processes which could better use that scheduling algorithm to that resource and mapping other process which don't benefit from that scheduling to another resource. If all scheduling algorithms were the same, the mapping could be "dumber" and arbitrarily assign processes. Mapping is described in the document [2].

Staring with section 6, we will give "step by step" instructions on how "vertical refinement" was done in the simple architecture along with how this can be done generically for an arbitrary model.

4.1 General Architecture Structure

In order to present a standardized structure of an architecture we propose the following structure for Metropolis architecture model structure which is based upon the original simple architecture structure at

examples/pip/architecture/simple. The example files mentioned are from that original simple architecture model.

An architecture description in Metropolis should consist of:

- Top level netlist - this instantiates the architecture; Top.mmm
- Architecture level netlist - this instantiates and connects the scheduling and scheduled netlist; Architecture.mmm
- Scheduling Netlist - defines which components are in the scheduling netlist; ArchSchedulingNetlist.mmm
- Scheduled Netlist - defines which components are in the scheduled netlist; ArchScheduledNetlist.mmm
- Request Definitions - defines the request structure for services. These structures are how requests are made to the Quantity Managers during the *request phase* of architecture execution; InterfaceSchedReq.mmm

- Services - these are items the provide services, CPU, BUS, etc; Mem.mmm CpuRtos.mmm Bus.mmm
- Quantity Managers (Schedulers) - this can be as few as global time or as many as one scheduler per service; MyScheduler.mmm SchedulerFIFO.mmm SchedulerTimeSliceBased.mmm
- Tasks - processes which utilize the resources; SwTask.mmm
- Service Access Definitions - provides the prototypes for the service interfaces and defines ports. This dictates what the high level API for the services in the architecture is; InterfaceScheduling.mmm InterfaceScheduled.mmm
- Support Structures - code which provides data structures needed for services; ProcessAccount.mmm ProcessRecord.mmm

Based on this structure we can talk generically about what the different refinement styles would consist of. All architecture should have these elements. What refinement will do is change the number and the connection between them. For example, a vertical refinement will add services and quantity managers. This will also lead to more service access definitions. Horizontal refinement will reduce the Quantity Managers but add services in the scheduling netlist.

4.2 Vertical Refinement

Vertical refinement is the notion that within the model changes are made ”vertically” where these changes are additions/subtractions/divisions of media. This will consist of topological changes to existing media as well. This means that you do not swap aspects/relationships/devices between netlists but rather you move within a particular netlist. Naturally, this contrasts to horizontal refinement.

Vertical refinement of an architecture can be seen as a whole spectrum of refinement with the levels being defined as to what elements are passive (media) and which are active (processes). For example, you can change the number and types of processes in the scheduled netlist or you can change the number and type of media in the scheduled netlist.

What this would imply is that the most abstract would only have the clock be the only active element while the least abstract would have all processes (vice versa depending on perspective).

The primary method of vertical refinement in practice is likely to be the addition of media. This style of refinement ultimately is the addition of **services**. This is adding a level of granularity to the abstract services provided initially. Vertical refinement is most likely the most common form of refinement from a structural standpoint concerning the netlists. This is also the most straightforward of the refinement styles. This will require the following types of changes:

- Need to add/create services themselves
- Need to add requests for each services
- Need to add ”schedulers” for these services. There is a one-to-one correspondence between the services and schedulers.
- Need to introduce these into the corresponding netlists

Notice that with a vertical refinement you are moving vertically in both netlists. For example, the addition of a service in the scheduled netlist requires an additional scheduler in the scheduling netlist.

4.3 Horizontal Refinement

Horizontal refinement is the moving of the scheduler’s (quantity manager’s) functionality into the scheduled netlist.

The spectrum of horizontal refinement really simply results by how many of the schedulers you move and what portion of the schedulers you move. You many not move all functionality but rather only certain aspects if you so desire. It is that freedom which provides the abstraction levels in this refinement style. Horizontal refinement is done to represent in the ”scheduled netlist” the components that will physically need to be present in an implementation. It is to make the design more ”tangible” and less reliant on simulation-based components. Horizontal refinement also changes the speed and semantics of the simulation.

This would require the following changes:

- Need to modify interfaces from services to scheduling netlist to reflect their new relationship with the migrated new services.
- Need to add the functionality to the services which now provide the previous scheduling duties.

