Metropolis Design Guidelines

Alessandro Pinto
University of California at Berkeley
545P Cory Hall, Berkeley, CA 94720
apinto@eecs.berkeley.edu

November 9, 2004

UCB/ERL M04/40

Contents

Preface and Acknowledgements

The Metropolis design guidelines document describes in which way a design flow should be implemented using the Metropolis-Meta-Model (MMM) language. The MMM language does not impose a modeling scheme since it is just a meta-language that leaves a lot of freedom to the users that may come up with more appropriate ways of implementing their methodology ideas. The guidelines that we present here are based on our experience in using the Metropolis framework, and on the formal definition of the general principle of platform-based design [12],

The guidelines described in this document are divided in three main parts:

- *functional description*;
- *platform description* (or architectural description);
- *mapping*.

For each section, we describe the theoretical aspects and we give examples of modeling. The modeling strategy is justified by the theory and by some considerations about the users expectations in each design stage.

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.

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),

and 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, University of Pisa.

Thanks to Trevor Meyerowitz, Luciano Lavagno and Felice Balarin for their useful feedback. Thanks to Alberto Sangiovanni-Vincentelli and Roberto Passerone for the theory of Platform-Based Design.

Chapter One

Introduction

The Metropolis design environment provides an infrastructure for designing embedded systems. Metropolis is particularly suited to modeling *heterogeneous* systems at *different levels of abstractions* and it is constantly under development with the idea of supporting the platform-based design [12] principle. The main objective is not merely to provide a simulation/verification/synthesis environment for designing electronic systems at a specific level of abstraction, but rather to develop and infrastructure that is flexible enough to allow the development of entire design flows for different application domains.

This ambitious goal is pursued by supporting design principles that are independent of the specific application. Metropolis supports two basic principles: *orthogonalization of concerns* and *platform-based design* [8]. The latter can be applied to many fields of engineering. In this design guide, we use a multimedia design flow as a representative example.

1.1 Platform-based design of Multimedia Systems

A simple example of platform-based design applied to the logic synthesis flow is described in appendix A. In this chapter, we give an overview of the method for multimedia applications.

Going from the denotational description of a function down to its implementation is a very hard problem. The number of possible design choices makes the design space too large to be explored efficiently. The big design gap can be subdivided into smaller steps by introducing a stack of platforms, each dedicated to the exploration of the design space along few directions (figure 1.1).

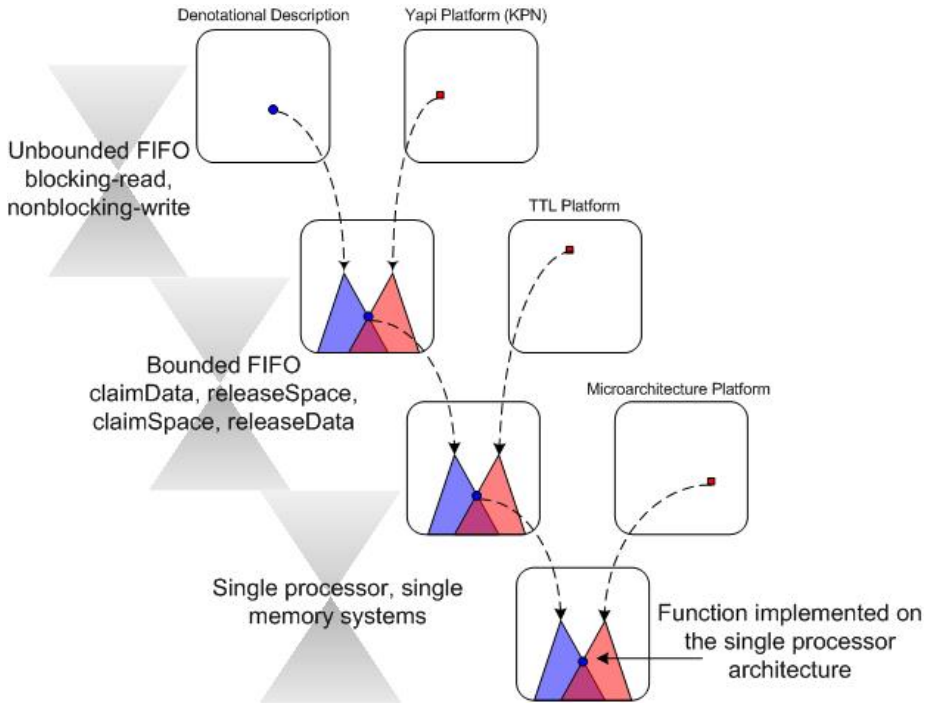

Figure 1.1: Platform stack for multimedia designs

When using a platform-based design methodology, the first important step is to define those levels of abstractions. Consider, for instance, the case of multimedia systems design as shown in figure 1.1. Starting from the denotational description of the function F the first important decision is to select a suitable platform (that a this level of abstraction is usually referred to as a model of computation) to describe F. The only property that we are interested in at this level is correctness. Kahn process networks (KPN) [7] is a convenient model of computation usually adopted as the first platform for modeling multimedia systems. The platform components are processes running on their own threads and

communicating through unbounded FIFOs with blocking read and non-blocking write semantics.

Mapping F onto the Kahn process networks platform implies re-expressing the denotational algorithm as interconnection of concurrent processes and unbounded FIFOs. For instance, an FIR filter can be described by interconnecting adders and multipliers together. At this level of abstraction, we can check if our algorithm is correct without caring about deadlocks due to resource limitations like FIFO boundedness. In addition, since processes are totally concurrent and writing is non-blocking, processes don't have to compete for shared resources. The designer can then explore the maximum amount of concurrency (parallelism) in the functional description without being constrained by resource limitations. However, the function is too abstract to be implemented on a real architecture that has resource limitations, e.g. memory size.

The next level of abstraction is represented by the TTL [4] (task transaction level) platform. The TTL platform is still composed of processes running on their own threads, but the communication among them is implemented with bounded FIFOs. Mapping a KPN function onto a TTL platform is a simple one to one mapping that can be done by a direct refinement of each unbounded FIFO into a more complicated channel as we will see in the next section.

At the TTL level of abstraction, we are concerned with memory size minimization. Each communication channel is parameterized by token-size and maximum number of tokens that it can contain. Depending on the interleaving policy between writer and reader of the same channel, buffer size can be reduced at the expense of a more frequent context switching between the two processes. The memory usage/switching overhead trade-off is explored at this level of abstraction.

The TTL description is mapped onto a real micro-architecture, for instance a single processor architecture. The platform is composed of a library of components that includes different kind of processors, busses and memories. At this level of abstraction, we are concerned with memory allocation and organization and task scheduling. Task scheduling is needed because the number of computational resources is limited to one meaning that the highly concurrent model of the function has to be implemented as a sequential stream of instruction. A real time operating system works as an adaptation layer between the high level of con-

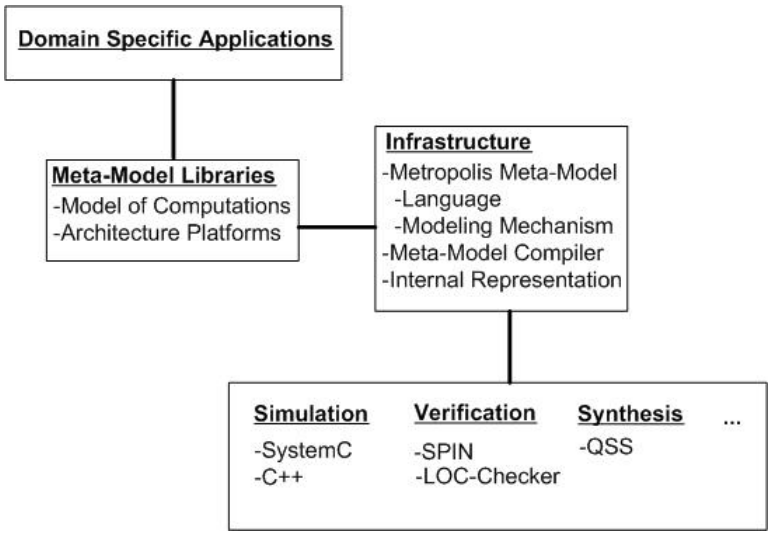

Figure 1.2: The Metropolis framework

currency of the TTL platform and the sequential operations of a single processor architecture.

The result of this mapping is the implementation of the original function on a target micro-architecture. We have to verify that the implementation satisfies the original constraints.

1.2 Metropolis design environment

The idea behind the Metropolis framework is to have an unambiguous common representation of designs and a set of tools that can interpret the common representation, manipulate it and generate a modified description using the same representation [3].

The Metropolis framework is pictorially represented as in figure 1.2

The Metropolis infrastructure is the core of the framework. The infrastructure is composed of a language for the design description that is called the Metropolis Meta-Model (MMM). An MMM program is compiled into an internal representation that retains all the semantic and syntactic information of the original MMM program.

A given application could be directly described using the MMM. However, this approach would lead to a continuous rewriting of code that is

common to many application. A more disciplined approach is to provide a set of platforms from the users can pick components that they need. This approach facilitate the description of functional models as well as architectural models.

There are two kinds of platforms: model of computations and architectures. As described in chapter 3, a model of computation is presented as a set of basic classes that the user extends to customize the behavior of processes and obtain the functional description. As described in chapter 4, an architecture is presented as a set of services and their legal compositions.

Given an application domain, the user selects a suitable model of computation from the available set of libraries. For instance, Kahn process networks (KPN) are widely used in modeling multimedia applications. The KPN library in Metropolis provides processes and media. A process is a basic class with an empty behavior. The user extends this class by adding ports for external communication and overriding the thread method to describe the set of behaviors that belong to the process. Media are unbounded FIFOs that users instantiate and connect to processes. The description of the entire system is still done using the MMM language but the library considerably simplifies its description.

After the functional description as been done, the user selects and architecture instance in the architecture platform. An architecture platform is usually available as a parameterized model. The parameters can be of different nature: number of processors, type of processor and scheduling policy etc. The user builds an architecture instance by setting all this parameters. The architecture is also described using the MMM language. Another way of providing an architecture platform is to provide a set of components with specified interfaces that limit the ways in which components can be interconnected (a port can be connected to a media only if the interfaces match).

Notice that there is no fundamental difference between models of computation and architecture platforms since both are provided as a library of components and composition rules.

Finally, a mapping is obtained by enforcing a synchronization of function and architecture. Each action in the function side is correlated with an action on the architecture side using synchronization constraints. Semantically, this means that the sequence of actions in the architecture side has to follow the execution of the function side which is equivalent

to the intersection of the set of behaviors of function and architecture in a common semantic domain (this concept will be explained in details in chapter 5).

