A Formal Approach to System Level Design: Metamodels and Unified Design Environments

Felice Balarin[1], Roberto Passerone[1], Alessandro Pinto[2] and Alberto L. Sangiovanni-Vincentelli[2]
[1]Cadence Berkeley Laboratories, Berkeley, CA 94704
[2]Department of EECS, University of California, Berkeley, Berkeley CA 94720

Abstract

The debate about efficient methods for hardware-software co-design has taken interesting turns over the years. In this paper, we argue that the essential problems to solve are prior to the decision on how to partition the system in hardware-software. We present a formal Platform-based design method we have proposed over the years and a design environment, Metropolis, supporting the methodology, which starts by capturing the design specifications at the highest level of abstraction and then proceed toward an efficient implementation by subsequent refinement steps.

We present the modeling strategy used in Metropolis based on formal semantics that is general enough to support the models of computation proposed so far and that facilitates the creation of new ones. Non-functional and declarative constraints can also be captured using a logic language.

1 Introduction

A well-structured design flow must start by capturing the design specifications at the highest level of abstraction and proceed toward an efficient implementation. The critical decisions are about the architecture of the system (processors, buses, hardware accelerators, memories, and so on) that will carry on the computation and communication tasks associated with the overall specification of the design. In a formalized platform-based design methodology [5, 8, 2] this design process is segmented into a series of similar steps. The principles at the basis of each step consist of hiding unnecessary details of an implementation, summarizing the important parameters of the implementation in an abstract model, and limiting the design space exploration to a set of potential platform instances. The design process is a meet-in-the-middle approach where the refinement from specification toward implementation is matched against a library of components whose models are abstractions of possible implementations. The process is complemented by a careful specification and identification of the communication mechanism that simplifies design reuse.

The Metropolis environment supports platform-based design in a unified framework. The idea is to provide an infrastructure based on a model with formal semantics that is general enough to support the models of computation proposed so far and that facilitates the creation of new ones. The model, called meta-model for its characteristics, can be used to capture and analyze the desired functionality, as well as to describe an architecture and the associated mapping of the functionality onto the architectural elements. Since the model has formal semantics, it can be used to support a number of synthesis and formal analysis techniques in addition to simulation. Non-functional and declarative constraints can also be captured using a logic language.

This paper presents a summary of the formal aspects of platform-based design and of the Metropolis infrastructure. We then focus on the basic structure of the meta-model, and define its semantics in terms of a formalism based on automata. Finally we describe the way denotational constraints are captured with a particular language called Language of Constraints. Metropolis is pretty unique in the landscape of system level design tools and environments available today because of the methodology it supports, its modeling strategy that allows to represent vertical and horizontal heterogeneity in a rigorous way, the capability of dealing with physical quantities and of mixing operational and denotations specifications.

Other frameworks provide languages for system level exploration. Ptolemy [1], SystemC [4] and SpecC [3] are all system-level modeling languages or frameworks, bearing several superficial similarities to the meta-model, since all share the notion of concurrent processes communicating through channels. However, they lack features that are necessary to orthogonalize functionality and architecture, such as the mapping between functional and architectural networks, or between different refinement levels. Secondly, they lack the ability to explicitly represent constraints. Finally, the use of the semantics of the underlying C/C++ language (in particular pointers) hinders automated synthesis and optimization.

Tools from Arexsys, Foresight, Artisan, and CardTools are very close in spirit and implementation to Metropolis. They all use a separation between functionality and architectural resources, and they all use a mapping to derive performance information. Of those, only ArchiMate from Arexsys has the capability to model system functionality using a formal language, SDL, while the others use C or C++. SDL, however, is a good modeling language for a specific class of systems like telecommunication protocols, but it lacks expressive power when it comes to modeling other aspects, such as digital signal processing and multi-media.

Commercial hardware/software co-verification tools from other companies (e.g. Mentor, Vast, Virtio and Axys) are essentially Instruction Set Simulators, linked to various hardware simulators. They provide functional and performance models for software-dominated embedded systems, but do not tackle the issues of high-level modeling of hardware, of separation of concerns, and of refinement. In particular, building a new platform model is an expensive and time-consuming task. Also setting up a performance simulation and various design space exploration activities, such as assigning a function to a task or using a different peripheral for a given hardware acceleration, are time-consuming operations. Simulating functional models annotated with performance information, as in Metropolis, is much easier but usually less precise. In practice, co-verification of this kind can be targeted as back-ends by Metropolis.

2 Platform-Based Design Theory

We can describe the process of platform-based design in general terms by identifying an abstraction layer by a set of computation components, called *agents*, and an operation of (parallel) composition that embodies the appropriate communication semantics among the agents. This approach has been developed in the theory of Agent Algebra [6]. This simple structure can be used to formally describe the process of successive refinement in a platform-based design methodology, where refinement is interpreted as the concretization of a *function* in terms of the elements of a *platform*. The process of design consists of evaluating the performance of different kinds of instances in the platform by mapping the functionality onto its different elements. The implementation is then chosen on the basis of a cost function. We use three distinct domains of agents to characterize the process of mapping and performance evaluation. The first two are used to represent the platform and the function, while the third, called the *common semantic domain*, is an intermediate domain that is used to map the function onto a platform instance.