Notice that horizontal refinement is less systematic in some sense when compared to vertical refinement since potentially there are more interactions that must be explored. Horizontal refinement will change the simulation semantics significantly since the access to services now are handled by the method created by the user as opposed to the semantics of the defined quantity managers.

4.4 Depth Refinement

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

The *Control Flow Automata (CFA)* backend currently available (by request; not a part of the initial release) for Metropolis could be viewed as depth refinement checking. When you change a process you can check trace containment on the function calls to media made by the process as compared to its abstract counterpart. This containment can give you an indication as to whether the internal change will affect the overall functionality of the system.

The CFA backend is available for use as a backend in Metropolis. It is very limited and requires the use of an external set of tools including Intel’s FORTE environment and the University of California, Berkeley’s Mocha [1]. Currently the CFA backend has had several issues pointed out as potential limitations by the Metropolis team.

- No notion of fairness or liveness
  - Empty Trace is always a refinement

- How to handle non-determinism
- Potential state explosion
- Witness module required by Mocha not automatically generated (perhaps not possible)

These are listed so that the key issues are documented. We will look to address these issues (and others as they arise) in a future revision of this document.

An initial attempt to use this backend for single threaded processes in Metropolis is in [3] at DATE04. This paper talks about the refinement process performed with researchers at Cypress Semiconductor. It details what the refined models looked like as well as how depth refinement on individual processes was verified at each refinement stage.

Five

Refinement Properties

The key issue for refinement is resolving what are the properties that are required to hold between the abstract model and the refined model. The first question is how do those properties manifest themselves as attributes of a model? For example if one is interested in the resulting latency of a process, what are the observable behaviors of the process that give me insight into this property? The second question is how are those attributes to be related between the two models? We introduce this definition of a property for the purposes of the following discussion.

Definition: MicroProperty - the combination of one or more attributes (quantities, properties) and a relation defined on these attributes.

Definition: MacroProperty - a property that encompasses a collection of MicroProperties. The satisfaction (i.e. the property holds or is true) of the MacroProperty ensures all MicroProperties covered by this MacroProperty are also satisfied. Since the implication does not commute between Micro and MacroProperties there are MacroProperties which share MicroProperties but they are themselves not the same nor do they imply the satisfaction of one another.

5.1 Properties

This section is where we will begin to talk about which properties (which make up MicroProperties) can be discussed during refinement. Right now the list is very sparse and lacks details but gives an intuition as to

what could be expected in this area. The majority of effort will now go into identifying these properties, their relationships, and how to check them.

You can categorize the properties as structural, functional, and performance. Notice that it is important to understand that in the tradition of *platform based design* and its *orthogonalization of concerns*, that these categories be explicitly distinct.

Examples of performance properties:

- Latency - time for a task to complete
- Throughput - number of tasks completed per unit time
- CPI - cycles per instruction (request)
- Jitter - random variation in a signal

Performance properties typically have to do with specifications regarding the desire for a certain level of performance. However sometimes these properties can actually be required for the correctness of a system. Many performance properties are related to one another.

Examples of functional properties:

- Mutex - mutual exclusion of a resource
- Data Consistency - data in equals data out

Functional properties typically have to do with the correctness of a system. Often they affect the performance properties of a system as well.

Examples of structural properties:

- Memory Size - size of memory elements such as FIFOs
- ALU operand size - the size of the ALU operands (i.e. bits)
- Datapath width - the size of the instruction

Structural properties typically have to do with both the performance and the correctness of a system. They will interact with other structural properties as well as with functional and performance properties. These properties often pertain more to the implementation issues associated

with the design of a system and therefore typically appear at lower abstraction levels.

What will be of key importance is the way in which properties are related and categorized so that we can:

- Determine which properties are related and how
- Determine which refinements related to which properties

In terms of grouping properties and initial attempt used is in [4] which has a method that uses the following terminology:

- Rule of Computation (CMP)
- Rule of Read Order (RO)
- Rule of Write Order (WO)
- Rule of Write Atomicity (WA)

You can group properties by this method. For example, the ”mutex” functional property is a WA property. While Data consistency is a RO and WO property. Perhaps this method can be used to examine Macro and Micro Properties relationships.

5.2 Property Relationships

Here are how property relationships can be established which is key to the Micro and Macro properties discussed earlier. This is an example. Future work will be devoted to establishing these relationships.