Function, architecture and mapping are all described using the MMM. Since all back-end tools are developed upon the internal data representation, they can be applied to the function, architecture and mapping netlist. It is possible for instance to simulate the architecture without any function mapped onto it.

A declarative language is defined in the MMM to specify properties (constraints) of a design. Two verification tools are provided in the current release to check that constraints are satisfied at each level of abstraction.

A meta-model netlist can be parsed and compiled to generate the internal representation using the command *metacomp*. This process goes through distinct phases of which *elaboration* is probably the most useful. Elaboration can be invoked by the following command line

```
metacomp -elaborator <topnet> <files.mm>
```

where topnet is the top netlist. The elaboration phase compiles the source files and runs the constructor of each component to determine its initial state. It also checks interfaces, types and modifiers.

A number of other tools are available in the current release. The most used is the SystemC simulator that, after elaboration, generates an equivalent description of the original specification in the SystemC language [9]. SystemC simulation back-end generates also a makefile to compile the set of C++ files obtained from the translation. The SystemC back-end can be run using the command line

```
metacomp -systemc  -top <topnet> <files.mm>
```

Instead of using command lines, the user can interact with a design through metroshell, a jacl [6] interface that provides APIs to browse the design and apply backend tools.

1.3 Audience and Organization

The design guideline is intended to give directions to users that want to design embedded systems in the Metropolis framework. This document

touches two aspects: what the user requirements are and how developers should satisfy them. This document does not describe syntax and semantics of the MMM, backend tools, or any other implementation related issues. We target these guidelines to two classes of users: system developers that have to describe applications, select architectures and perform mapping, and libraries provider that have to provide the necessary infrastructure to make the system developers job easier. This guidelines is then an overview of *how* the Metropolis framework, and specifically the MMM, should be used.

The Metropolis guidelines document is organized as follows:

- **Chapter two** describes the basic components used to describe a system. Processes, media, quantity managers and netlists are introduced through a simple example and a brief description of their usage is given.

- **Chapter three** shows how a model of computation library should be used and consequently how it should be developed. Two examples are given: KPN and a multirate synchronous domain that are both provided by the Metropolis release.

- **Chapter four** shows how an architecture platform is assembled and used. It describes how the services offered by an architecture should be exposed to a user and consequently how all components should be developed and described. An example of simple architecture is given.

- **Chapter five** describes the mapping phase. This chapter describes the strategy that we use in the Metropolis framework to describe a mapping and also gives a small example.

Chapter Two

Basic notions on modeling

The Metropolis Meta-Model (MMM) language provides basic building blocks that are used to describe models. These components represent computation, communication and synchronization which are three basic ingredients needed to define a model of concurrency. This chapter uses a simple producer consumer example to introduce the basic objects defined in the MMM language. For a detailed explanation of MMM syntax and semantics refer to [2].

2.1 Processes, Interfaces and Media

Consider a simple producer consumer example. The producer generates a sequence of integers in ascending order starting from 1 and communicates them to the consumer that sinks them.

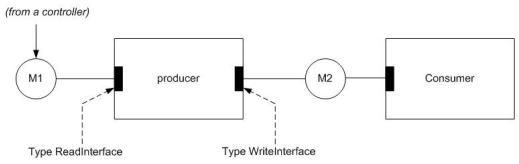

Figure 2.1: A simple producer consumer example

Producer and consumer are two objects that execute two distinct algorithms. More specifically, the producer follows a certain number of steps to implement the original specification. These steps are a set of "actions". Objects of this type are defined as processes in the MMM language. A process runs on its own thread. A thread is a sequence of actions that can be thought of as instructions, sub-instructions (in which an instruction can be decomposed), function calls and awaits (for a formal definition of actions please refer to [2]).

The code implementing the producer is shown in listing 2.1.

Listing 2.1: Meta-Model description of the producer

```
1 process Producer {
    port ReadInterface r ;
3   port WriteInterface w ;
    parameter int N_writes ;
5   int v;
    Producer( String n , int nw ) {
7     super( n ) ;
      N_writes = nw;
9   }
    public void thread( ) {
11    int j;
      for( int i = 0 ; i < N_writes ; i++ ) {
13      v = r.read( ) ;
        j = i + 1 ;
15      out.write( j ) ;
      }
17  }
};
```

The process has two ports (lines $2-3$) and one parameter (line 4) which is specified at instantiation time. Notice that ports do not have direction, but have a type associated with them. The type of a port is an interface which declares services that can be called by the process. In this example, the producer process used the read and write services respectively offered by the `ReadInterface` and `WriteInterface` interfaces.

Figure 2.1 also shows a possible connection of the process to the rest of the system. Processes cannot connect directly to other processes but the interconnection has to go through a medium which has to define (i.e. implement) the services declared by the interface associated with the ports

that access the medium. The constraint of having always a medium between two ports comes form the need for orthogonalization of computation and communication. The meaning of the communication is written in a medium that can be changed and refined without changing the computation description that resides in the processes.

An interface is an abstract class which just declares a set of functions with their associated signatures. The `WriteInterface` for instance is specified as in listing 2.2

Listing 2.2: Meta-Model description of the writing interface

```
interface WriteInterface extends Port {
2   eval void write(int data);
}
```

The interface declares a service `write` which takes an integer `data` as the only parameter. It does not define the service so it is impossible to say what its semantics is.

The syntax used in MMM is very close to Java. A process, like any other object in the MMM, has a constructor which in this case takes a parameter `nw` indicating the number of integers to be produced in ascending order starting from one.

A process specifies a `thread` function (line 10). In this example, producer reads a triggering signal from port `r` (line 13)and writes an integer to port `w` (line 15).

From the outside, the process behavior is observed as a trace of read and write actions, or better yet as a trace of services calls to the media connected to its ports. More precisely the behavior is a sequence of events which are the begin and end event of each action. Looking at the process thread, the behavior is more complex. After reading the trigger, an internal variable (belonging to the state of the process) is assigned to the result of a sum and written to the output port. The behavior, then, not only includes reads and writes but also begin and end events of the assignment and sum actions.

A process is then an object that generates a sequence of events. Each process in a system evolves by executing one event after the other. At each step (which is formally described in [2] in terms of a global execution index), each process in the system executes one event. This is the semantics resembles somehow the semantics of a synchronous language but a special event called NOP is defined in such a way that it can al-

ways be interleaved between to events of a process. The NOP event correspond to the stalling of a process making it possible to implement asynchrony. For instance, if the producer wants exclusive access to medium `M2` of figure 2.1 we can force the consumer to execute the NOP event while the producer is writing a data.

The juxtaposition of events from all processes is an event vector. Intuitively, a program is a big automata where the transition between states is labeled by event vectors. At each step, there is a set of event vectors that could be executed to make a transition from the current state to the next state. If there no scheduling constraints are specified (either imperatively or declaratively), then the choice among all possible transitions is done non-deterministically.

While a process uses services declared by interfaces, a communication medium is an entity that implements services. It does not have a thread of execution but rather inherits the thread from the process that uses the services. A communication medium implements a protocol to exchange information between processes. Separation between processes and media follows the principle of orthogonalization of concerns described in [8].

Listing 2.3 shows a simple channel that implements two services: read and write.

Listing 2.3: Meta-Model description of a communication medium

```
1 medium Channel implements ReadInterface, WriteInterface {

3   int[] storage;
    int space, n, reading, writing ;
5   int length;

7   public Channel(String name, int nelement) {
      n = 0 ;
9     space = nelement ;
      storage = new int[ nelement ] ;
11    length = nelement ;
      reading = 0 ;
13    writing = 0 ;
    } // Constructor
15
    public update void write(int token) {
```

```
17      await {
              ( space > 0 ; ; ) {
19              space = space - 1 ;
                n = n + 1 ;
21              storage[ writing ] = w ;
                writing = writing + 1 ;
23              if ( writing == length ) writing = 0 ;
              } // Critical section
25          } // await
    } // write
27
    public update int read( ) {
29    int _retval = 0 ;
      await{
31          ( n > 0 ; ; ) {
              n = n - 1 ;
33            space = space + 1 ;
              _retval = storage[ reading ] ;
35            reading = reading + 1 ;
              if ( reading == length ) reading = 0 ;
37            return _retval ;
            } // Critical section
39        } // await
    } // read
41 }; // Channel
```

The `Channel` medium implements two interfaces (line 1) which means that it will define the operation code for each of the services declared in those interfaces.

The `await` statement [2] (line 17) can define parallel branches. Each branch has a premises and a critical section. The premises has the form `(guard;testlist;setlist)`. In this example we only have a guard condition that is a boolean expression. If the guard is true than the critical section can be entered. The meaning of *testlist* and *setlist* will be explained later through an example.

The channel has a limited storage space. A reading process will be blocked until there is at least one token in the FIFO, while a writing process will be blocked until there is at least one token space in the FIFO. Note that a channel implementing the same interfaces but using a different implementation could change the communication semantics. For in-

stance, we could implement a non-blocking write operation by just overwriting the current data.

This example show that the interface is an agreement on the services that ares offered and the services that are required in the sense that a process can only use services declared in the interface and on the other hand a medium connected to that port has to provide those services.

In this example, there is no restriction on the simultaneous access to the storage variable in the medium. A reader and a writer can access the medium at the same execution step and modify the internal state simultaneously. When this situation occurs, the final value of the variables is not known. In order to avoid data corruption, a synchronization method between the two processes has to be implemented. The synchronization protocol can be described directly in the medium (using the set list and test list in the await statements) or using an external scheduler that we call quantity manager.

We could use an empty interface to act as a semaphore and include it in the await test and set lists as in listing 2.4.

Listing 2.4: Meta-Model description of a communication medium preventing data corruption

```
1  interface Semaphore {} ;
   medium Channel implements ReadInterface,
3                            WriteInterface,
                             Semaphore {
5
     ...
7    public update void write( int token ) {
       await {
9              ( space > 0 ; Semaphore ; Semaphore ) {
                 ...
11             }
             }
13         }

15   public update int read( ) {
       await{
17       ( n > 0 ; Semaphore ; Semaphore ) {
           ...
19       }
```

```
    }
21  }
}
```

Informally, We can imagine that each interface has an associated boolean flag. Each time a critical section is entered, the flags of all the interfaces in the set list are set to true. Before entering a critical section, though, not only the guard has to evaluate to true but also the flags of all the interfaces in the test list have to be false. In our implementation of the communication channel, if the producer is executing a write operation (line 10) it means that the `Semaphore` interface was flagged and hence a consumer that is about to read will be forced to execute the `NOP` event. In this example the await statements are used to synchronize (schedule) the two processes. The same result can be achieved by using another object called quantity manager as explained in section 2.3

2.2 Netlists

A netlist is an object used to instantiate and connect other components like processes, media and netlists. A netlist has a constructor where all the components are instantiated and interconnected. An API is provided to add a component to a netlist and to connect a port of an object to a medium. The netlist code for our example is shown in listing 2.5.