A platform, depicted in n the right, corresponds to the implementation search space.

Definition 2.1 A platform consists of a set of elements, called the *library elements*, and of *composition rules* that define their admissible topologies of interconnection.

To obtain an appropriate domain of agents to model a platform, we start from the set of library elements D_0. The domain of agents D is then constructed as the closure of D_0 under the operation of parallel composition. In other words, we construct all the topologies that are admissible by the composition rules, and add them to the set of agents in the domain. Each element of the architecture platform is called a *platform instance*.

A domain of agents D can then be constructed as follows. If $p_0 \in D_0$ is a library element, we include the *symbol* $\mathbf{p_0}$ in the set of agents $\mathcal{Q}.D$. We then close the set D under the operation of parallel composition. However, we represent a composition $p = p_1 \| p_2$ in $\mathcal{Q}$ as the *sequence of symbols* $\mathbf{p_1} \parallel \mathbf{p_2}$. By doing so, we retain the *structure* of the composite, since all the previous composition steps are recorded in the representation. We call this process a *platform closure*.

Definition 2.2 Given a set of library elements D_0 and a composition operator $\|$, the *platform closure* has domain

$$D = \{\mathbf{p} : p \in D_0\} \cup \{\mathbf{p_1} \parallel \mathbf{p_2} : \mathbf{p_1} \in D \wedge \mathbf{p_2} \in D\} \quad (1)$$

where $p_1 \| p_2$ is defined if and only if it can be obtained as a legal composition of agents in D_0.

The construction outlined above is general, and can be applied to building several different platforms. The result is similar to a term algebra with the "constants" in D_0 and the operation of composition. Unlike a term algebra, however, our composition is subject to the constraints of the composition rules. For example, an "architecture" platform may provide only one instance of a particular processor. In that case, topologies that use two ore more instances are ruled out. In addition, the final algebra must be taken up to the equivalence induced by the required properties of the operators. For example, since parallel composition must be commutative, $\mathbf{p_1} \parallel \mathbf{p_2}$ should not be distinguished from $\mathbf{p_2} \parallel \mathbf{p_1}$. This can be accomplished by taking the appropriate quotient relative to the equivalence relation. The details are outside the scope of this paper.

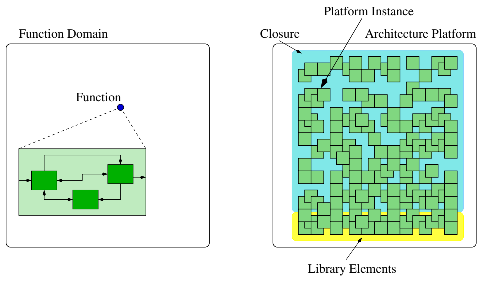


Figure 1. Architecture and Function Platforms

On the other hand, the function, depicted in n the left, is represented in a *specification domain*. Here the desired function may be represented denotationally, as the collective behavior of a composition of agents, or may retain its structure in terms of a particular topology of simpler functions, as shown above. The denotational representation is typically used at the beginning of the platform-based design process, when no information on the structure of the implementation is available. Conversely, after the first mapping, the subsequent refinement steps are started from the mapped platform instance, which is taken as the specification. Thus, a *common semantic domain*, described below, is used as the specification domain. However, contrary to the mapping process that is used to select one particular instance among several, when viewed as a representation of a function the mapped instance is a specification, and it is therefore fixed.

The function and the platform come together in an intermediate representation, called the *common semantic domain*. This domain plays the role of a common refinement and is used to combine the properties of both the platform and the specification domain that are relevant for the mapping process. The domains are related through abstraction functions.

Definition 2.3 *Given a platform $\mathcal{Q}^P$ and specification domain $\mathcal{Q}^S$, a* common semantic domain *is a domain of agents $\mathcal{Q}^C$ related to $\mathcal{Q}^P$ and $\mathcal{Q}_S$ through abstractions Ψ^P and Ψ^S, respectively.*

In particular, we assume that the inverse of the abstraction is defined at the function that we wish to evaluate. The function therefore is mapped onto the common semantic domain as shown in This mapping also includes all the refinements of the function that are consistent with the performance constraints, which can be interpreted in the semantic domain.