Data Consistency $\rightarrow$ Data Storage Space Sufficient (FIFO Size), Read Access, Write Access

In this case if the *MacroProperty* ”Data Consistency” is shown it then implies the *MicroProperties* Data Store Space Sufficient, Read Access, and Write Access. Notice that the MicroProperty ReadAccess does not imply anything at this point.

The keys to these relationships are (1) There must be a method to prove the relationship (2) The MacroProperties cannot be more expensive to check then the sum of MicroProperty checking costs (3) The MicroProperties must be non-trivial.

Six

Initial Simple Architecture

Figure 6.1 shows the standard simple architecture. This is discussed in the ”A Simple Case Study in Metropolis” document found in this release [2]. Please refer to this document as the following discussion assumes that the reader is familiar with the concepts and terminology contained there.

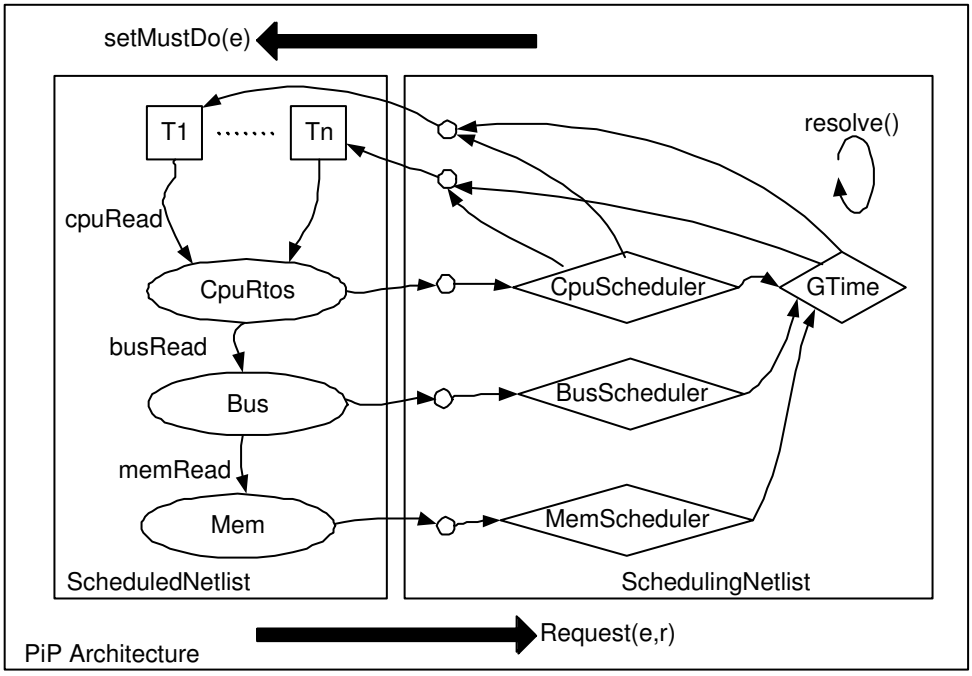

Figure 6.1: Initial Simple Architecture Model

Seven

Specific Vertical Refinement Modifications

This refinement will start with the files that compose the ″simple″ architecture. Having an understanding of this architecture is the foundation for the refinement. This structure will aid in process of documenting how the refinement will evolve from that model. The following sections detail what will be changed in the current architecture model for a vertical refinement as shown in figure 7.1 and provided in *examples/pip/architecture/simple*.

7.1 Netlists

A vertical refinement requires many changes to be done to the netlists. Primarily services need to be introduced to the scheduled netlist and their quantity managers need to be put into the scheduling netlist. This requires that the same structures which instantiate and parameterize the existing services need to be duplicated for the new services. In addition, the connections need to be changed to reflect the new topology.