Listing 2.5: Meta-Model description of a netlist

```
netlist ProducerConsumer {
2   public ProducerConsumer( String n , int nw ) {
      super( n ) ;
4     producer p = new producer( "Prod" , nw ) ;
      consumer c = new consumer( "Cons" ) ;
6     controller ctrl = new controller( "Contr" ) ;
      Channel ch = new Channel( "CommCh" ) ;
8     addcomponent( p , this , "ProdInstance" ) ;
      ...
10    connect( p , w , ch );
      ...
12  }
}
```

Using a Java like syntax, an object is created with the keyword `new` (lines $4-7$). After creation, even if an object resides in memory, it has to be added to the netlist before using it. The function `addcomponent` (line 8) takes as parameters an object, a netlist and a string. This function call adds the object to the netlist and associates the string to the instance name. The `connect` function (line 10) is used to connect the port of an object to a medium.

Netlists can be used to organize a design implementing hierarchy.

2.3 Quantity managers

Quantity managers are used to assign tags to events. A tag is an abstract quantity from a partially order set. Time, for instance, is a real number so in this case the set of tags is totally ordered. When an event has to be tagged with a quantity, an explicit request is made to the manager of that quantity. Due to concurrency of processes, multiple requests can be issued to a quantity manager that has to resolve them and schedule the processes in order to satisfy the ordering relation on the set of tags. Consider the simple producer/consumer example of figure 2.2. We want

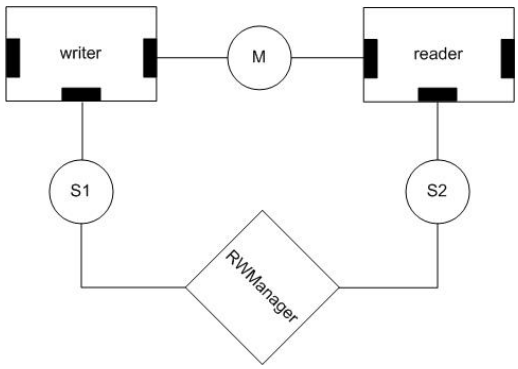

Figure 2.2: A quantity manager

to make sure that internal variables of the medium `M` are not accessed simultaneously by a reader and a writer. We can use a quantity manager to scheduled the two processes. A sample code is shown in listing 2.6.

Listing 2.6: Meta-Model description of a quantity manager

```
1 quantity RWManager implements QuantityManager {
```

```
  port StateMediumSched reader ;
3 port StateMediumSched writer ;

5 public RWManager( String n ) {
    super( n ) ;
7   _pending = new ArrayList( ) ;
     // Constructor code here
9 }

11 public eval void request( event e , RequestClass rc ) {
    _pending.add( (Object)rc.clone( ) ) ;
13 }

15 public update void resolve( ) {
    // For all pending requests
17  // order them depending on the tags ordering
  }
19
  public update void postcond( ) {
21  // Do the first event in the order
    // Do not do the others
23  _pending.clear( ) ;
  }
25

27 public eval boolean stable( ) {
    return true;
29 }

31 ArrayList _pending ;
}
```

Instead of inserting await statements in the medium `M`, we insert a special request code to the quantity manager before the `read` and `write` code. The set of tags that we consider, contains four elements: $T = br, er, bw, ew$. When a process uses the `read` service, before modifying the medium state variables, a request is issued to the quantity manager. The request asks to annotate the begin event of the read function with the tag br. Before returning from the read function, another request is issued to annotate the end event of the read function with the tag er. A similar

set of requests is generated by the write service.

The quantity manager specifies an order on this set of tags that could be, for instance, $br < er$, $bw < ew$, $bw < br$. The last inequality says that in the case of simultaneous access of a reader and a writer, the read is executed first.

In general, quantity managers are used to adapt two different concurrency models. On one side, the set of processes are fully concurrent and can freely execute the sequence of events defined in their threads. On the other side, we have a specific process scheduling mechanism that we want to implement in order to give our program the semantics that we want. For instance, in a discrete time model, time is a totally ordered set of tags and hence a process that wants to execute and event at time t_i has to wait for other processes that are executing events at time $t_j < t_i$.

2.4 Constraints

An important part of the design specification is represented by constraints. They are declarative formulas that are used to specify properties that the implementation has to satisfy. In our simple example, we might want to declare that the time needed to complete a write operation has to be less than a certain quantity T.

Two kinds of formulas are provided by the MMM language: linear temporal logic (LTL) and logic of constraints (LOC). The syntax is explained in [2].

Listing 2.7 shows the declaration of a data consistency constraint.

Listing 2.7: Meta-Model declaration of LOC constraints

```
constraint{
  event Wevent = beg( p , p.write ) ;
  event Revent = end( c , c.read ) ;
  loc( forall ( int i )
         ( v@( Wevent , i) == v@( Revent , i ) )
     ) ;
}
```

The LOC formula (line 4) says that each data that is written by the producer is read by the consumer. The index i is called global execution index (GXI) and is described in **??**.

Constraints are propagated down while marching towards the implementation and have to be constantly checked at each level of abstraction. For instance, if we refine the communication protocol into a more detailed transaction level medium, the formula will not change and has still to be satisfied by the refinement. A back-end tool called `LOC-Checker` is included in the current release to verify that LOC formulas are satisfied.

Chapter Three

Developing platforms for functional description

A design process always starts with the idea of what the system is supposed to do. In our terminology, this is the *function*. It can be described, informally, using natural languages like English, or its description can be more formal. A denotational definition of the function usually involves formulas describing outputs in terms of inputs. A maximum likelihood estimator, for instance, can be described as the solution of a maximization problem:

$$\max \ln P(r|c)$$

where r is the vector of the received bit stream and c is the vector representing the code. The description of the algorithm is denotational and only tells us what the system should do.

Practical and implementable algorithms exist to solve this problem. For example, the Viterbi decoding algorithm is an interconnection of blocks whose final result is an approximation of the maximum likelihood estimation. Being an approximation means that the result is different from the denotational description by a quantity e. If nothing has been stated about the error, then it could be any number but what we really want is that the implementation follows as close as possible the denotational description, where "as close as possible" is estimated by the error

e. Usually another important part of the functional description is a set of *constraints* that an implementation of the algorithm is subject to. In our example, for instance, a bound on the maximum admissible error can be declared as a constraint.

The denotational function is then mapped onto a platform that allows its description in a more "structured" way, usually as interconnection of sub-functions. Precise rules have to be imposed on this structure in order to have a non ambiguous description, or better, we need a uniquely defined way of interpreting the model that gives all and only the system's execution traces.

The set of rules define the semantics of a *model of computation* (MoC). An MoC is defined in terms of how computation, communication and coordination must be carried out in a structured interconnection of objects.

This chapter gives few design guidelines for building MoCs (that we shall call functional platform) in the Metropolis framework and introduces some examples of models of computation.

3.1 Models of computations for describing functions

This section is divided in three parts. This introduction explains briefly what are the properties characterizing a model of computation. The second part shows how users want a MoC to be exposed to them in order to make the functional description easier. The third part shows how a developer should build a platform and expose a model of computation to the users.

We will use the tagged signal model [10] (TSM) formalism (whose basic definitions are given in appendix B) as a denotational framework for stating properties about a model of computation.

Informally speaking, a model of computation can be defined by the set of values V, the set of tags T and the ordering relation on the tags $\leq$, the legal processes P and their communication semantics.

For instance, Khan process networks are characterized by a set of tags that is partially ordered (it is an untimed model) and, since a communica-

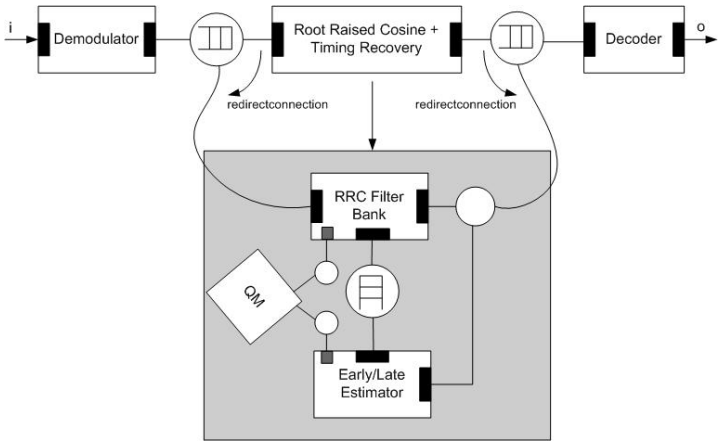

Figure 3.1: Block diagram of a digital receiver

tion channel has a first-input-first-output semantics, each signal is totally ordered.

An MMM process is used to describe all the possible behaviors of a process P while media are used to enforce a communication semantics. An order on the set of tags can be enforced by using a quantity manager that has the task of assigning tags to events. Consequently, a quantity manager has to decide the scheduling of the processes in order to satisfy the ordering of tags.

3.2 Function platforms use-case

Description of applications is considerably simplified when a suitable model of computation is chosen. Depending on the application domain, the first step in a design flow is to choose a natural function platform where the description and interconnection of functions is easy to carry out. For instance, digital signal processing algorithms are naturally expressed using a model where computation is done by blocks called actors that communicate through FIFOs.

Consider the following example. We would like to design a digital receiver that is the cascade of a digital demodulator, a root raised cosine filter, a timing recovery loop and a decoder. The block diagram is shown in the top part of figure 3.1.

The system input signal **i** is a set of totally ordered events. As a first choice we select the Khan process networks [7] (KPN) model of computation. We would like to describe the system as in listing 3.1.

Listing 3.1: Meta-Model description of the digital receiver

```
1 netlist DigitalReceiver {
    public DigitalReceiver( ... ) {
3     // Components instances
      StimuliGen s = new StimuliGen( ... ) ;
5     Demodulator demod = new Demodulator( ... ) ;
      RRCTiming rrct = new RRCTiming( ... ) ;
7     Decoder dec = new Decoder( ... ) ;
      // Add components to this netlist
9     addcomponent( s , this , "sinstance" ) ;
      ...
11    // Interconnections
      FIFOinstance( s.out , demod.in ) ;
13    FIFOinstance( demod.out , rrct.in ) ;
      FIFOinstance( rrct.out , dec.in ) ;
15  }
}
```

We first instantiate all components and add them to the current netlist and finally we interconnect them. Ideally, we would like to have a way of specifying a connection by calling a function and passing the source and destination ports as parameters (line 12). In our example, this function is `FIFOinstance(port src, port dest)` that will instantiate a FIFO channel and connect the two ports to it. If the domain that we are using is KPN, there is no need for scheduling the processes since the blocking read and non-blocking write communication semantics will take care of the process synchronization. As result, the platform will not include a quantity manager or other synchronization constraints (for bounded simulation purposes we might want to control the stimuli generator).