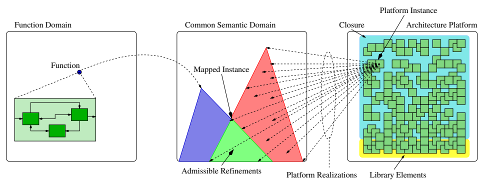


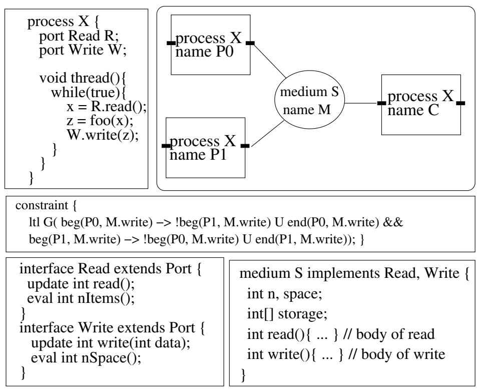
Figure 2. Mapping of function and architecture

If the platform includes programmable elements, the correspondence between the platform and the common semantic domain is typically more complex. In that case, each platform instance may be used to implement a variety of functions, or behaviors. Each of these functions is in turn represented as one agent in the common semantic domain. A platform instance is therefore projected onto the common semantic domain by considering the collection of the agents that can be implemented by the particular instance. This projection, represented by the rays that originate from the platform in may or may not have a greatest element. If it does, the greatest element represents the non-deterministic choice of any of the functions that are implementable by the instance.

The semantic domain is partitioned into four different areas. We are interested in the intersection of the refinements of the function and of the functions that are implementable by the platform instance. This area is marked “Admissible Refinements” in Each of the admissible refinements encodes a particular mapping of the components of the function onto the services offered by the selected platform instance. These can often be seen as the *covering* of the function through the elements of the platform library. Of all those agents, those that are closer to the greatest element are more likely to offer the most flexibility in the implementation. Once a suitable implementation has been chosen (by possibly considering different platform instances), the same refinement process is iterated to descend to an even more concrete level of abstraction. The new function is thus the intersection of the behavior of the original function and the structure imposed by the platform. The process continues recursively at increasingly detailed levels of abstraction to come to the final implementation.

3 The Metropolis Meta Model

The Metropolis meta-model is a language used to specify networks of concurrent objects, each taking actions sequentially. The behavior of a network is formally defined by the execution semantics that we explain in Section 4. This section describes how the meta-



```
process X {
   port Read R;
   port Write W;

   void thread(){
      while(true){
          x = R.read();
          z = foo(x);
          W.write(z);
      }
   }
}
```

```
constraint {
   ltl G( beg(P0, M.write) -> !beg(P1, M.write) U end(P0, M.write) &&
   beg(P1, M.write) -> !beg(P0, M.write) U end(P1, M.write)); }
```

```
interface Read extends Port {
  update int read();
  eval int nItems();
}
interface Write extends Port {
   update int write(int data);
   eval int nSpace();
}
```

```
medium S implements Read, Write {
  int n, space;
  int[] storage;
  int read(){ ... } // body of read
  int write(){ ... } // body of write
}
```

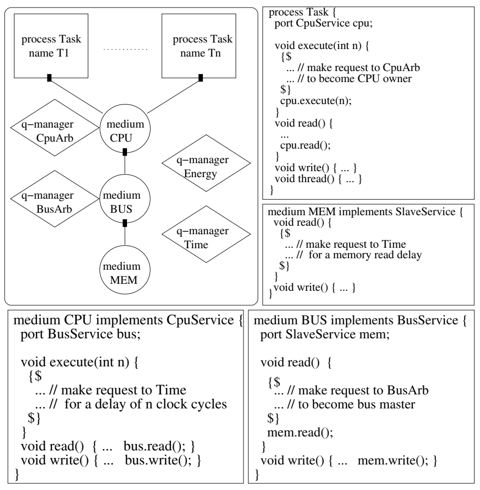
Figure 3. Functional Model of Two Producers and One Consumer

model can be used to specify different aspects of the design, such as functionality, architecture and mapping.

3.1 Function modeling

The function of a system is described as a set of processes that concurrently take actions while communicating with each other. Each process is associated to a sequential program called *thread*. Processes communicate through *ports*. A port is specified by an *interface* that declares a set of methods that can be used by the process through the port. In general, one may have a set of implementations of the same interface, and we refer to objects that implement port interfaces as *media*. Any medium can be connected to a port if it implements the interface of the port. This mechanism allows the meta-model to separate the computation of the processes from their communication. This separation is essential to facilitate the description of the objects to be reused for other designs. hows a network of two producer processes and one consumer process that communicate through a medium.