Top.mmm

No major changes required. May simply change the name of the architecture netlist to reflect that this is a refinement of the original. In this case, Top.mmm became **top_vert.mmm**. Here are the steps that required:

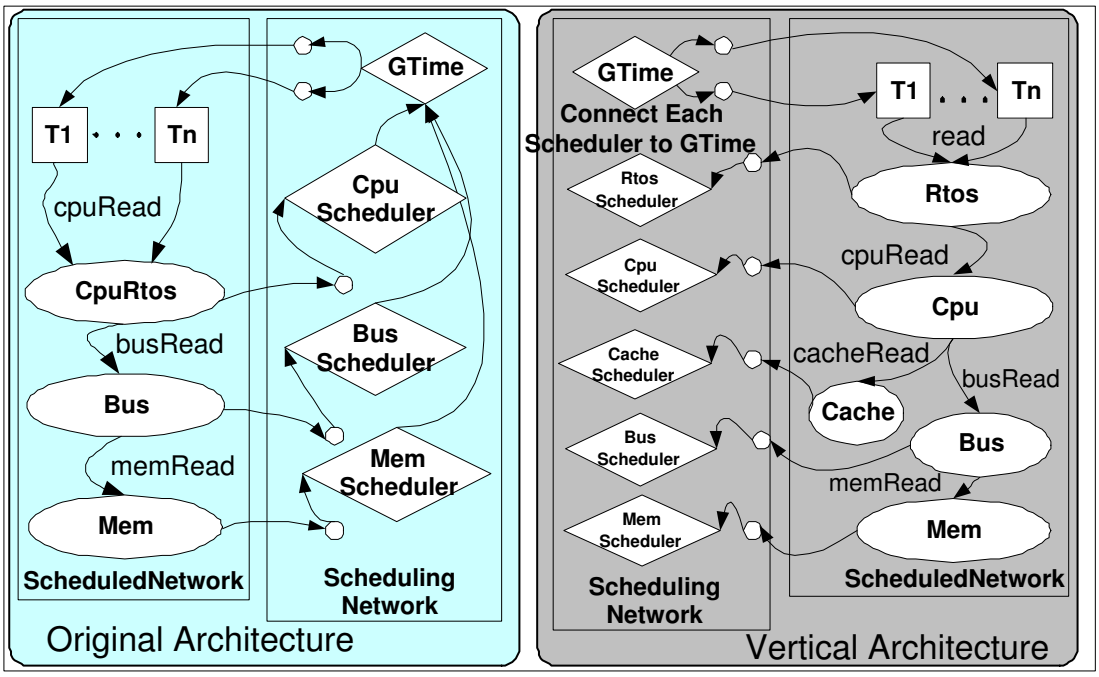

Figure 7.1: Vertical Refined Architecture Model

1. Need to add parameters for whatever new services you are adding.
    - This should represent some aspect of the service that will effect the simulation.
2. Need to introduce those parameters into the instantiation of the Architecture
    - Make sure to change the constructor for the architecture.
3. Optional: Change the names of the instances if desired.
    - Useful if you view the elaborated netlist.

Essentially for each service which is currently found one would need to add duplicate parameter structures for each new service that will be introduced.

Architecture.mmm

Changes here are also minimal (perhaps only name changes to netlists). Naturally instantiations of the scheduling and scheduled netlist will occur. Architecture.mmm became **architecture_vert.mmm**.

This is the netlist that brings the scheduling and scheduled Netlist together.

1. Change the number of quantities to reflect the new value.
    - This will be one for each new service.
2. This requires the introduction of constructor to reflect top level netlist call.
    - This is defined in the top-level netlist.
3. Fix the constructors to pass the values to the scheduling and scheduled netlists
4. Majority of work entails using the same structure in place to add the other service(s). Just add the same structure for each new object in terms of variables and connections.
5. Optional - change the Netlist name arguments.

ArchSchedulingNetlist.mmm

This netlist needs to introduce the new items for the entire scheduling netlist. In a vertical refinement, this will be new schedulers for services. This became **piparchscheduling_vert.mmm**.

This is the scheduling netlist. Since this is a vertical refinement there is no swapping between the two netlists, just the addition of schedulers for each service you added in the scheduled netlist.

1. Add schedulers for each service
2. Update parameters (Quantity count and TAGs)
3. Change constructor if need be
4. Instantiate the schedulers
5. Add components and make connections

ArchScheduledNetlist.mmm

This now will include a RTOS media and cache media. In addition, it needs to reorganize the connections appropriately. This file is now **piparchscheduled_vert.mmm**.

Since this is a vertical refinement there is not an exchange to this netlist from the scheduling netlist. All that is necessary is to add the new services.

1. Provide necessary parameters for new services.
    - This involves changing the constructor for netlist.
2. Instantiate new services.
3. Add the services into the netlist.
4. Make necessary connections.
    - This will involve not only the addition of new connections but also changing existing ones. *In this case the tasks are now connected to the RTOS instead of the CPU.