Consider now the refinement of the timing recovery loop shown in the bottom part of figure 3.1. The domain that we want to use in this case has different properties from the previous one. In particular, we want to enforce a specific scheduling of the two processes. The sequence of reaction should be an infinite sequence of the RRC filter bank and the error detector. Using a quantity manager for ordering the events is the right solution. Every time a process wants to read its inputs and com-

pute its outputs, it has to ask the quantity manager first. This request is basically asking to annotate the read-compute-write sequence of events. The quantity manager, based on its pending queue of events, enforces the right scheduling deciding which process can proceed and which has to wait

A user of this model of computation would like to describe the new netlist as in the previous case without having any concern about the quantity manager and synchronization schemes.

A refinement of the original system is obtained by using the keyword `refine` which is explained in [2]. It is also necessary to redirect the original connection to the refining netlist. In this particular case, the communication semantics of the two domains is the same so a direct connection is possible.

However, a complex system could require the use of a heterogeneous model of computation because it spans multiple application domains. A communication system, for instance, is the interconnection of the radio sub-system to the base-band sub-system, which is in turn connected to the data-link sub-system. Each of them has very different properties and a single model for their description is not the right solution. The radio sub-system is best described in a continuous time domain while the base-band sub-system is basically a dataflow kind of application. A direct connection of the two is not possible but a specific interface has to be designed to transform a dataflow signal into a continuous time one.

A user of the framework would like to have a set of interfaces between models that are available to be used. Moreover, each interface should be parameterizable so that the impact of translation of one signal into the other can be evaluated at this level of abstraction.

The following sections explain how to build a model of computation based on this few considerations.

3.3 Architecture of a model of computation

A model of computation is provided as a library of components that a user extends and interconnects to describe a function. Section 3.2 has introduced the user expectations that will determine the design guidelines for platform developers.

When developing a new platform we focus on the components to be exposed to the user, the way of customizing their behavior and a way of interconnecting them. The basic components, together with the APIs they should provide, are shown in figure 3.2:

- Processes for describing computation. A base class has to be provided together with a set of API to access ports values. After extending this class, a user has only to override a method, say `execute` to describe the process behavior.

- Media for describing communication between processes. The communication semantics is usually fixed in a model of computation, so a medium should not be touched by the platform user. However, some parameters could be exposed to configure the media like, for instance, the length of a FIFO.

- Quantity Managers for enforcing a scheduling policy of processes. Scheduling is also a property of a model of computation and should be out of the user control. The instantiation and interconnection of quantity managers should be totally transparent to the user. For instance, in a synchronous model of computation all events in an event vector have the same tag. Platform users know this property but they should not be concerned of how it is enforced.

- Scheduled netlist for describing interconnection of other objects. This netlist should provide a set of APIs to instantiate processes and connect them through media in an automatic way that is dependent on the model of computation.

- Scheduling netlist for instantiating and interconnecting quantity managers and state-media.

- Top netlist to instantiate the scheduled netlist and initialize it. The initialization function should also create the scheduling netlist and connect it to the scheduled netlist.

In the Metropolis framework, platforms for functional specification are provided as packages. All platforms reside in the repository tree under the directory `metro/lib/metamodel/plt`. It is good practice to

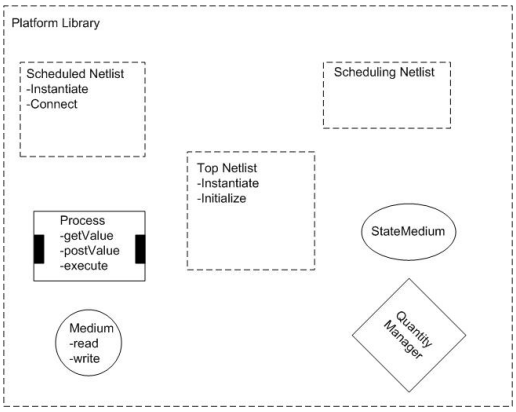

Figure 3.2: Components of a platform

create a new directory for each platform under development. For instance, a dataflow platform would reside under `metro/lib/metamodel/plt/dataflow`.

The rest of the chapter shows two examples of platforms available in the Metropolis framework. The first one does not use quantity managers while the second one, whose semantics has the notion of computation steps, needs a synchronization of the processes execution.

3.4 YAPI: Y-Chart Application Programming Interface

For a complete description of the YAPI semantics refer to [5]. Briefly, YAPI is a KPN model of computation where a non-deterministic choice has been added. More precisely, a process is allowed to check on the number of tokens on an input channel, which is a way of implementing non-determinism in a model of computation.

Processes communicate through unbounded FIFOs. Processes can read from and write to communication channels using two functions: `T[] read( int n )` and `void write( T[] data )` where `T` is the data type. It is possible to parameterize the functions with respect to the data types by using templates but another valid option is to define an interface that all valid data types have to implement. We will use the first

option because it is more natural for designers to use templates that are widely used in C++.

The first things to define are the communication interfaces, meaning the services that the processes need to exchange information. The interfaces definition is shown in listing 3.2.

Listing 3.2: Definition of the communication primitives in the YAPI library

```
package metamodel.plt.yapitemplate ;
  // Interface to check fifos
  public interface yapiinterface extends Port {
    eval boolean checkfifo( int n , int dir ) ;
  }
  // Write interface
  template(T)
  public interface yapioutinterface
                   extends yapiinterface {
    update void write( T data ) ;
    update void write( T[] data , int n ) ;
    update void write(T[][] data , int n , int m ) ;
  }
  // Read interface
  template(T)
  public interface yapiininterface
                   extends yapiinterface {
    update T read( ) ;
    update void read( T[] data , int n ) ;
    update void read( T[][] data , int n , int m ) ;
}
```

We define a service to check how many tokens are present in a FIFO (line 4). This service is actually more general because it can be used on an output port to check the number of available spaces in a FIFO (for a complete description of a reference implementation of the YAPI platform please refer to the platform user's manual). Then we define a writing interface that declares three services for writing data: one for writing matrices, one for vectors and one for single data. Finally, we define a reading interface for reading data. This is sufficient for a designer that does not really care about the services implementation but only about

their signature and semantics. A communication channel is implemented as follows:

Listing 3.3: Implementation of the communication interfaces using a medium

```
1  template( T )
   public medium yapichannel implements yapiininterface-< T >- ,
3                                       yapioutinterface-< T >- ,
                                        rdi , wri , cki {
5    // Definition of internal buffers
       ...
7    // Constructor
     public yapichannel( String n , int isize ) {
9      super( n ) ;
       ...
11   }

13   public update void write( T[] data , int n ) {
       await{
15           (true ; this.rdi , this.cki ; this.wri ) {
               int i ;
17             if ( (ntokens + n) >= size ) {
                 // Resize internal buffer
19               ...
               }
21             else { //OK we can write directly
                 for( i = 0 ; i < n ; i++ )
23                 FIFO[ ( wp + i ) % size ] = data[ i ].clone( ) ;
                 wp = ( wp + n ) % size ;
25               ntokens = ntokens + n ;
               }
27           }
         }
29   }

31   public update void read( T [] data , int n){
       await{
33           (ntokens > n-1 ; this.wri , this.cki ; this.rdi ) {
               for( int i = 0 ; i < n ; i++ ) {
35               data[ i ] = FIFO[ rp ] ;
```

```
                rp = ( rp + 1 ) % size ;
37              }
                ntokens = ntokens - n ;
39            }
        }
41    }
  }
```

The channel is defined as a medium that implement all the interfaces shown before. It also implements some dummy interfaces that are used only to prevent data corruption. Simultaneous access to the same data is prevented using an await statement that before entering a critical section checks if another process is using one of the dummy interfaces. Since the write is non-blocking, the guard condition on the corresponding await statement is always true (line 15). It might happen that the FIFO has to be resized because there is no more space to write. This method is a way of implementing an unbounded FIFO.

The await statement in the read function instead (line 33) has a guard condition that checks the number of tokens in the FIFO before executing the read operation. If the number of tokens is less than requested, the calling process is blocked.

The YAPI platform process offers a function to select non-deterministically a port among a set of ports where a read or write operation can eventually be completed. This function is useful for handling inputs coming from a user. Reading from these inputs will block a process execution until a new token arrives. It is important to provide a way of continuing the execution if there are no tokens available on that channel. This situation occurs for instance when an input is used to change a process behavior during execution. The YAPI platform implements a process as in listing 3.4.

Listing 3.4: Implementation of the library process

```
template(T)
2 public process yapiprocess{

4   public yapiprocess( String n ) {
      super( n ) ;
6   }

8   public int select( ArrayList ports , int[] tokens ) {
```

```
    await{
10          ( atLeastOnePortEnabled( ports , tokens ) ; ; ) {
              return selectOnePort( ports , tokens ) ;
12      }
    }
14  }

16  boolean atLeastOnePortEnabled( ArrayList ports ,
                                   int[] tokens ) {
18    // OR of all checkfifo
    }
20
    int selectOnePort( ArrayList ports , int[] tokens ) {
22    // non-deterministic selection of one port
    }
24
    void thread( ) {
26    execute( ) ;
    }
28
    public void execute( ) { }
30
}
```

The select function is provided in the process implementation. It takes an array of ports and an array of integers where for each port $p[i]$ we request to read/write $n[i]$ tokens. The select function first checks whether there is a least one ports where read or write will not block the process execution. If this condition is true, it selects one among these ports non-deterministically.

The `thread()` function calls an execute function that the platform user override to describe the process behavior.

As an example of usage of the YAPI platform, consider the cascade interconnection of a producer, a filter and a consumer. The filter block implements two algorithms to filter the producer data. A controller decides which filtering algorithm to use (figure 3.3).

The filter code is shown in listing 3.5.