The behavior of a network is precisely defined by the meta-model semantics as a set of *executions*. First, we define an execution of a process as a sequence of *events*. Events are entries to or exits from some piece of code in a program. For example, for process X in the beginning of the call to $R.read()$ is an event, as is its termination. An execution of a network is defined as a sequence of sets of events, where each set contains one event for each process. The semantics of a network consists of the set of its possible executions. The meta-model also supports non-deterministic behavior, which is useful to describe abstract specifications of part of the design. Declarative *constraints* in the form of logic formulas can be used to further restrict the set of executions of a network (Section 4). For example the constraint in pecifies the mutual exclusion of the two



```
process Task {
  port CpuService cpu;

  void execute(int n) {
    {$
      ... // make request to CpuArb
      ... // to become CPU owner
    $}
    cpu.execute(n);
  }
  void read() {
    ...
    cpu.read();
  }
  void write() { ... }
  void thread() { ... }
}
```

```
medium MEM implements SlaveService {
  void read() {
    {$
      ... // make request to Time
      ... //  for a memory read delay
    $}
  }
  void write() { ... }
}
```

```
medium CPU implements CpuService {
  port BusService bus;

  void execute(int n) {
    {$
      ... // make request to Time
      ... //  for a delay of n clock cycles
    $}
  }
  void read()  { ...   bus.read(); }
  void write() { ...   bus.write(); }
}
```

```
medium BUS implements BusService {
  port SlaveService mem;

  void read()  {

    {$
     ... // make request to BusArb
     ... // to become bus master
    $}
    mem.read();
  }
  void write() { ...   mem.write(); }
}
```

Figure 4. An architectural model

producers when one of them calls the write method of the medium. Constraints allow one to describe coordination of processes, or to relate behavior of networks through mapping or refinement as presented later.

3.2 Architecture modeling

Architectures are distinguished by two aspects: the functionality that they are capable of implementing, and the efficiency of the implementation. In the meta-model, the former is modeled as a set of *services* which are simply methods bundled into interfaces [9]. The efficiency of an implementation is represented by the cost of each service. This is done first by decomposing each service into a sequence of events, and then by annotating each event with a value representing the cost of the event.

To decompose the services into sequences of events, we use networks of media and processes, just as in the functional model. These networks often correspond to physical structures of implementation platforms. For example, Figure 4 shows an architecture consisting of n processes, *T1, . . . , Tn*, and three media, *CPU*, *BUS* and *MEM*. The processes model software tasks executing on a CPU, while the media model the CPU, the bus, and the memory. The services *offered* by this architecture are the *execute, read* and *write* methods implemented in the *Task* processes. The *thread* function of a *Task* process repeatedly and non-deterministically executes one of the three methods. This way, we model the fact that the *Tasks* are capable of executing these methods in any order. The actual order will be fixed only after the system function is mapped to the architecture, when each *Task* implements a particular process of the functional model.

While a *Task* process offers its methods to the functional part of the system, the process itself *uses* services offered by the *CPU* medium, which, in turn, uses the services of the *BUS* medium. In this way, the top-level services offered by the *Tasks* are decomposed into sequences of events throughout the architecture.

The meta-model includes the notion of *quantity* used to annotate individual events with values measuring cost. For example, in Figure 4 there is a quantity named *energy* used to annotate each event with the energy required to process it. To specify that a given event takes a given amount of energy, we associate with that event a *request* for that amount. These requests are made to an object called *quantity manager*, represented by a diamond-shaped symbol, which collects all requests and fulfills them, if possible.

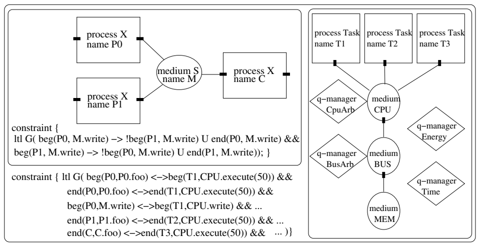
Quantities can also be used to model shared resources. For example, in Figure 4 quantity *CpuArb* labels every event with the task identifier of the current CPU owner. Assuming that a process can progress only if it is the current CPU owner, the *CpuArb* manager effectively models the CPU scheduling algorithm.

The meta-model has no built-in notion of time, but time can be modeled as yet another quantity that puts an annotation, in this case a time-stamp, to each event. Managers for common quantities, such as time, are provided as standard libraries with Metropolis, and are understood directly by some tools (e.g. time-driven simulators) for the sake of efficiency. However, quantity managers can also be written by design flow developers, in order to support quantities that are relevant for a specific application domain.

3.3 Mapping and Platforms

To evaluate the performance of a particular implementation, a functional model needs to be mapped to an architectural model. In the meta-model, this is possible without modifying the functional and architectural networks. A new network can be defined to encapsulate the functional and architectural networks, and to *relate* the two by synchronizing events between them. This new network is called a mapping network.

The synchronization mechanism roughly corresponds to an intersection of the sets of executions of the functional and architectural models. Functional model executions specify a sequence of events for each process, but usually allow arbitrary interleaving of event sequences of the concurrent processes, as their relative speed is undetermined. On the other hand, architectural model executions typically specify each service as a timed sequence of events, but exhibit non-determinism with respect to the order in which services are performed, and on what data. The mapping eliminates from the two sets all executions except those in which the events that should be synchronized always appear simultaneously. Thus, the remaining executions represent timed sequences of events of the concurrent processes.



