The Metropolis Meta Model
Version 0.4

The Metropolis Project Team

Technical Memorandum UCB/ERL M04/38
University of California, Berkeley, CA 94720, September 14, 2004

Contents

Chapter 1

Introduction and Core Syntax

This document presents the syntax and semantics of the Metropolis meta-model, the mechanism employed internally in the Metropolis design environment to represent design behavior and constraints. For a more detailed overview of the concepts underpinning metropolis, and how to use it please refer to the design guidelines in [4]. The meta-model does not commit to the semantics of any particular model of computation. Rather, it provides building blocks necessary to define a *computation* and *communication* semantics, so that many models of computation often employed in system designs can be represented with this single set of building blocks.

The Metropolis meta-model describes designs using networks. A network instantiates components of the network and specifies their connections. Once a network is specified, the execution semantics defines the *behavior* of the network. The first section lists what are the general operators and constructs from Java are usable in the metamodel. The second section explains the creation of general classes in the metamodel. The third section explains the different data types supported in the metamodel. The fourth section explains the packages used in the metamodel, and the use of the package statement.

Chapter 2 explains the different elements of the network, and how to create them. Chapter 3 explains the execution semantics of the metamodel, and specialized execution statements, like await and bounded loop. Chapter 4 presents mechanisms for *annotating* the behavior of a network with constraints and quantities. This is helpful when one wants to specify certain properties with the behavior, such as the time it takes or energy required for executing the behavior. The behavior of a network may be related with the behavior of another network. With this mechanism, a design may be specified as a related set of independent subsystems, and this plays the central role for specify designs in consistent manner over multiple levels of abstraction. This is also helpful for maintaining the reusability of descriptions of the subsystems. Section 4.5 describes this mechanism.

The syntax of the meta-model is similar to that of Java [2], although we employ restriction in its usage and also have introduced additional keywords; the execution

semantics is also different. When we present keywords that are also available in Java, we explain them only if their meaning or usage is different in Java in the context.

1.1 General Constructs and Operators

General constructs and operators refer to those defined in Java and supported by the meta-model:

- looping constructs: while, do-while, for
- decision making constructs: if-else, switch-case
- branching constructs: break, continue, return
- arithmetic operators: +, -, ∗, /, %
- relational operators: >, >=, <, <=, ==, !=
- conditional operators: &&, ||, !, &, |, ∧
- shift and logical operators: ≪, ≫, &, |, ∧, ∼
- assignment operators: =, +=, -=, ∗=, /=, %=, &=, |=, ∧=, ≪=, ≫=
- dynamic allocation operator: new

Note that not all constructs and operators in Java are supported by the meta-model. For example, synchronized is not supported. See Appendix A for Java keywords not supported in the meta-model. In addition, the meta-model supports two conditional operators for convenience: -> and <->. They represent the logical implication and logical equivalence respectively. Further, the meta-model supports special constructs and operators defined in the following section.

1.2 Regular Classes

This section explains the creation of regular classes in the Metropolis metamodel, which have a very similar syntax to those declared in Java and C++. This syntax will be used as a basis for the syntax of `processes`, `media`, `quantities`, and other metamodel objects.

1.2.1 Declaration

The syntax to declare a regular class is:

```
modifier class ClassName extends SuperClassName implements InterfaceName {
  // the body of this class declaration
}
```

- **modifier** must be either unspecified, or one of the following keywords: `public`, `protected`, `private`, `abstract`, `static`, `final`. The meanings of these modifiers (and the unspecified modifier) are the same as they are in Java.
- **ClassName** is the identifier of this class.
- **SuperClassName** is the identifier of a class from which this class is derived. Note that `ClassName` and `SuperClassName` must be different.
- **InterfaceName** is a comma-separated list of the identifiers of interfaces implemented by the class. For each identifier in the list, it is required that this class declares all the members of the interface in its body[1].

1.2.2 Fields

If a field is declared `static`, it must also be `final`, i.e. it has to be a constant only. This is to prevent implicit communication through static fields. Constants defined in classes can be accessed only from constructors or postElaborate functions. Fields in regular classes should only have the type of regular classes or primitive data types (and arrays of these).