Listing 3.5: A filter described in the MMM using the YAPI library

```
1 import metamodel.plt.yapi.*;
```

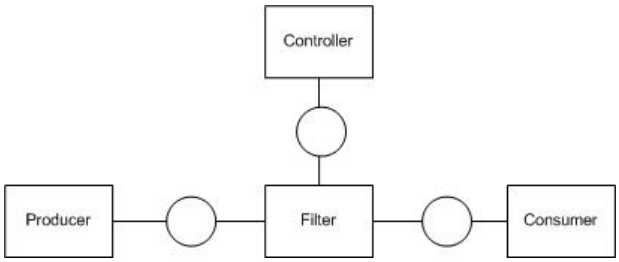

Figure 3.3: Block diagram of the producer-filter-consumer example

```
process filter extends yapiprocess {
  parameter int N_pixels_per_block;
  port yapiininterface -< yapiint >- prod ;
  port yapioutinterface -< yapiint >- cons ;
  port yapiininterface -< yapiint >- ctrl ;

  int p , bc ;
  pixel[] blk ;
  pixel[] blk_out ;
  int filtertype ;

  filter( String n , int npb , int ft ) {
    super( n ) ;
    //Constructor code omitted
  }

  public void execute( ) {
    int enabled ;
    ArryList = new ports ArrayList( ) ;
    int[ ] tokens = new int[ 2 ] ;
    tokens[ 0 ] = N_pixels_per_block ;
    tokens[ 1 ] = 1 ;
    ports.add( prod ) ;
    ports.add( ctrl ) ;

    while ( true ) {
      enabled = select( ports , tokens ) ;
      /* filter always */
      if ( enabled == 0 )
```

```
31        filtertype = ctrl.readint( ) ;

33        if ( filtertype == 1 ) {
            // Use algorithm 1
35        } else {
            // Use algorithm 2
37        }
      }
39    }
  }
```

Users first import the required library (line 1). In order to describe a process, they extend the basic class, define ports and parameters and then override the execute method. The filter process behavior has an infinite loop that selects between two inputs: the producer and the controller. If the controller input is selected, the filter type is read. If the controller has not requested any filter type change, the process execution can go ahead and filters the input data.

Finally, the top level netlist is described is shown in listing 3.6.

Listing 3.6: The top level netlist

```
  import metamodel.plt.yapi.*;
2 public netlist pfcnetlist extends yapinetlist {
    public pfcnetlist( String n ) {
4     super( n ) ;
      producer p = new producer( "Prod" , 48 , 72 ) ;
6     consumer c = new consumer( "Cons" , 48 , 72 ) ;
      filter f = new filter( "Filt" , 72 , 1 ) ;
8     controller ctrl = new controller( "CTRL" , 300 , 150 , 2 ) ;

10    yapichannel-< yapiint >- cpf =
              new yapichannel-< yapiint >-( "CPF" , 100 ) ;
12    yapichannel-< yapiint >- cfc =
              new yapichannel-< yapiint >- ( "CFC" , 100 ) ;
14    yapichannel-< yapiint >- ccc =
              new yapichannel-< yapiint >-( "CCC" , 100 ) ;
16    yapichannel-< yapiint >- cfiltdecision =
              new yapichannel-< yapiint >-( "CFD", 100 ) ;
18    yapichannel-< yapiint >- ccp =
              new yapichannel-< yapiint >-( "CCP" , 100 ) ;
```

```
20      // Add all componets here
        addcomponent( p , this , "ProdInstance" ) ;
22      ...
        // Connect components
24      connect( p , out , cpf ) ;
        ...
26    }
  }
```

The effort from the platform developer point of view is to make all those descriptions (a process, and a netlist of objects) easy and hide as much as possible the parts that are fixed.

3.5 Muti-rate Synchronous model of computation

The semantics of this platform can be informally defined as a sequence of rounds. Let $\{P_i\}$ be a set of communicating processes. When a process P_i executes, it execute a sequence of three actions $write_i$, $read_i$ and $execute_i$. $write_i$ writes the content of the internal output buffer O_i to the output ports. $read_i$ read from the input channels and store the data into an internal buffer I_i. A process is characterized by two numbers: $P_i.r$, which is the execution rate and $P_i.p$, which is the process priority.

Each round has to satisfy the following constraints:

- process P_i has to be executed $P_i.r$ times;
- if $P_i.p > P_j.p$ then P_i has to finish his execution in this round before P_i can execute.

This condition is clearly an ordering of the processes execution depending on rates and priorities. A quantity manager is then needed to model this situation. A detailed description of this platform can be found in the "Polybhrony user's manual" located in `metro/doc/polychronyscaled`.

Each process is connected to a quantity manager that is instantiated into a scheduling netlist. The quantity manager is defined as in listing 3.7.

Listing 3.7: Quantity manager of the polychrony platform

```
1 quantity SynchScheduler implements QuantityManager {
```

```
  port StateMediumSched[] synchprocesses ;
3 public SynchScheduler( String n , int st , int nproc ) {
    super( n ) ;
5   // Constructor code
    ...
7   }

9 public eval void request( event e , RequestClass rc ) {
    // Add requests to the pending list
11 }

13 public update void resolve( ) {
    if ( !_inround &&
15        ( _pending.size( ) == _numberofsynchprocesses ) ) {
      _inround = true ;
17    toWerePending( ) ;
      fairScheduling( ) ;
19  } else
    if ( _pending.size() == _numberofsynchprocesses ) {
21    if ( _st == 0 ) {
        fairScheduling( ) ;
23    } ;
    } else {
25    SynchRequest sr ;
      for( int i = 0 ; i < _werepending.size( ) ; i++ ) {
27      sr = ( SynchRequest ) _werepending.get( i ) ;
        _notdoevents.add( (Object)sr.clone( ) ) ;
29    }
    }
31 }

33 void fairScheduling( ) {
    int currentpriority , subround ;
35  _inround = false ;
    SynchRequest sr ;
37  currentpriority = findMaxPriority( _werepending ) ;
    for( int i = 0 ; i <_werepending.size( ) ; i++ ) {
39    sr = (SynchRequest) _werepending.get( i ) ;
      subround = sr.getClock( ) ;
41    if ( ( subround > 0 ) &&
```

```
            ( sr.getP( ) == currentpriority) ) {
43          _inround = true ;
            sr.setClock( subround - 1 ) ;
45          _doevents.add( (Object)sr.clone( ) ) ;
          } else{
47          _notdoevents.add( (Object)sr.clone( ) ) ;
          };
49      };
    }
51  ...
    public update void postcond( ) {
53    int i ;
      int id ;
55    SynchRequest sr ;
      event e ;
57    for( i = 0 ; i < _doevents.size( ) ; i++ ) {
        sr = (SynchRequest) _doevents.get( i ) ;
59      e = sr.getEvent( ) ;
        synchprocesses[ sr.getId( ) ].setMustDo( e ) ;
61      id = sr.getId( ) ;
      };
63    for( i = 0 ; i < _notdoevents.size( ) ; i++ ) {
        sr = (SynchRequest) _notdoevents.get( i ) ;
65      e = sr.getEvent( ) ;
        synchprocesses[ sr.getId() ].setMustNotDo( e ) ;
67    };
      _doevents.clear( ) ;
69    _notdoevents.clear( ) ;
      _pending.clear( ) ;
71  }

73  public eval boolean stable( ) {
      return true ;
75  }
}
```

If the `resolve()` method is called then the quantity manager checks whether we are in the middle of a round or if a new round should start (line 14).

If we are in a round, then the `fairScheduling` algorithm is called.

This algorithm first computes the maximum priority among all processes with pending requests. Then collects all processes with this priority in a queue. These processes are the one that have to be executed while all the others have to wait. Finally, the `postcond` method sends commands to the processes that force them to execute the `NOP` event (corresponding to the `setMustNotDo` command) or the event that they have requested (corresponding to the `setMustDo` command)

If we are not in a round, then we first wait until all processes have made a request, then we save all requests in an internal array (function `toWerePending`) and then we call the scheduling algorithm.

The rest of the components of this platform are similar to what we have shown for the YAPI platform.

Chapter Four

Developing platforms for architectural description

There is no significant difference between function and architecture platforms. A platform is a set of library components and a set of rules that define their legal compositions. In the case of architecture, it is important to have a way of associating costs and performances to each component and a way of estimating costs and performances of a composition of components. This is very important in order to be able to explore the implementation space and compare solutions to decide which one is the best.

There is though a radical difference between a functional description and an architectural description. Once the denotational description of a function has been mapped onto a platform representing a model of computation, the resulting interconnection of components only implements the original specification. In the case of architecture, an interconnection of library components results in a structure that can implement a set of functions. For instance, a single processor architecture can be used to implement an MPEG decoder or a finite impulse response filter depending on the software that is poured into the program memory.

A platform provider should expose all possible functions that an architecture can support and let the system developer chooses one by determining the architecture behavior in the mapping phase.

4.1 Architecture platforms use-case

We have to consider two different views: platform developers that assemble a platform starting from a set of preexisting library components and platform users that use a platform, configure it and map a set of applications on it. Looking at figure A.1, platform developers would like to define the red point in the platform space starting from the set of IPs that are available in the library. Platform users, instead, want to have control over the total power that the platform can offer which is represented by the light red triangle in the common semantic domain.

Platform developers view. Platform developers have two main concerns:

- they want to define a point in the platform space with certain characteristics (that are usually requested by platform creators) like number of operations per seconds, number of processors, energy per operation etc.
- they needs a way of exposing all implementable functions to the platform users. In fact, they need a way of defining the pink triangle and make it available for mapping a function onto the architecture.

Each component in the library should be characterized with parameters that configure its properties. Parameters like bus arbitration policy, instruction set architecture or packet length for a serialization block are examples of configuration options.

Each component should hide its implementation details and only expose its interfaces that declare the services that the component implements and requires. A bus, for instance, exposes the master interface and has ports whose type is a slave interface. Interfaces constraint the way in which components can be connected. A platform developer would connect components together being aware of the valid compositions that can be statically checked by the Metropolis compiler.

Each component is also characterized by costs and performances. These quantities are modeled using quantity managers. When multiple components are connected together quantities could become dependent from each other. Physical time, for instance, is a global quantity that should be the same for all components. Even for quantities that are not global, a

dependence could arise from the composition. Consider a real time operating system and a bus arbiter. When a task mapped on a CPU reads a variable from the memory, the CPU has to ask permission to the bus arbiter. If the bus is busy in another transfer, the opinion of whether the task should continue its execution or not is different for the two schedulers. A coordination mechanism is then needed in presence of multiple quantity managers.

Coordination among quantity managers could become very cumbersome for platform developers. Ideally, they want the coordination to be totally transparent. A standard coordination algorithm should be provided to the platform developer with very few parameters to act on. For instance, each quantity manager could have a configuration parameter indicating its priority with respect to the other managers. When a platform is built, platform developers could set priorities in order to make the decision of a quantity manager dominate the one of a lower priority manager. In our example, the bus arbiter would have a higher priority than the real time operating system.