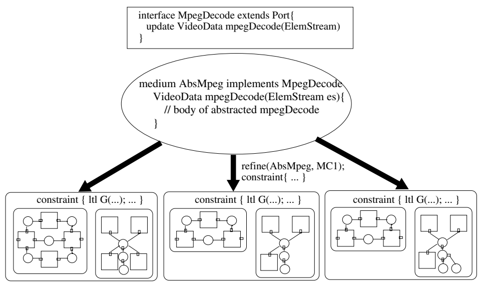
```
constraint {
 ltl G( beg(P0, M.write) −> !beg(P1, M.write) U end(P0, M.write) &&
 beg(P1, M.write) −> !beg(P0, M.write) U end(P1, M.write)); }
constraint { ltl G( beg(P0,P0.foo) <−>beg(T1,CPU.execute(50)) &&
           end(P0,P0.foo) <−>end(T1,CPU.execute(50)) &&
           beg(P0,M.write) <−>beg(T1,CPU.write) && ...
           end(P1,P1.foo) <−>end(T2,CPU.execute(50)) && ...
           end(C,C.foo) <−>end(T3,CPU.execute(50)) &&   ... )}
```

Figure 5. Mapping of function to architecture

For example, hows a mapping network that combines the functional network from ith the architectural network from Figure 4. Events of the two networks are synchronized using the `constraint` clause in the mapping network. For example, executions of *foo*, *read*, and *write* by *P0* have been synchronized with executions of *execute*, *read*, and *write* by *T1*. Since *P0* executes its actions in a fixed order while *T1* chooses its actions non-deterministically, the effect of synchronization is that *T1* is forced to follow the decisions of *P0*, while *P0* "inherits" the quantity annotations of *T1*. In other words, by mapping *P0* to *T1*, *T1* becomes a performance model of *P0*. Similarly, *T2* and *T3* become performance models of *P1* and *C*, respectively.

The network of ay itself be considered an implementation of a certain service. The algorithm for implementing the service is given by the functional network while its performance is defined by the architecture counterpart. The service can be defined in an interface declaration, and it can be used in other models in the form of a medium that implements the interface at an appropriate level of abstraction. We can then relate the medium and the implementation network using the meta-model construct `refine` and constraints. For example, the constraints may say that some event of the medium is synchronized with an event of a process in the network, or that the values of variables in the two models agree.

In general, many mapping networks may exist for the same service with different algorithms or architectures, as shown in Such a set of networks, together with constraints on event relations for a given interface implementation, constitutes a *platform.* The elements of the platform provide the same service with different costs, and one is favored over another for given design requirements. This concept of platforms appears recursively in the design process. In general, an implementation of what one designer conceives as the entire



```
interface MpegDecode extends Port{
  update VideoData mpegDecode(ElemStream)
}
```

```
medium AbsMpeg implements MpegDecode
    VideoData mpegDecode(ElemStream es){
        // body of abstracted mpegDecode
    }
```

```
refine(AbsMpeg, MC1);
constraint{ ... }
```

```
constraint { ltl G(...); ... }
```

```
constraint { ltl G(...); ... }
```

```
constraint { ltl G(...); ... }
```

Figure 6. Multiple Mapping Netlists forming a Platform

system is a refinement of a more abstract model of a service, which is in turn employed as a single component of the larger system. For example, a particular implementation of an MPEG decoder is given by a mapping network, but its service may be modeled by a single medium, where the events generated in the medium are annotated with performance quantities to characterize the decoder.

4 Meta Model Semantics

In this section we give a brief overview of the meta-model semantics. Full details can be found in [10]. The semantic domain we use to interpret executions of meta-model networks is a set of sequences of *observable events*. An observable event is a beginning or an ending of an *observable action*, and observable actions are calls of media functions made through ports.

While the behavior is defined by observable actions only, we also use other actions to help us define the semantics. This extended set of actions include all the statements in the program, as well as certain expressions within a statement. Some actions have a name associated with them, e.g. functions, while others are anonymous. For function with names we use `beg(p,f)` to denote the event of process with identifier `p` beginning to execute action with name `f` (more than one process can execute an action with the same name). Similarly, `end(p,f)` denotes the event of process `p` ending the execution of action `f`.

The execution of a network evolves through a sequence of *state transitions*:

$$s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_i} s_i \xrightarrow{\sigma_i+1} \dots \tag{2}$$

where σ_i is a set of named actions (at most one for each process), and s_i is a network *state*. A state of network consists of two parts. The first is the state of the *memory* which consists of assignments to all variables in all processes. The second part of the state corresponds intuitively to the program counters and stack of all the processes in the network.