For all the fields of a class, it is not allowed to have two fields with the same name. This is the case even if one field is transitively inherited from another process while the other is declared in the current process[2].

1.2.3 Constructors

The following is the syntax of a regular class constructor:

```
modifier RegularClassName(args){
  // the body of this constructor declaration
}
```

The `modifier` must be either unspecified, or one of the following keywords: `public`, `protected`, `private`. Its accessibility is same as Java. `ProcessName` is the identifier of a process in which this constructor is defined. `args` is a comma-separated list of additional arguments that may be empty. We support overloading, and more than one constructor may be defined, as long as their lists of arguments are not identical; two lists of arguments are said to be *identical* if the numbers of the arguments are equal and

[1] Note that these interfaces are not `port interfaces`, which are described in the next chapter.
[2] This differs from Java and C++ where it is is legal

the types at the i-th arguments are same for each i. As with Java, one may specify the `super` method at the very first line of the body of a constructor declaration. If it is not specified, then it is assumed that the method is called with `InstanceName` being the only argument. The constructors are not inherited by subclasses.

1.2.4 Methods

The methods declaration has to be in the following syntax:

```
modifier type methodName(args) {
  // the body of this method
}
```

The `modifier` must be one of the following keywords: `public`, `protected`, `private`, `abstract`, `static`, `final`. The `type` should be a primitive type or some type derived from a regular class [3]

Compared with methods defined in media, in classes, methods do not have any effect modifiers. This implies that no port connection can be made to classes. Since classes do not have ports either, there cannot exist port connections going out of classes. However, regular interfaces can be implemented by regular classes, therefore, it is possible to assign a regular class to a variable of a regular interface type as long as the class implements the regular interface or its sub-interfaces.

1.3 Types

The meta-model adopts the Java type system. It uses a statically typed system; that is, all variables must be associated with a type when they are declared. All types fall into one of the following four categories: primitive type, compound type, derived type, and type parameters. Further, the meta-model type system is inheritance-based.

1.3.1 Primitive Types

Primitive types are built-in to the meta-model, see Appendix A.1 for the primitive types supported in the meta-model.

1.3.2 Compound Types

Compound types are created from primitive types and are defined as classes. Compound types defined in Java that are also supported in the meta-model are String, Array, List, Set, HashSet, HashMap and Hashtable. These types and a set of methods associated

[3] It may be possible to have metamodel objects (e.g. ports, media), but this violates the purpose of the regular class.

with each of them can be directly used in the meta-model (See [3] for specific methods of these classes).

Other than these compound types, the meta-model does not support classes defined in Java. However, it allows users to define their own compound types by defining appropriate classes.

1.3.3 Derived Types

All classes (regular, process, media, netlist, etc), and port-interfaces are data types called derived types.

1.3.4 Templates

The declaration of a template object or interface is the same as that of an ordinary one, but prefixed by the following declaration:

Syntax 1.3.1

template (< *TypeParameter* >)

where *TypeParameter* consists of a name *ParameterName* that is used in the place of the type of variables or arguments in the template, and optionally a list of legal types that *ParameterName* can be assigned to [4]. It takes the following form:

Syntax 1.3.2

ParameterName << : *LegalType* >>

All *LegalType*s must follow a colon. If more than one *LegalType*s are listed, they must be separated by a a comma. A template may have more than one *TypeParameter*s, in which case they should be separated by a semicolon.

For example, suppose that in an instance of Mem that types m of SIZE, START_ADDR, and storage inside Mem are changed from int, int and byte[] to type parameters size_t, addr_t, and storage_t[] respectively. Suppose also that the legal type of size_t and addr_t is int, while storage_t may be either int or byte. Then the declaration of Mem can be changed to the following:

[4] In the current Metropolis infrastructure, standard usages of templates as in C++ are supported, but the legal type list is not yet supported.