Platform users view. Platform users start with a functional description that is then mapped onto several architectures. Each mapping explores the cost/performance trade-off of the particular architecture under consideration. The goal of this design step is to select the best mapping, or better, the best implementation of the original function at a lower level of abstraction.

A platform user operates in the common semantic domain. The architecture is presented as a set of services and all their valid compositions. The function is described using the same king of services. The mapping is then the solution of a covering problem.

Two are the features that platform users require:

- a way of modifying the characteristics of an architecture. This requires that a platform expose a set of parameters through which it is possible to configure its elements. A platform developer has to export the important parameters, characterizing each architecture component, to the level of the platform users.

- A way of "intersecting" the function with the architecture in the common semantic domain. This step can be viewed as a determination of the architecture non-determinism and will be clarified in

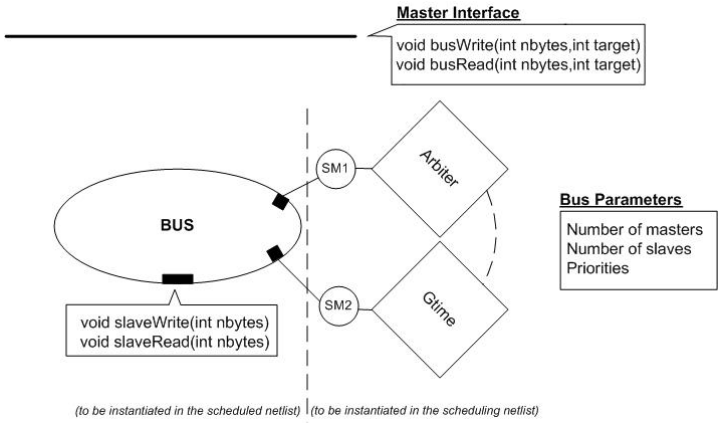

Figure 4.1: A model of a bus

chapter 5. Intuitively, it is a way of assigning pieces of functions to architectural resources that can implement them.

4.2 Architecture platforms development

Platform developers could miss components that they need to built a platform. In this section we provide guidelines for doing two things: providing components and providing an interconnection of components that can implement several functions.

A component offers services that can be used at a cost. A very abstract view of a component is then a medium that exposes some interfaces. Later on a medium can be refined into a netlist if a more detailed description is needed.

Figure 4.1 shows an example of a bus component. The elements to define are:

- the master interface which declares the services that a master can use. In this example there is one service for writing to a specific target and a symmetric service for reading.

- the set of parameters to configure the bus. In this example they are the number of connected masters, the number of connected slaves and the priority of each master.

- A quantity manager that schedules accesses to the bus. This component represents the bus arbiter and decides the ordering of bus accesses among all requests coming form the masters.
- A global time quantity manager. This component is already provided by the Metropolis framework. It is a global quantity and it is in the picture only to show that there is a connection between the bus and the time manager. By using the global time, it is possible to model performances in terms of time required for each operation.
- Statemedia for bus to quantity managers communication. This component implements the communication protocol between the resource and the algorithm that handles it. At sufficiently high level of abstraction, this component implements the identity function passing requests from one component to the other. At lower level of abstraction, the communication protocol could be more complicated and the statemedia should be refined to implement it.

The left hand side of the diagram will be part of a scheduled netlist while the right hand side will be instantiated in the scheduling netlist. The developer should be relieved from the burden of instantiating components, quantity mangers and statemedia and connecting them together. For each component, a function should be provided with the following signature:

Listing 4.1: API for instantiating architectural components

```
public myBus addmyBus( netlist theScheduledNetlist ,
                       netlist theSchedulingNetlist ,
                       ArrayList parameters ,
                       int priority )
```

The function will create instances of all components and set their parameters. Then, it will add components to the proper netlist and make connections. The last parameter is needed in the case of multiple quantity managers interacting with each other. The priority parameter will be used in the resolution phase to decide which manager has precedence over the others (the priority parameter only could be not enough).

Each component can be described following this basic structure and then refined into a more detailed entity. Refinement could require moving part of the scheduling algorithm from the quantity managers to the scheduled netlist.

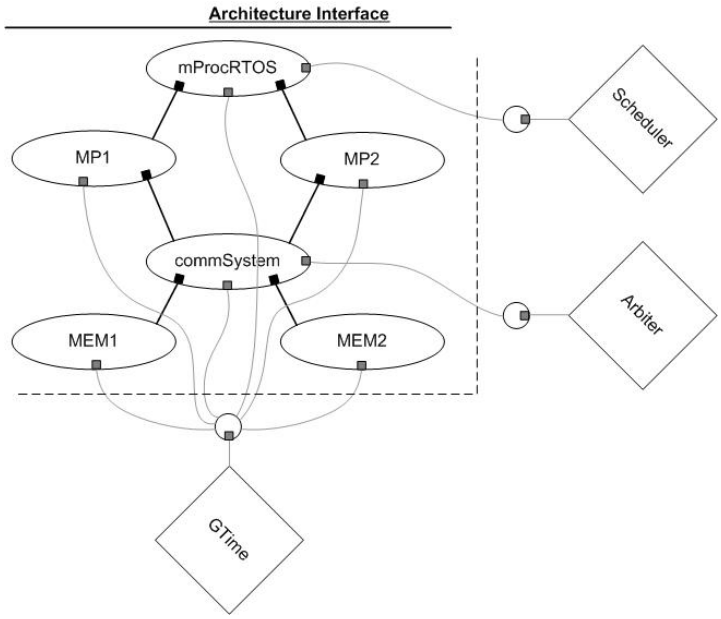

Figure 4.2: Block diagram of a double processor architecture

Consider now a library of components. As platform developers, we want to interconnect them to provide an architecture that can support a variety of applications. We will not deal with the characterization of an architecture as a stand-alone object regardless of the function that is mapped onto it. We rather speculate on how elements of an architecture are connected together, how their parameters are exported and how its potentials are exposed to the platform user.

We chose to provide a double processor architecture. The structure is clear and is shown in figure 4.2.

The architecture exposes an interface that is the real time operating system view of the underlying platform. We select a multi-processor real time operating system, two processors, a communication system, and two memories. We have not specified what kind of components we want to use nor what are the event traces that can be generated by this interconnection of components. Actually, in this abstract architecture there are no processes and hence no events can be generated, or better in the tagged signal model terminology, the only possible execution is the empty behavior.

Before discussing how events are generated and exposed, we describe how components are interconnected and how parameters are exported.

Listing 4.2: Example of architectural netlist

```
public netlist doubleProc {
  public doubleProc( String n ,
                     String comm ,
                     String rtossched ,
                     double timeslice ,
                     double clk1 , double clk2 ) {

    // Instance of components
    Netlist dpscheduled = new Netlist( n + "scheduled" ) ;
    SchdulingNetlist dpscheduling =
          new SchedulingNetlist( n + "scheduling" , true ) ;

    MPRTOS myrtos =
          new MPRTOS( n , dpscheduled , dpscheduling ,
                      rtossched   , timeslice ) ;
    MProc mp1 = MProc( n , dpscheduled , dpscheduling ,
                      clk1 ) ;
    MProc mp2 = MProc( n , dpscheduled , dpscheduling ,
                      clk2 ) ;
    if ( comm == "xbar" ) {
      XBar cmm = new XBar( n ) ;
    } else {
      FCFSBus cmm = new FCFSBus( n ) ;
    }
    ....
    connect( myrtos , p1 , mp1 ) ;
    ...
  }
}
```

The netlist shown in listing 4.2 is an example of how the architecture is described. The important thing to notice is that the netlist represents a family of architectures that are not only different because of the parameters of each component but also because users can decide which communication architecture they wants to use. The architecture will still be double-processor-double-memory but it is possible to select a differ-

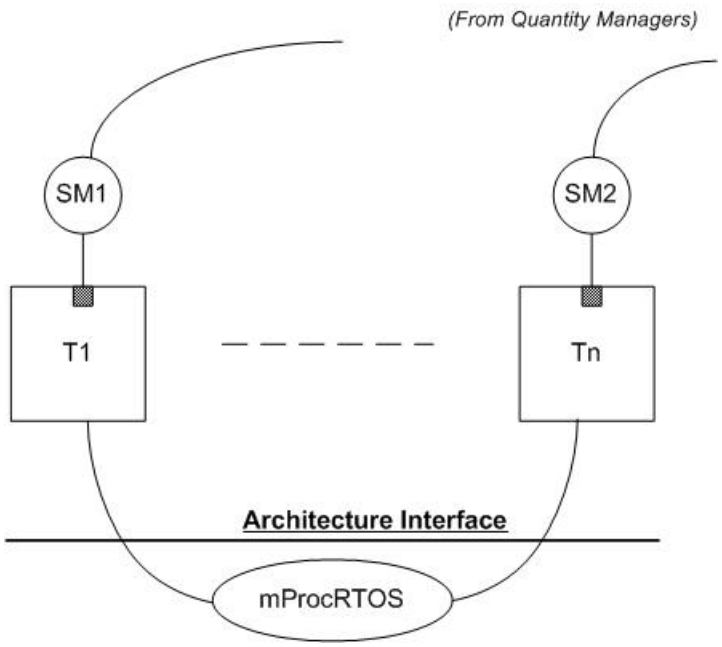

Figure 4.3: Connection of tasks to the architecture

ent way of communicating between the two masters and the two slaves. Parameters are simply exported from the components to the top netlist constructor.

Until now, an architecture appears as an interface that offers services. In our example, the multi-processor real time operating system offers services like request from a task to use the CPU, sleep, request of ownership of a semaphore etc. We now want to represent all possible functions that and architecture can implement that corresponds to giving a description of all legal sequences of those services. We use non-deterministic requests of the services by a set of tasks. We might want to limit not only the number of tasks that can be mapped on the real time operating system but also the way in which the services are called and the parameters that are passed to the services. For instance, we cannot read a memory location that is out of the memory space or we cannot execute a bus read if we do not ask the ownership of the CPU before.

Figure 4.3 shows how processes are connected to the architecture model. Before giving a pseudo code for the thread of a process, we observe that

all quantity managers have to be connected to the process in order to schedule them.