7.2 Services

The nice thing about a vertical refinement is that ideally many of the services can remain unchanged. Only those interacting with the new services will need to be modified. The main effort in this area is simply in creating the new services themselves.

CPURtos.mmm

This will now become just a CPU media. The functions it provides should be the same. The CPU does not have to resolve tasks requests. It will be given a request and simply provide services for that request. Tasks will not be able to access the CPU until given permission by the RTOS. The main change is now the CPU is simply set to use the FIFO scheduling policy. The RTOS will select between the FIFO and the TimeSlice based scheduling policies. This file now is **Cpu.mmm**.

This is the CPU from the Simple Arch model with the following changes:

1. Need to change the ports to accommodate any new services (Cache).

2. Need to make sure the interface is defined.

    - In this case cacheInterface.

3. Need to add any needed functionality for the new service that it interacts with.

    - In this case it needs to determine if service calls should be made to the Memory or Cache port depending on a cache hit or miss.

4. Need to change what service it implements (i.e. now it is an RtosSlave).

Mem.mmm

An excellent example of a service not affected by vertical refinement since it does not interact with a new service. This file is the same as the simple architecture's, **Mem.mmm**.

This lack of interaction leads to an easier vertical refinement. Observing this it might be advantageous to keep interactions between services minimal. This demonstrates that a clean, modular design will make the refinement process much smoother. It also points out a potential check for automation. A tool could simply check the connection structure to determine whether a service needs to be modified.

Naturally, some modifications for debugging such as blackbox statements might be useful but can be trivially added.

Bus.mmm

This file will stay the same as before as **Bus.mmm**. This is because of the fact that in vertical refinement the only services that have to change are those that interact with the newly introduced services. The bus just interacts with the CPU and Mem as it did before.

7.3 Service Access Definitions

These typically just have to change to accommodate the new services. Ideally the interfaces will stay the same for reasons such as interface checking and compatibility. The interfaces often are a way in which you can relate on abstraction level model to another level.

InterfaceSched.mmm

This file is still named **InterfaceSched.mmm**. Changes here result usually from a set of interfaces for each newly added service. In this case, it has the following changes:

1. Adds the ”cacheInterface” interface.
2. Adds the ”RtosSlave” interface.

InterfaceScheduling.mmm

This file is still named **InterfaceScheduling.mmm**. Nothing is changed in this file and if the architecture structure remains as in these examples then nothing should change.

7.4 Tasks

SwTask.mmm

These should now only have ports to the RTOS and the statemedia that communicate to the schedulers. The functions that these tasks can perform should be unchanged (other than the fact they reflect the new name change). This file keeps the same name **SwTask.mmm**, to reflect the ideology that SwTasks should not be concerned with what service handles their requests.

1. SwTasks are the same regardless of architecture type (vertical, etc). It is the responsibility of the designer to implement a service that provides a SwTaskService interface consistent with the one shown in this example.

2. Need to modify the port to reflect what service the task can interact with (in this case, it stays the same but the RTOS now provides the SwTaskService interface)

3. Optional - Added a simulate flag to switch which thread runs; i.e. mapping(0) or simulation (1)

The changes in this file really reflect more a change in thought process which demonstrate the fact that SWtasks should not really specify what they are running on (other than through port connections). Changes to SwTask.mmm need to be closely coordinated with the mapping effort since they will be mapping processes during the mapping phase of the design.

7.5 Quantity Managers

In a vertical refinement, a key property is that for each new service added to the ”scheduled netlist” there is a corresponding quantity manager added in the ”scheduling netlist”. However due to the generic nature of the schedulers in the simple architecture example, these do not have to change at all for the vertical refinement.

MyScheduler.mmm

This stays untouched and keeps the same name, **MyScheduler.mmm**.

SchedulerFIFO.mmm

This stays untouched and keeps the same name, **SchedulerFIFO.mmm**.

SchedulerTimeSliceBased.mmm

This stays untouched and keeps the same name, **SchedulerTimeSlice-Based.mmm**.

7.6 Request Definitions

These are requests for scheduling. These should be consistent across refinement since the schedulers are not changing.

InterfaceSchedReq.mmm

This stays untouched and keeps the same name, **InterfaceSchedReq.mmm**.