Example 1.3.1

```
template ( size_t: int; addr_t: int; storage_t:int,byte )
medium Mem implements MemAPI {
    ...
    parameter size_t SIZE;
    parameter addr_t START_ADDR;
    int first;
    int last;
    storage_t[] storage;

    Mem (size_t size, addr_t start_addr) {
        SIZE = size;
        START_ADDR = start_addr;
    }
    public update void memWrite (byte[] dest, byte[] src){ ... }
    public eval void memRead (byte[] dest, byte[] src){ ... }
    public constant int memSize (){ ... }
    ...
}
```

When a template object is instantiated, all of its type parameters must be defined, as follows:

Syntax 1.3.3

ObjectType -< < *ParameterType* > >-

where *ParameterType* is a primitive, compound, derived data type, or a type parameter. If there are more than one *ParameterType*s, they must be separated by a semicolon, and must be listed in the same order as *TypeParameter*s given in the template of *ObjectType*. This order defines the correspondence between the *ParameterType*s and *ParameterName*s. If *LegalType*s are specified for a given *ParameterName* in the template, its *ParameterType* is considered legal if the following conditions hold. If *ParameterType* is a primitive type, the type must be included in the list of *LegalType*s. If *ParameterType* is a compound or derived type, there must exist a *LegalType* for which it is legal in Java that the *ParameterType* can be assigned to a variable of the *LegalType* [5]. If *ParameterType* is a type parameter, it is always legal. Further, if no *LegalType* is specified in the template for the *ParameterType*, the latter is always legal. -< and >- are part of the syntax, and must appear in the specification.

For example, the template *Mem* can be instantiated as:

[5] This roughly means that the *ParameterType* is a subclass of the *LegalType*, while the precise definition is given in [2].

Example 1.3.2

Mem -< int; int; byte >- *mem* = new *Mem*-< int; int; byte >-(100, 10);

A subclass can inherit from a template class, where each of the type parameters of the template class can be either instantiated to a specific type, or left as a type parameter. If the subclass still has type parameters, it has to be defined as a template class. Otherwise, it becomes a concrete class:

Example 1.3.3

```
// Mem1 is a template class derived from Mem
template ( size_t; addr_t; storage_t )
medium Mem1 extends Mem-< size_t; addr_t; storage_t>- {
    ...
}

// Mem2 is another template class with specific types for size_t and addr_t
template ( storage_t: byte )
medium Mem2 extends Mem-<int; int; storage_t>- {
    ...
// Mem3 becomes a concrete class
medium Mem3 extends Mem-<int; int; byte>- {
    ...
}
```

Here, Mem1 is declared without specifying legal types. It is interpreted that the legal types of its three type parameters are same as those specified in Mem, the superclass of Mem1. It is possible to specify only a subset of the original legal types as legal types in the extended template. For example, byte is the only legal type for storage_t in Mem2.

The same mechanism can be used for interfaces. For example, in Example 1.3.4, the type of dest and src inside MemAPI can be parameterized as follows:

Example 1.3.4

```
template (data_t)
interface  MemAPI extends Port {
    update void memWrite(data_t[] dest, data_t[] src);
    eval void memRead(data_t[] dest, data_t[] src);
    constant int memSize();
}
```

When a port is declared with this interface, the type has to be specified. However, since legal types are not specified, data_t can be defined with any type.

Example 1.3.5

port *MemAPI* -< byte >- port0;

In defining a class that implements an interface with type parameters, one can either instantiate each parameter with a specific type or leave it as a parameter; if some type parameters are left, the class has to be defined as a template class. In the following example, the type parameter of the interface MemAPI is instantiated with the byte type when implemented by the medium Mem.

Example 1.3.6

template (*size_t:int*; *addr_t:int*; *storage_t:int, byte*)
medium *Mem* **implements** *MemAPI*-<byte>- {
    ...
    public **update** void *memWrite*(byte[] *dest*, byte[] *src*){ ... }
    public **eval** void *memRead*(byte[] *dest*, byte[] *src*) { ... }
    ...
}

Alternatively, it is possible to implement MemAPI without instantiating the type parameter. In this case, the type parameter has to be added in the type list of the template:

Example 1.3.7

template (*size_t:int*; *addr_t:int*; *storage_t:int, byte*; *data_t*)
medium *Mem* **implements** *MemAPI*-<*data_t*>- {
    ...
    public **update** void *memWrite*(*data_t*[] *dest*, *data_t*[] *src*){ ... }
    public **eval** void *memRead*(*data_t*[] *dest*, *data_t*[] *src*) { ... }
    ...
}

1.3.5 Polymorphism

The meta-model supports only static polymorphism. This is done through the use of template objects, where types of type parameters are defined at instantiation time (see Example 1.3.2). Even in the case of template interface, the types of the arguments still must be determined statically (at compile time).