The semantics of a meta-model process execution is quite similar to standard programming languages. It differs only in places affected by the fact that the execution of the entire network consists of concurrent executions of many process. The problem arises when two processes concurrently execute actions that are in conflict, i.e. both modifying the same variable, or one reading a variable being modified by the other. In general, our approach is conservative, if there is any possibility of a conflict, the result may be arbitrary. This is the most general approach that covers any possible implementation.

To eliminate such conflicts, the meta-model provides the `await` statement to coordinate concurrent processes. The syntax of the `await` statement is:

$$\texttt{await}\{(e_1; T_1; S_1)stmt_1; \cdots (e_k; T_k; S_k)stmt_k; \} \ .$$

where e_i's are side-effect-free Boolean expressions, $stmt_i$'s are meta-model statements, and T_i's and S_i's are lists of pairs of the form $m.f$, where m is a communication medium, and f is an interface implemented by m. With each T_i we associate the set $[\![T_i]\!]$ that is the union of the sets of actions associated with each pair $m.f$ in T_i. An action a is associated with $m.f$ if it is an action of a process other than the one executing the `await` statement, and one of the following holds:

- a is a call to a member function of medium m which is declared in interface f,
- a is a statement appearing in some other the `await` statement $\texttt{await}\{\cdots(e_j; T_j; S_j)a\cdots\}$, and $m.f$ appears in the list S_j.

Similarly, with each S_i we associate the set $[\![S_i]\!]$ that is the union of the sets of actions associated with each pair $m.f$ in S_i, and an action a is associated with $m.f$ if it is an action of a process other than the one executing the `await` statement, and one of the following holds:

- a is a call to a member function of medium m which is declared in interface f,
- a is a statement appearing in some other await statement $\texttt{await}\{\cdots(e_j; T_j; S_j)a\cdots\}$, and $m.f$ appears in the list T_j.

After an `await` statement begins its execution, one of the statements $stmt_1, \dots, stmt_k$, may start executing. However a statement $stmt_i$ may start execution only if e_i evaluates to *true* in the current state, no actions in $[\![T_i]\!]$ are currently executing and no actions in $[\![S_i]\!]$ are starting execution. Furthermore, no action in $[\![S_i]\!]$ can start executing as long as $stmt_i$ is executing.

Intuitively, we can associate a flag with each pair $m.f$. The statement $stmt_i$ is enabled only if the guard is true and none of the flags in T_i are set. Flags in S_i are set while $stmt_i$ is executing. In addition, each $m.f$ flag is set while some function in medium m from interface f is being executed.

5 Annotating and Restricting Network Executions

Annotations are typically used to represent cost of performing certain computations on a given architecture, e.g. delay or energy information. In the meta-model, annotations are defined with a concept called *quantity*, i.e. for each annotation, there is a quantity with the same name. Each quantity has an associated type, for example, `double` for time. In the meta-model, for each quantity there is an object called a *quantity manager* that is responsible for assigning annotations to events. Code executed by a process can make a *request* for a specific annotations. For example, to model that the delay between two events e_1 and e_2 is 10, the process makes a request that a time-stamp of e_2 must be equal to the time-stamp of e_1 plus 10. It is the responsibility of a quantity manager to collect all requests and satisfy them. If a request cannot be satisfied, the manager must disable the event for which that request was made. For example, if one process wants to execute event e_1 and request for it time-stamp 10, and the other process wants to execute e_2 with time-stamp 20, time manager must set the current time to 10, let e_1 occur, and disable e_2. In the meta-model, all the objects related to quantities are grouped in a separate network called the *scheduling network*. In contrast, the network containing all "ordinary" objects is called the scheduled network.

In the meta-model semantics, annotations are represented as a set of *annotation functions* $A_1, A_2, \ldots, A_i, \ldots$ that are associated with network execution as in Equation 2. For each event $e \in \sigma_i$, and each annotation name f, the annotation function $A_i(e, f)$ holds the value of annotation f of event e.

As explained earlier, the semantics of the scheduled network is described as a sequence of state transitions. In every state there is a set of enabled events and possibly a set of annotation requests. Execution of the scheduling network determines which events should actually occur, and what their annotations should be. In other words, execution of the scheduling network disables some of the enabled events, and annotates the rest.

In addition to restricting network executions by scheduling networks, the meta-model provides a declarative alternative. The user may use certain logic formulas over sequences of state transitions to express constraints. In this way, a given sequence of annotated state transitions is a legal behavior of a meta-model restriction if and only if (1) it can be generated by the execution of the scheduled network restricted by the scheduling network, (2) it satisfies all the constraints specified by logic formulas. In the rest of this section we explained both of these mechanisms in more detail.

5.1 Quantities and Scheduling Network

In contrast to ordinary ("scheduled") networks, scheduling networks have some methods associated with them. Boolean function `ifTop()` returns true if and only if that network is at the top level, i.e. it is not contained in any other *scheduling* network. The user should not redefine it. The default function `resolve()` of type `void` recursively calls the `resolve()` function of all the subnetworks. The user may redefine it. One typical way to redefine it is to keep calling the `resolve()` function of all quantity managers in its scope until a fixed point is reached. The default function `postcond()` of type `void` recursively calls the `postcond()` function of all the subnetworks. The user may redefine it.