7.7 Support Structures

These structures are in place in order to facilitate some operation in the original model. Unless you remove the component that uses these structures then you will simply keep them. If you do remove the component then you will also no longer need the support structures. This is always true in a vertical refinement. If this is not true, this may fall under the category of a depth refinement.

ProcessRecord.mmm

This stays untouched and keeps the same name, **ProcessRecord.mmm**.

ProcessAccount.mmm

This stays untouched and keeps the same name, **InterfaceSchedReq.mmm**.

7.8 Vertical Additions to Simple Architecture

This section will detail the new files that I added.

Rtos.mmm

This is a new service that is now inserted between the tasks and the CPU. It now calls the CPU for the tasks. This would allow for multiple CPU services as well as new scheduling mechanisms that can utilize the services more efficiently. It represents the addition of an RTOS service. This is simply an intermediate media that dispatches tasks to the

CPU. A ”depth refinement” would be to add to the RTOS scheduler the ability to change the order in which the task go to the CPU. Basically it just relies on the same structure as the CPU used to have but has the notion of an RTOS overhead parameter. This is a new service not present in the original model. This looks like a CPU essentially but 1) It has both the Time Slice Scheduling option and the FCFS FIFO scheduling option 2) It does not have cache ports or the calls to caches 3) it has a different service cycle interface and parameter settings.

1. Whatever is the ”top” service must interface with tasks and have this interface. In this case, the ”top” service is this RTOS.
2. Need to add a new port to access the CPU.
3. Change the constructor.
4. Change parameters as necessary.
5. Add to the scheduled netlist.

Cache.mmm

This Cache model is based on the caches provided with the processor model examples in Metropolis. The cache instantiated is an Associative Cache set to be two-way set associative, with a block size of 32 bits, and a miss penalty of 10 cycles.

1. Need to make an interface definition for this service.
2. Need to add the appropriate ports
3. Need to add the appropriate parameters
4. Need to add this to the scheduled netlist

Eight

Proposed Horizontal Refinement Modifications

This refinement will start with the files that we have created as the simple architecture as the foundation for the refinement. This structure will aid in process of documenting how the refinement will evolve from that model. The following sections detail what will be changed to the current architecture model for a horizontal refinement as shown in figure 8.1.

8.1 Horizontal Additions to Simple Architecture

Here are the steps that would need to transpire for a horizontal architecture to be created from an existing architecture:

1. New services would have to be created for the scheduled netlist. There would be one new service for each scheduler object that you want to migrate.