1.4 Packages

In metamodel, packages are used to organize design entries. This is the same concept as in Java. One package corresponds to one directory in the file system. Every sub-packages in the same package must occupy one sub-directories, and so on. The hierarchy of the packages reflects exactly the same hierarchy of the file system. It is a requirement that all files (metamodel object descriptions) in the same package must be put in the same package therefore in the same directory. At the beginning of each file, it must declare the package that all objects in this file belong to. The syntax to declare a package is

Syntax 1.4.1

package *PackageName*[.*Sub-PackageName*]*;

Here, *PackageName*[.*Sub-PackageName*]* is the fully qualified name of the package, i.e. it shows every intermediate hierarchies and in the same order from the top level package to the current level package. For instance, if the top level package is pkg0, the current file is in a sub-package of pkg0 with the name pkg1, then the package declaration would be package pkg0.pkg1;

Though packages are a good way to organize design entries clearly, it is not required that every file must declare thus belong to a package. In case no package declaration exists, all objects defined in the file belong to a default package called `UNNAMED_PACKAGE`. Note that `UNNAMED_PACKAGE` is just for compilation purpose. User can not explicitly refer to it.

Besides `UNNAMED_PACKAGE`, metamodel also pre-defines other packages:

- `metamodel.lang` is in metro/lib/metamodel/lang directory. It defines all the basic metamodel language object prototypes, which participate in the syntax and semantics checks during compilation.
- `metamodel.util` is in metro/lib/metamodel/util directory. It provides a set of utility classes resembling those in Java util package. Right now, this package includes ArrayList, BitSet, Collection, HashMap, HashSet, Hashtable, LinkedList, SortedMap, SortedSet, TreeMap, TreeSet and Vector.
- `metamodel.plt` is in metro/lib/metamodel/plt directory. It provides a collection of communication platforms, architecture platforms etc.
- `SYSTEM_PACKAGE` is a virtual package. It serves as the root package of all other top level packages. Its existence simplifies the package management in compilation process. Users cannot refer to it explicitly.

In order to use those defined in other packages, a keyword `import` must be used. In a metamodel file, `import` statements must follow the package declaration if there is one. The syntax of `import` is

Syntax 1.4.2

import *PackageName*[.*Sub-PackageName*]*.*ClassName*;

or

Syntax 1.4.3

import *PackageName*[.*Sub-PackageName*]*.*;

The first syntax is to import a particular object definition from a package. For example, `import` metamodel.util.LinkedList; imports the class LinkedList defined in metamodel.util package. The second syntax is called import-by-demand. It specifies only the package name, not the specific class name. Later on, when a class is used and it does not exist in the current package or other explicitly imported classes, import-by-demand comes into play. It will search that package to see whether there is such a class. If so, import it. For example, `import` metamodel.util.*; If in the following metamodel code, LinkedList is used, metamodel.util package will be searched. Once the definition of LinkedList is found, it will be imported.

Note that metamodel.lang is imported by default. All other packages including metamodel.util, metamodel.plt and user defined ones must be imported before use.

Chapter 2

Creation Of Networks

The components of a network must be of the following types: `process`, `medium`, `quantity`, `statemedium`, `netlist`. These types are explained in this section.

2.1 Processes

In metropolis `process` is an active element, in that it has its own thread of control. A process can have one or more `ports`, which are used to communicate to other processes via `media`.

A process is declared with a keyword `process`, and has the syntax below:

```
modifier process ProcessName extends SuperProcessName {
  // the body of this process declaration
}
```

The `modifier` must be either unspecified, or one of the following keywords: `public`, `protected`, `private`, `abstract`, `static`, `final`. If it is unspecified, the process has default visibility, which is accessible only to all defined in the same package, and it is non-abstract, non-static and non-final. `ProcessName` is the identifier of this process, and `SuperProcessName` is the identifier of a process from which this process is derived. `ProcessName` and `SuperProcessName` must not be literally identical.