The scheduling network is executed each time the scheduled network reaches a state in which it is about to execute an event for which there is an associated request. A scheduling network consists of the following stages:

1. **request making:** In this phase annotation requests for the enabled events are issued to the scheduling network.

2. **negotiation:** In this phase, the `top` function of the topmost scheduling netlist is executed. This will typically result in iterative calling of `resolve` functions of quantity managers.

3. **selection:** In this phase, the number of enabled events is reduced to one per process. No code that a user can modify is executed in this phase. In practical terms, the simulator looks at all enabled events, looks at all the active constraints, and selects a vector of enabled events that is consistent with constraints.

4. **annotation:** In this phase the `postcond` function of the topmost scheduling netlist is executed. This typically results in calling `postcond` functions of all quantity managers, which in turn results in making the annotation and doing any clean-up

that may be required. The default version of a quantity manager's `postcond` leaves all events with undefined annotations, i.e. only user provided versions will actually do the annotation.

5.2 Constraints

There are two basic types of constraints in the meta-model, and each type is expressed in a different logic. Coordination constraints from are represented by formulas of *linear temporal logic (LTL)*, quantity constraints are represented in the logic called *logic of constraints (LOC)*.

Given a meta-model netlist, and its execution as in Equation 2 we interpret LTL formulas over the sequence:

$$(s_0, \sigma_1), (s_1, \sigma_2), \ldots, (s_i, \sigma_{i+1}), \ldots \ .$$

In the meta-model, the atomic LTL propositions are either Boolean meta-model expression, or expressions of the form `beg(p,f)` or `end(p,f)`, where `p` is a process identifier, and `f` is an action name. We say that (s_i, σ_{i+1}) satisfies a meta-model expression if it evaluates to *true* in state s. We also say that (s_i, σ_{i+1}) satisfies `beg(p,f)` if it appears in σ_i. Similarly, we say that (s_i, σ_{i+1}) satisfies `end(p,f)` if it appears in σ_i. This defines the semantics of atomic LTL formulas. The semantics of all other LTL constructs is defined in the standard way [7].

We have designed the other meta-model logic, LOC, to meet the following goals, which we believe are essential for a constraint specification formalism to gain wide acceptance:

- it must be based on a solid mathematical foundation, to remove any ambiguity in its interpretation,
- it must feel natural to the designer, so that typical constraints are easy to specify,
- it must be compatible with existing functional specification formalisms, so that language extensions for constraint specification can be easily defined,
- it must be powerful enough to express a wide range of interesting constraints,
- it must be simple enough to be analyzable, at least by simulation, and ideally by automatic formal techniques.

LOC formulas are defined relative to a multi-sorted algebra $(\mathcal{A}, \mathcal{O}, \mathcal{R})$, defined by the meta-model type system. In the meta-model, the set of sets (sorts) $\mathcal{A}$ includes value domains of all the types in the meta-model. In general, $\mathcal{O}$ is a set of operators, and $\mathcal{R}$ is a set of relations on sets in $\mathcal{A}$. More precisely, elements of $\mathcal{O}$ are functions of the form $T_1 \times \cdots \times T_n \mapsto T_{n+1}$, where n is a natural number, and $T_1, \ldots, T_{n+1}$ are (not necessarily distinct) elements of $\mathcal{A}$. If $o \in \mathcal{O}$ is such a function, than we say that o is n-ary and T_{n+1}-valued. Similarly, an n-ary relation in $\mathcal{R}$ is a function of the form $T_1 \times \cdots \times T_n \mapsto \{true, false\}$. In the meta-model, $\mathcal{R}$ contains all the relational and Boolean operators, and $\mathcal{O}$ contains all other meta-model operators.

LOC formulas may contain only one variable, namely i. The domain of i is the set of integers. Having only one variable may seem very restrictive, but so far we have not found a natural constraint that required more than one. The advantages of a single variable are simpler syntax (fewer names), and simpler simulation monitoring.

The basic building blocks of LOC formulas are *terms*. We distinguish terms by their types:

- i is a integer-valued term,
- for each value domain $T \in \mathcal{A}$, and each constant $c \in T$, c is a T-valued term,
- if τ is a integer-valued term, e is an event name, and f is a name of a T-valued annotation, $f(e[\tau])$ is a T-valued term,
- if $o \in \mathcal{O}$ is a T-valued n-ary operator, and $\tau_1, \ldots, \tau_n$ are appropriately valued terms, then $o(\tau_1, \ldots, \tau_n)$ is a T-valued term.