2. Would need to introduce those services into the ”scheduled netlist”.

3. Would need to change the connections from those new services into the ”scheduling netlist”.

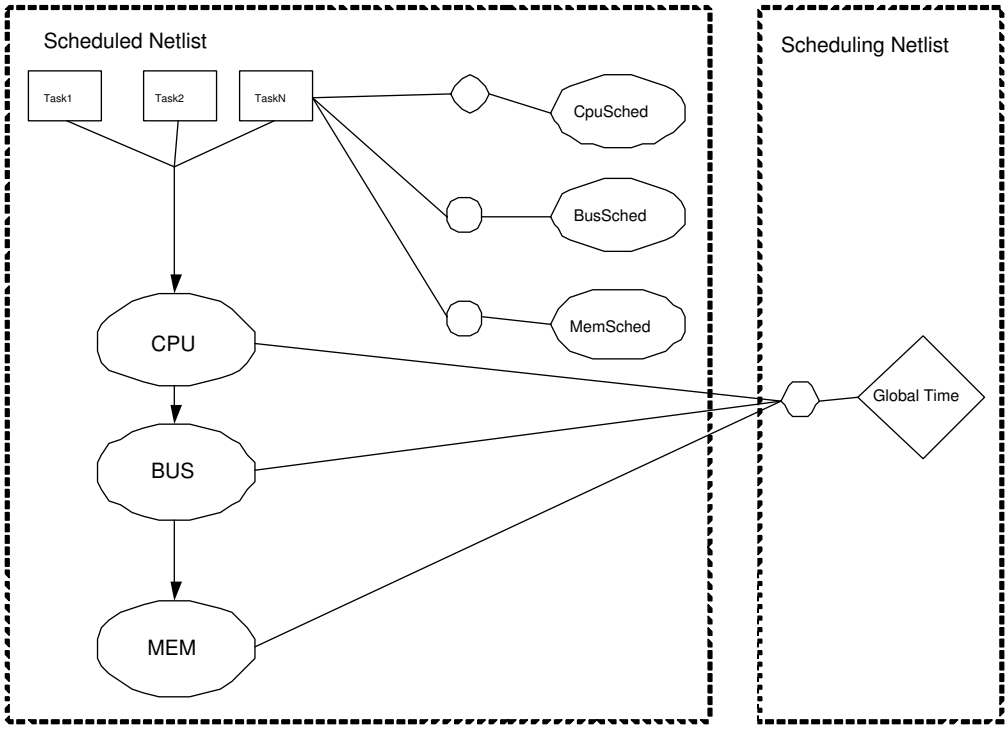

Figure 8.1: Horizontal Refined Architecture Model

Nine

Proposed Hybrid Refinement

Figure 9.1 shows the proposed hybrid refinement. The main features to note are:

- The RTOS is now a process. It uses media to provide services to perform its functions. Tasks must go through the RTOS in order to gain access to various hardware resources. It should be noted that the RTOS is still a ”task” in itself which runs on the CPU.
- The quantities from the previous model now are media in the scheduled netlist. These are the services the RTOS can access to provide tasks access to the ”hardware” resources.

9.1 Tasks transformed to Media

Currently the tasks are processes. One future approach may be to make tasks media. Some hierarchy would be created always with a process as the ”top” of the hierarchy. This process now makes calls to media lower on the hierarchy, which make calls to media lower on the hierarchy etc. This would build a kind of library of media that can compose an application.

This is similar to how the hybrid refinement deals with the RTOS. The RTOS is a process which calls ”tasks”. The RTOS is the top process in the hierarchy and naturally the lower media functions can create a hierarchy that builds functionality that is more complex.

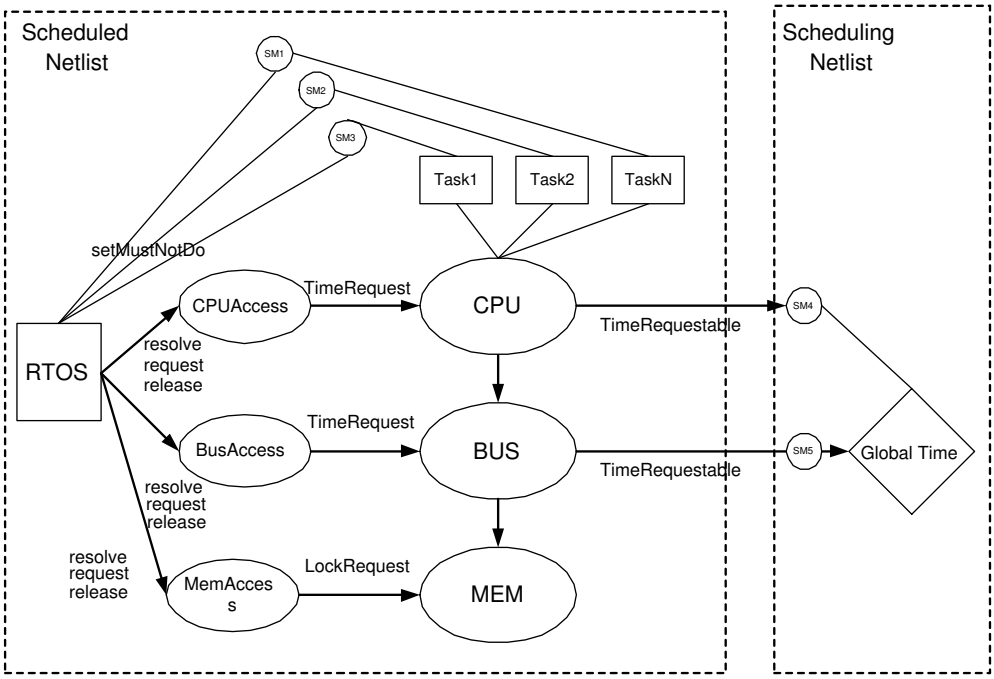

Figure 9.1: Hybrid Refined Architecture Model

Ten

Xilinx Architecture Platform

Ultimately, it would be nice to have a set of "real" architectures that can be targets of the refinement and mapping process in Metropolis. One architecture family which we have access to and is ideal (due to its FPGA and embedded processor design) is the Xilinx Virtex II Pro. We have the part available to use in our lab at Berkeley as well as access to the tools to program it. Work is underway which models key parts of this device along with peripherals to create and embedded system with it.