The body of a process declaration consists of three parts: fields, constructors, and methods.

2.1.1 Fields

The fields can be divided into four kinds: `regular fields`, `port fields`, `constant fields`, and `parameter fields`. Regular fields are the same as those allowed in regular classes.

2.1.1.1 Port Fields

A port field is declared with a keyword `port`:

```
port modifier InterfaceName PortName;
```

- **modifier** must be either public or unspecified.
- **InterfaceName** is the identifier of a port interface, and is the type of the declared port.
- **PortName** is the identifier of this port field.

Ports are used to connect components and components can exchange information through ports. Port interfaces are explained in Section 2.2. A port field may be accessed either in the methods within the process in which the field is declared, or using a meta-model keyword `connect`, explained in Section 2.6. It is not accessible to the constructors of this process.

One may declare multiple fields with a single `port` declaration, by providing multiple identifiers, separated by comma:

```
port InterfaceName PortName0, PortName1;
```

In this case, `InterfaceName` is the type for all the declared fields.

One may also declare an array of ports. As with Java, this can be done either of the following ways:

```
port InterfaceName PortName0, PortName1[], PortName2[][];
port InterfaceName[] PortName3, PortName4[];
```

In the first case, `PortName0` declares a single port field, `PortName1[]` declares an one-dimensional array of port fields, and `PortName2[][]` declares a two-dimensional array of port fields. `InterfaceName` is the type for all the port fields. The second case provides an alternative way to declare the one-dimensional and two-dimensional port fields, respectively.

2.1.1.2 Constant Fields

Constant fields are common to all the instances of this process. They are specified with the keyword `final`[1], and have the syntax below:

```
modifier final type ConstantName = Initializer;
```

[1]Unlike Java, the `static` keyword is not used, because we require an instance of a field for each component instantiated in a network.

- **modifier** must be either unspecified or the keyword `private`. If it is unspecified, the field is inherited by subclasses of this process.

- **type** is can be one of the primitive types described in Section 1.3.1 or an array of primitive types, and is referred to as the type of this field.

- **ConstantName** is the identifier of this field.

- **Initializer** sets the value of this field. The `Initializer` is either an expression or an array initializer, and is subject to exactly the same rules as Java [2].

A constant field is accessible only within the process in which the field is declared. By definition, it is assigned exactly once. It is important to note that this grammar does not completely guarantee that this field can be always a constant. That is, if the field holds a reference to an object, then although the value of the field will always be the reference, the contents of the referred object may change. It is subject to the use policy whether constant fields are treated as constants.

As with port fields, one may declare more than one field and arrays of fields, as follows:

```
final int AREA = 20, SAMPLE[] = {0, 5, 15};
```

2.1.1.3 Parameter Fields

Constants whose values may be specific to particular instances are called *parameter fields*. They are declared with a `parameter` keyword:

```
modifier parameter type ParameterName;
```

A parameter field is accessible only within the process in which the field is declared, and may be assigned only inside constructors or in a special method called `postElaborate` that is explained in section 2.1.3.1. Unlike constant fields, parameter fields may not have initializers. The `modifier` must be either unspecified or a keyword `private`. If it is unspecified, this field is inherited by subclasses of this process. `ParameterName` is the identifier of this field. One may declare more than one field and arrays of fields in the same way as port fields.

Parameter fields, are declared in the same way as constant fields, with two exceptions. First, the `final` keyword must not be used. Second, the use of `Initializer` is optional and thus may not be used. The accessibility and inheritance are also same. The reason for this rather limited accessibility is to force the communication to be specified explicitly through the port fields. This aspect is explained more in Section 2.3. These fields may be assigned both in constructors and in the methods.

2.1.2 Constructors

The following is the syntax of a constructor:

```
modifier ProcessName(String InstanceName, args) {
  // the body of this constructor declaration
}
```

- **modifier** must be either unspecified, or one of the following keywords: `public`, `protected`, `private`. Its accessibility is same as Java.
- **ProcessName** is the identifier of a process in which this constructor is defined. The meta-model requires that a constructor of a process must have the `String` type at the first argument. This is used as the name of an instance created by the constructor.
- **args** is a list of additional arguments that may be empty. As with the ordinary case, the arguments are separated by comma, with each made of the type and the identifier of the argument separated by spaces.