Terms are used to build *LOC formulas* in the standard way:

- if $r \in \mathcal{R}$ is an n-ary relation, and $\tau_1, \ldots, \tau_n$ are appropriately valued terms, then $r(\tau_1, \ldots, \tau_n)$ is an LOC formula,
- if ϕ and ψ are LOC formulas, so are $\overline{\phi}$, $\phi \wedge \psi$, and $\phi \vee \psi$.

For example, if `a` and `b` are event names, and `f` and `g` are integer-valued annotations, then the set of LOC formulas includes the following:

```
    f(a[i])==5 && g(a[i+1])==5  ,
    f(a[i-4])+f(b[g(a[i])])<20  ,
    !(f(a[i])==0) || f(b[i])==0  .
```

When reading these formulas, it is helpful to think of `i` as being universally quantified, as clarified in the LOC semantics next.

We interpret LOC formulas over network executions defined as in Equation 2 with associated annotation functions $A_1, A_2, \ldots, A_i, \ldots$. We first define the *value* of LOC terms and formulas as follows:

Table 1. Extensions of Boolean operators to undefined values.

x	y	x&&y	x||y	x->y	x<->y
$true$	$\perp$	$\perp$	$true$	$\perp$	$\perp$
$false$	$\perp$	$false$	$\perp$	$true$	$\perp$
$\perp$	$true$	$\perp$	$true$	$true$	$\perp$
$\perp$	$false$	$false$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

- The value of variable i may be any integer.
- If τ is a term with integer value n, e is an event, and f is an annotation name, then the value of $f(e[\tau])$ is $A_i(e, f)$, where i is such that e appears for the n-th time in the sequence exactly in σ_i, i.e. $e \in \sigma_i$, and the set $\{j \leq i : e \in \sigma_j\}$ has exactly n elements. If n is not positive, e appears in the sequence less than n times, or $A_i(e, f)$ is undefined, then the value of $f(e[\tau])$ is undefined.
- The value of all the operators in O, as well as all the relational operators in R is defined as in the meta-model, provided that all their sub-terms have defined values. If that is not the case, the value is undefined.
- The value of Boolean operators in R is determined as in the standard three-valued extension of Boolean logic. In this extension, the negation of an undefined value is undefined, and the extensions of other operators is summarized in Table 1 where $\perp$ denotes an undefined value.

Finally, we say that a network execution with annotation functions *satisfies* an LOC formula if the value of the formula is either *true* or undefined for all possible values of i.

6 Conclusions

We presented briefly a formal system level design methodology, platform-based design, and some aspects of the Metropolis environment that supports the methodology. In particular, we focused on the modeling strategy of the environment. Metropolis is based on a meta-model with formal semantics that can be used to capture designs from specification languages and to support simulation, formal analysis and synthesis. The framework is conceived to encompass different application domains, and therefore supports heterogeneity through the orthogonalization of function and architecture and of communication and computation. The formal semantics of the meta-model allows embedding of models of computation in a rigorous framework thus favoring design reuse and design chain support.

The environment also offers a set of analysis and synthesis tools that are examples of how the framework can be used to integrate flows. At this time, we are exploring the automotive, wireless communication and video application domains in collaboration with our industrial partners. As we understand better what the critical parts of the design are and what needs to be supported to facilitate design hand-offs, we plan to tune the meta-model and to increase the power of the infrastructure for the support of successive refinement as a major productivity enhancement.

References

[1] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *Interntional Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[2] L. P. Carloni, F. D. Bernardinis, A. L. Sangiovanni-Vincentelli, and M. Sgroi. Platform-based and derivative design. In R. Zurawski, editor, *The Industrial Information Technology Handbook*, pages 1–15. CRC Press, 2005.

[3] D. Gajski, J. Zhu, and R. Domer. *The SpecC language*. Kluwer Academic Publishers, Boston; Dordrecht, 1997.

[4] T. Grotker, G. Martin, S. Liao, and S. Swan. *System design with SystemC*. Kluwer Academic Publishers, Boston; Dordrecht, 2002.

[5] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, Dec. 2000.

[6] R. Passerone. *Semantic Foundations for Heterogeneous Systems*. PhD thesis, Department of EECS, University of California at Berkeley, May 2004.

[7] A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual IEEE Symposium on Foundations of Computer Sciences*, pages 46–57, 1977.

[8] A. Sangiovanni-Vincentelli, L. Carloni, F. D. Bernardinis, and M. Sgroi. Benefits and challenges for platform-based design. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 409–414, New York, NY, USA, 2004. ACM Press.

[9] S. Solden. Architectural services modeling for performance in HW-SW co-design. In *Proceedings of the Workshop on Synthesis And System Integration of Mixed Technologies SASIMI2001, Nara, Japan, October 18-19, 2001*, pages 72–77, 2001.

[10] The Metropolis Project Team. The metropolis meta model. Technical report, Technical Memorandum UCB/ERL M04/38, University of California Berkeley, CA 94720, 2004.