A thread of a process T_i looks like listing 4.3:

Listing 4.3: Thread of an architecture task

```
1 void thread( ) {
    while( true ) {
3     await{
        ( true ; ; ) {
5         cpuRequest( nondeterminism( int ) ) ;
        }// Number of cycles
7       ( true ; ; ) {
          dataRead( nondeterminism( int ) ,
9                   nondeterminism( int ) );
        } //Address, number of bytes
11      ( true ; ; ){
          dataRead( nondeterminism( int ) ,
13                  nondeterminism( int ) );
        }
15      ...
      }
17  }
}
```

The `thread()` function is an infinite loop with only one await statement belonging to it. The await statement has parallel critical sections (lines 5,8,12) all enabled at the same time. It semantically says that, at each time the while loop is executed, one of the critical section is non-deterministically selected. The parameters of each service are non-deterministic values even if bounds should be given in order to satisfy constraints like memory limitations.

4.3 The single-processor-single-memory architecture

In this section, we will create a single processor architecture. We start from the interfaces and then we describe the implementation.

Let us start from the description of a bus. It defines the interfaces to use it. Each bus has its own set of interfaces. The AMBA [1] bus for

instance defines the signals that a master and a slave interfaces have to have. Those are very low-level characterization of the interconnection and here we want to focus more on the services that are provided/requested by/from a bus. For instance, the Open Core Protocol (OCP) [11] defines different kind of transactions that can be requested by a master and also defines the set of services that a slave has to provide.

We want to provide very basic services to transfer n bytes to a target slave T (listing 4.4).

Listing 4.4: Bus interfaces

```
public interface MasterInterface extends Port {
  eval void read ( int target , int addr ,
                   int n , int p ) ;
  update void write( int target , int addr ,
                     int n , int p ) ;
}

public interface SlaveInterface extends Port {
  eval void read ( int target , int addr , int n ) ;
  update void write( int target , int addr , int n ) ;
}
```

Parameters of this functions are:

- $target$: the target slave;
- $addr$: the address on the target memory space;
- n: the number of bytes to transfer;
- p: the priority assigned to the master.

Note that we are not concerned with the actual transfer of data even if adding this information is easy. The parameters are needed only to estimate the time required for transferring the n bytes. A bus needs two quantity managers: the global time manager that annotates time for the begin and end events of a transaction, and a quantity manager that decides the bus owner (the bus arbiter). When a write or read transaction is requested by a master, a request is issued to the bus arbiter that checks whether the bus is available. If multiple requests are pending, the highest priority master will get the bus ownership. Listing 4.5 gives an idea of how the read service is implemented.

Listing 4.5: Implementation of the read service

```
1 public eval void read ( int target, int addr, int n, int p ) {
    event e, r ;
3   br1{@;
      {$
5       beg{
          e = beg( getthread( ) , this.bri ) ;
7         _src.setSchedReqClass( e , REQUEST ) ;
          _portSM.request( e , _src ) ;
9       }

11      $}@};

13  {
      _portSlaves[ target ].busSlaveRead( target, addr, n ) ;
15  }
    br2{@;
17    {$
        end{
19        r = end(getthread( ), this.br2 ) ;
          _src.setSchedReqClass( r, e , RELEASE) ;
21        _portSM.request( r, _src ) ;
        }
23      $}@} ;
}
```

The service uses two labels: `br1` marks the initial request of the bus while `br2` marks the release of the bus indicating that the transfer has been succesfully done. With reference to figure 4.1, when a master calls a read service on a bus, a requrest is issued at the begin event of label `br1`. `_portSM` is a port to the statemedium `S1` that passes the request to the bus arbiter. The arbiter decides if the request can be satified in which case the process that issued the request continues with its execution reads on the target slave. When the read finishes, a request to release the bus is issued to the arbiter that selects a new owner of the bus (if there are pending requests).

The bus arbiter can also ask the global time manager to account for arbitration overhead in which case the begin event of the read operation on the target slave can begin only if the amount of time needed for arbitration has elapsed. Also, the slave read can take a certain amount of

time meaning that the end event of the read function can be executed only if such amount of time has elapsed. This algorithm is implemented in the global time quantity manager.

A master could be for instance a CPU. In order for the CPU to be connected to the bus, it must have a port of type `MasterInterface`. A real time operating system is connected on top of the CPU to share the resources among multiple tasks. Instead of going into the details of the CPU code, whose implementation in MMM can be intuitively understood looking at the bus implementation, we want to give an idea of how multiple quantity managers coordinate to guide the architecture execution.

As in the case of the bus and bus arbiter, the real time operating system has a scheduler that is implemented as quantity manager. The question is: how do we coordinate them in order to make the right decision? Each scheduler implements three functions: `resolve()`, `postcond()` and `stable`. The `resolve()` function looks at the pending requests and based on its scheduling algorithm annotates tags to events. It also decides if an event can be executed or not. The `postcond()` function is used to let each quantity manager know about decisions of the others. The `stable()` function returns a boolean value that should be true only if the schedulers decisions has not changed since last iteration.

The three functions are implemented by the user. The quantity managers are instantiated in a scheduling netlist that is a regular netlist with a resolve method that has to be implemented by the user. The resolve method is called whenever a scheduling decision is needed (for a detailed explanation of this method please refer to [2]). The scheduling netlist resolve method can recursively call the quantity managers resolve methods until all of them are stable (a conditions that can be checked by calling the stable function on each quantity manger). Finally, it calls the `postcond()` method of each quantity manager.

In our case, we could call the bus resolve method first and then the real time operating system scheduler, thus giving the bus arbiter a higher priority. The real time operating system will execute his scheduling algorithm considering only the pending requests that can be executed, without considering tasks that have requested the bus and have to wait until it is available.

After all components have been connected together, the real time operating system exposes the following set of services:

Listing 4.6: Services exposed by the operating system

```
public interface SwTaskService extends Port {
 eval void request( int n ) ;
 eval void read ( int target, int addr, int n ) ;
 update void write( int target, int addr, int n ) ;
 eval void readProtect ( int target, int addr, int n ) ;
 update void writeProtect( int target, int addr, int n ) ;
 eval void readLongProtect ( int target, int addr,
                             int n, int[] data ) ;
 update void writeLongProtect( int target, int addr,
                               int n, int[] data ) ;
}
```

The `request` method only asks the CPU for computation. Three read and write services are provided. The protect version will first try to acquire a semaphore and then accesses the memory location at address $addr$ on target $target$. The long version will transfer the actual data.

The last components that we need are the tasks that use the services provided by the operating system. In our experiment, we want to provide services to transfer data and a service for carrying out computation. The task will look like the following:

Listing 4.7: A non-deterministic software task

```
public process SwTask {
  Nondet _target, _addr, _nTimes ;
  // Constructor
  port SwTaskService rtos ;
  public SwTask( String n, int num_sms, int num_cpus ) {
    super( n ) ;
    rtos = new SwTaskService[ num_cpus ] ;
    ...
  }

  public void thread( ) {
    while( true ){
      await{
        ( true ; ; ) query_space();
        ( true ; ; ) guard_query_space();
        ( true ; ; ) claim_space();
        ( true ; ; ) release_space();
```

```
19          ( true ; ; ) query_data();
            ( true ; ; ) guard_query_data();
21          ( true ; ; ) claim_data();
            ( true ; ; ) release_data();
23          ...
          }
25      }
    }
27
    public void query_space() {
29      rtos[0].readProtect(_target.get(), _addr.get(), 1);
    }
31
    public void release_space() {
33      int[] tokens = new int[1];

35      rtos[0].readLongProtect(_target.get(),
                                _addr.get(), 1, tokens);
37      tokens[0] = tokens[0] - _nTimes.get();
        rtos[0].request(1);
39      rtos[0].writeLongProtect(_target.get(),
                                 _addr.get(), 1, tokens);
41  }
}
```

Part of the code has been omitted. Nondeterministic data structures have been defined to contain the target slave, the address and the number of bytes to transfer (respectively `_target`, `_addr`, `_nTimes`). These variables are non-deterministic quantities in the architecture model but they will be assigned to particular values during the mapping phase.

This architecture provides services for transferring data from and to the memory. They are the same set of services that characterize communication in the Task Transactions level [4] platform described in section 1.1.

For instance, the `release_space` (line 32) method will first read the number of tokens in a FIFO. The number of tokens is a variable that is stored in memory, so the action of reading it requires to access the memory. Moreover the read has to be protected to avoid data corruption. Then the number of tokens in the FIFO has to be updated. This operation

also requires an arithmetic instruction to be executed on the CPU. The request function (line 38) takes into account this overhead. Finally the new number of tokens are written back to memory (line 39).

Chapter Five

Mapping a function onto an architecture

Traditionally, a mapping has been thought of as the assignment of pieces of function to architectural resources. If the part of the function S is assigned to resource R it means that R can implement S but R might be able to implement other functions as well. Assigning S to it is a way of restricting the set of its behaviors. This chapter uses the more general idea of mapping as intersection of function and architecture in a common semantic domain.

Another important aspect of mapping is scheduling. Function and architecture have two different concurrency models. The amount of concurrency on the architecture side is usually determined by the number of computational resources that are available (number of processors for instance). A model of computation like KPN could have a different model of concurrency. If processes do not communicate at all for instance, they all run concurrently. Mapping then asks for the introduction of scheduling of shared resources (real time operating systems, bus arbiter etc.). A scheduler usually have parameters whose optimal value is assigned during mapping.

5.1 Mapping functions onto architectures

The mapping phase has to objectives: implementing a function onto an architecture using its services and selecting the "best" architecture for a particular function (or, more generally, a set of functions belonging to the same application domain). This last objective is usually called *architecture exploration* or *design space exploration*.

A mapping can be done automatically (if a tool exists to do so) or manually. A designer who is purposed to the mapping phase, starts with a function on one side and a set of architectures on the other side. The designer wants to find a "good" mapping. Words like "best" and "good" are quoted because they are uninterpreted by now, but they implicitly assume that mappings can be compared using a metric representing a trade-off between performance and cost.

An important point that has to be considered is the level of abstraction at which the design exploration is carried out. Each level, in fact, corresponds to a design decision, or a set of decisions, that have to be made in order to reach the implementation level. The common semantic domain should be defined in such a way that these decisions can be made. In this section, we assume that such domain as been selected already which means that function and architecture are described in terms of a common set of services. Note that this assumption is not required by the Metropolis framework but it is suggested by the methodology that we advocate.