Also of interest is how a refined architecture could be made of the Xilinx platform. A coarse approach is for IP blocks to be modeled as media that provide various tasks. Applications could be designed in the previously mentioned way by calling functions on those media. Here are some rough initial correspondences:

- CPU $\rightarrow$ PowerPC - could create a PowerPC like service media
- BUS $\rightarrow$ CoreConnectBus - bus like services
- MEM $\rightarrow$ IP Memory Blocks
- Custom Functions $\rightarrow$ Hierarchy of FPGA CLB blocks

Currently there are rough models of the following components in Metropolis: PowerPC405, MicroBlaze SoftProcessor, CoreConnect Processor Local Bus (PLB) and On-Chip Peripheral Bus (OPB), PLB/OPB bridges, BlockRam Memory, and Synthetic Master and Slave devices that can replicate FPGA hardware. These are not part of the release and are

still in their infancy. Information regarding them can be made available upon request.

Eleven

More Information and Sponsors

In order to begin to consolidate the information on refinement this document begins to cite papers that we feel address this topic. The bibliography here will continue to grow and is by no means complete. This should serve as a reference for those interested in refinement. If you have papers to add to this list please let us know at densmore@eecs.berkeley.edu. This address should be used to further direction questions and comments regarding this document's structure or content. Finally see [5] for other concepts regarding these topics.

11.1 Referencing This Document

Here is an example of how to reference this document:

```
Douglas Densmore, "Metropolis Architecture Refinement Styles and Methodology",
Technical Memorandum UCB/ERL M04/36, University of California, Berkeley, CA 94720,
September 14, 2004.
```

11.2 Sponsors

This work was supported in part by the following corporations:

- Cadence
- General Motors
- Intel

- Semiconductor Research Corporation (SRC)
- Sony
- STMicroelectronics
- and the following research projects:
  - NSF Award Number CCR-0225610 and the Center for Hybrid and Embedded Systems (CHESS, http://chess.eecs.berkeley.edu)
  - The MARCO/DARPA Gigascale Systems Research Center (GSRC, http://www.gigascale.org)

The Metropolis project would also like to acknowledge the research contributions by:

- The Project for Advanced Research of Architecture and Design of Electronic Systems (PARADES, http://www.parades.rm.cnr.it/) (in particular Alberto Ferrari)
- Politecnico di Torino
- Carnegie Mellon University
- University of California, Los Angeles
- University of California, Riverside
- Politecnico di Milano
- University. of Rome
- La Sapienza
- University of L'Aquila
- University of Ancona
- Scuola di Sant'Anna and University of Pisa

src/com/JLex JLex has a copyright that is similar to the Metropolis copyright.

src/metropolis/metamodel Portions of the Java code were derived from sources developed under the auspices of the Titanium project, under funding from the DARPA, DoE, and Army Research Office. The Java code was further developed as part of the Ptolemy project. The Java code is released under Metropolis copyright.

src/metropolis/metamodel/frontend/Lexer Portions of JLexer are: "Copyright (C) 1995, 1997 by Paul N. Hilfinger. All rights reserved. Portions of this code were derived from sources developed under the auspices of the Titanium project, under funding from the DARPA, DoE, and Army Research Office."

src/metropolis/metamodel/frontend/parser/ptbyacc ptbyacc is in the public domain.

Bibliography

[1] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In *Computer Aided Verification*, pages 521–525, 1998.

[2] Abhijit Davare, Douglas Densmore, Vishal Shah, and Haibo Zeng. A simple case study in metropolis. Technical Memorandum UCB/ERL M04/37, Univerity of California, Berkeley, CA 94720, September 2004.

[3] Doug Densmore, Sanjay Rekhi, and Alberto Sangiovanni-Vincentelli. Microarchitecture development via metropolis successive platform refinement. In *Design Automation and Test Europe*, 2004.

[4] Ratan Nalumasu, Rajnish Ghughal, Abdelillah Mokkedem, and Ganesh Gopalakrishnan. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In *Computer Aided Verification*, pages 464–476, 1998.

[5] Dong Wang, Pei-Hsin Ho, Jiang Long, James H. Kukula, Yunshan Zhu, Hi-Keung Tony Ma, and Robert Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Design Automation Conference*, pages 35–40, 2001.