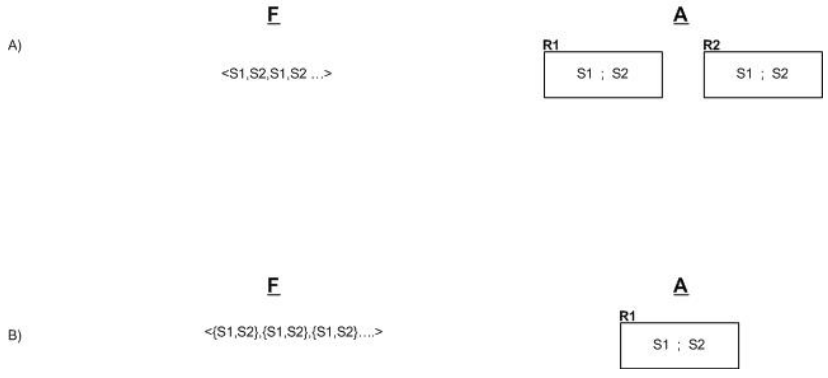

Figure 5.1: Examples of function and architecture services requests/offers

Figure 5.1 shows two mapping scenarios. The function F is described as a set of traces where each trace is a sequence of services. The architecture A is pictorially represented as a set of resources offering services. For instance, case A shows an architecture with two resources R_1 and R_2. Each resource offers two services S_1 and S_2 but only one of them can be in execution at a time.

For each service we shall distinguish between its begin and end event. We shall denote with $beg(S_i)$ the begin event of S_i and with $end(S_i)$ its end event.

Using this simple notation, the function execution trace of case A can be denoted with the sequence $< beg(S_1), end(S_1), beg(S_2), end(S_2)... >$. The architecture execution is described by a trace of pairs of events, each realtive to one resource. Two examples of its partial behaviors are the following:

$$< (beg(S_1), NOP), (end(S_1), NOP), (NOP, beg(S_2)), (NOP, end(S_2))... >$$
$$< (beg(S_2), beg(S_2)), (end(S_2), end(S_2)), .. >$$

Even without a formal definition of common semantic domain and intersection, the reader can easily recognize that while the first trace is compatible with the function trace, the second is not because service S_1 is never executed. Moreover, the first trace corresponds to the assignment of S_1 to resource R_1 and service S_2 to resource R_2.

Case B requires some more observations. Function F requires the concurrent execution of service S_1 and S_2. If there are no constraints on the begin and end events of these two services, then they can be mapped both on resource R_1 and executed in whichever order (S_1 first and then S_2 for instance). This is possible if the events in the function are only partially ordered. If we add the constraint that $beg(S_2)$ as to be executed before $end(S_1)$ then the mapping is possible only if a preemptive scheduling is available on R_1.

If a mapping is not feasible, intersection of function and architecture results in the empty execution.

5.2 Describing mappings using metamodel synchronization constraints

A mapping can be specified by synchronizing the function execution with the architecture execution. First, we have to define synchronization of events. Two events e_1 and e_2 are synchronized if $e_1 \in v \iff e_2 \in v$ where v is an event vector. The synchronization relation is transitive.

Synchronization among events can be enforced using constraints. Consider for instance case A of figure 5.1. We want to map service S_i on resource R_i. This result can be obtained by synchronizing the begin event of S_i in the function side, with the begin event of S_i on resource R_i in the architecture side. We also want to synchronize their end events.

Synchronization is obtained using linear temporal logic (LTL) formulas whose declaration is as follows:

Listing 5.1: A non-deterministic software task

```
ltl synch(e1,e2,...[: v1@(e1,i)==v2@(e2,i),...])
```

This formula means that all the events involved in it are in the synchronization relation. The second part is optional and is used to assign values during mapping. In chapter 4 we have seen that values in the architecture are non-deterministic. They become deterministic during mapping, where an actual value is decided for them. For instance, the memory address of a variable or the priority of a task is decided in the mapping phase.

The LTL formulas are interpreted and synthesized by the simulator that will try to schedule events in order to satisfy them.

There are two important things to highlight:

- a mapping does not require any change in the function or architecture code. This property is important because the design exploration of complex systems could require the evaluation of many different mappings and we don't want to change our models for each new mapping. Using this methodology will just require to change a set of constraints.

- The result of the mapping is again a function at a lower level of abstraction. The mapping will implement the function on the architecture which now has much less non-determinism since the con-

straints have intersected the architecture's set of behaviors with the function's set of behaviors.

- The events of the mapped function are annotated with the quantities produced by the architecture model. Constraints that were declared at the beginning of the design process, like power consumption, and that could not be interpreted at the function level (because power estimation was not available), now can be checked. The implementation has, of course, to satisfy all constraints.

AppendixAppendix

Appendix A

Platform-Based Design example

The platform-based design methodology is a recursive paradigm where the action of mapping a function onto an architecture generates a new function described at a lower level of abstraction and therefore more detailed than the original one.

A design process should start with a denotational description of the function that we want to implement, plus a set of constraints that the implementation has to satisfy. Filtering a signal $x(t)$, for instance, can be denotationally described as $x(t) \otimes h(t)$, namely the convolution of the signal with the filter impulse response $h(t)$. Design constraints are usually specified as propositional formulas over the system quantities. In the case of filtering, for example, we can specify a lower bound on the off-band signal attenuation. Constraints specified at this level of abstraction are propagated down to all subsequent levels, until the implementation level is reached.

While constraints are propagated in a top-down fashion, performances are abstracted in a bottom-up manner. Performance abstraction is the process of hiding details that are not relevant for the level of abstraction under consideration. In fact, each level of abstraction focuses on a particular design choice on which only few quantities have impact. Abstraction of quantities that are not relevant is essential for speeding up the design space exploration.

In this section we focus on one step of the design flow characterized by a function, an architecture and the mapping of the former onto the latter. We use a simplified logic synthesis flow as a representative example.

Figure A.1 shows the design process. The function is described in

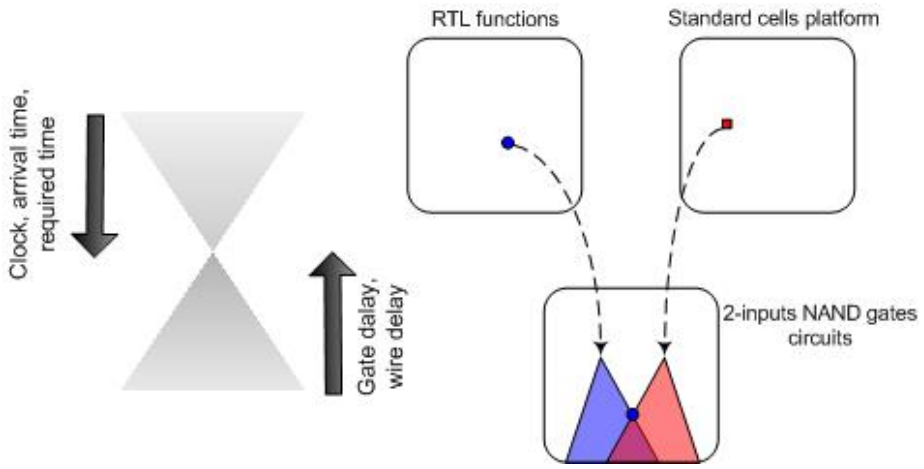

Figure A.1: Platform based representation of a simplified logic synthesis flow

the register transfer level (RTL) domain. In this domain a function is described as interconnection of combinational logic blocks communicating through registers. Furthermore, each combinational block takes zero time to compute its logic function.

The platform is composed of all possible logic functions that can be implemented on a chip using standard cells technology. The library of components, from which the platform is built up, contains pre-characterized logic functions that are usually custom designed to achieve extremely high performances. Each library element is characterized by the its logic function, performance and cost. An OR logic gate, for instance has a truth table, gate delay and power consumption associated with it.

A common semantic domain for the RTL functional description and the standard cell platform is the domain of all circuits built out of 2-inputs NAND gates. In fact, every logic function can be expressed only using NAND gates. Mapping an RTL function onto the standard cell platform can be done in the following way. First, we analyze the RTL description and, for each combinational block, we express its function using 2-inputs NAND gates. Secondly, we express each library cell logic function using the same elementary gate. Finally, the problem reduces to a minimum cost covering of the original function with the library elements with a given cost function (e.g. power or minimum slack).

The covering problem is an optimization problem subject to the constraints coming from the original specification. Typical constraints that are associated with the functional specification are clock speed, inputs arrival time and output required time. After the covering algorithm has finished, those constraints must be checked (using timing analysis). If they are not met, the designer has two choices: going back and modifying the RTL description by using a different block partitioning, or moving the covering algorithm out of the local minima.

The resulting netlist is an interconnection of standard cells which is indeed a new function F'. This function represents a refinement of the original one which was described at a much higer level of abstraction. F' is the new specification for the next design step whose platform library is composed of transistors and wires. Original constraints are propagated further down and must be satisfied by the transistor level implementation.

The simple design process described above turns out to be very general. Once a common semantic domain has been defined so that both the function and the platform elements can be expressed using a common mathematical formalism, design exploration reduces to a covering problem. The covering algorithm has to minimize a cost function which is the sum of the costs of each platform component that has been used during the covering. The whole optimization problem is subject to the original constraints.

Appendix B

Tagged Signal Model Definitions

The tagged signal model is a denotational framework to compare models of computation. In this framework an event e is an element of the cross product $V \times T$ where V is a set of values and T is a set of tags. An order relation defined on T induces and ordering of events. A signal s is a set of events and can be viewed as subset of $V \times T$. It is useful to define a set of signal s and also the set S^N of all sets of N signals. A behavior of a process is an element of the set S^N and a process P is a subset of S^N. s $\in S^N$ satisfies a process if s $\in P$ and it is called a behavior of the process P.

Bibliography

[1] ARM. http://www.arm.com/products/solutions/ambahomepage.html.

[2] Felice Balarin and Yosinori Watanabe. Metamodel language.

[3] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, Apr 2003.

[4] J-Y. Brunel, E.A. de Kock, W.M. Kruijtzer, H.J.H.N. Kenter, and W.J.M. Smits. Communication refinement in video systems on chip. *7th International Workshop on Hardware/Software Co-Design*, May 1999.

[5] W. J. M. Smits P. van der Wolf J.-Y. Brunel W. M. Kruijtzer P. Lieverse K. A. Vissers E. A. de Kock, G. Essink. Yapi: Application modeling for signal processing systems. *Proceedings of the Design Automation Conference*, June 2000.

[6] Jacl. http://www.tcl.tk/software/java/java.html.

[7] Gilles Kahn. The semantics of a simple language for parallel programming. *Proceedings of IFIP Congress 74*, August 1974.

[8] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonolization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[9] SystemC language. http://www.systemc.org.

[10] E. Lee and A. Sangiovanni-Vincentelli. A denotational framework for comparing models of computation. *Technical Memorandum UCB/ERL M97/11*, November 1997.

[11] Open Cores Protocol. http://www.ocp.org.

[12] Alberto Sangiovanni Vincentelli. Defining platform-based design. *EEDesign of EETimes*, February 2002.