We support overloading, and more than one constructor may be defined, as long as their lists of arguments are not identical; two lists of arguments are said to be *identical* if the numbers of the arguments are equal and the types at the i-th arguments are same for each i. As with Java, one may specify the `super` method at the very first line of the body of a constructor declaration. If it is not specified, then it is assumed that the method is called with `InstanceName` being the only argument. The constructors are not inherited by subclasses.

2.1.3 Methods

A method is accessible only within the process in which it is declared. The following is the declaration syntax:

```
modifier Effect type methodName(args) useport PortNames{
  // the body of this method declaration
}
```

- **modifier** must be either unspecified or one of the keywords `public`, `protected`, `private`, `abstract`, `static` and `final`. If it is unspecified, this method is inherited by subclasses of this process, which may be overridden by the subclasses.
- **type** is the result type, either one of the types described in Section 1.3 or a keyword `void`. `methodName` is the identifier of this method and `args` is a list of arguments.

- **useport**[2] is a keyword which may be optionally present in a method declaration. If used it is followed by a comma-separated list of port identifiers that might be accessed during execution.

Overloading is supported for methods in processes, i.e. methods with the same identifier may exist in a process if their lists of arguments are not identical.

2.1.3.1 Special Methods

There are two special kinds of methods in processes: `thread` and `postElaborate`.

The `thread` method of a process defines the behavior of the process, explained in detail in Chapter 3. Any process must have exactly one method with the identifier `thread`. The result type has to be `void` and the argument list has to be empty.

The special process methods is `postElaborate`. The meta-model compilation has a phase called *elaboration*, used for creating networks that constitute a design; Section 6.2 briefly explains it. The `postElaborate` method is called in this phase for each component of a network being created, if the method is specified in the component. This method must have the `void` result type with the empty list of arguments. This method has a particular rule on accessing the fields: the parameter fields of a process may be assigned, and the port fields may not be accessed.

2.1.4 A Process Example

We conclude this subsection with a simple example of a process.

```
public process DualProducer extends StreamProcess {
  port Reader  input;
  port Writer mode, data;
  final int REF = 1;
  parameter int hirate, lowrate;
  parameter int stream_size;
  byte[] stream;

  public DualProducer(String instanceName) {
    super(instanceName);
  }
  public DualProducer(String instanceName, int hi, int low, int size) {
    hirate = hi;  lowrate = low;  stream_size = size;
    stream = new byte[stream_size];
  }
```

[2]It is wise to not rely on useport in writing a backend, because a useport list can be quickly derived from a method, and manual entry can lead to errors.

```
  void thread() useport input, mode, data {
    // the body of this thread declaration
  }
}
```

The example is an extension of the `StreamProcess` and has 3 ports, a Reader `input`, and 2 Writer's named `mode` and `data`. It also has parameters for the rates and the size of stream, and a field for holding the byte steam. Finally, it has two constructors and a thread method.

2.2 Port Interfaces

Port interfaces are a special kind of interfaces that declare methods which can be used through ports. These methods would be implemented by a medium, and called by a process or another medium.

The syntax is as follows:

```
public interface InterfaceName extends SuperInterfaceName {
  // the body of this port interface declaration
}
```

- **interface** declares a port interface if `SuperInterfaceName` is either the identifier of a port interface, or `Port`, which is the identifier of a built-in interface in the meta-model.
- **InterfaceName** is the identifier of the declared port interface, which is said to be a subclass of `SuperInterfaceName`.

The body of a port interface declaration is either empty, or else consists of declarations of methods. Unlike interfaces in Java, a port interface may not declare fields. We say that a method is a member of a port interface if either it is declared in the body of the port interface declaration or it is a member of `SuperInterfaceName`. The built-in interface `Port` has no method as its member. The following is the syntax of method declarations:

```
effect type methodName(args);
```

- **effect** must be one of the following keywords: `update`, `eval`, `constant`. These keywords indicate how the declared method accesses and modifies the values of the fields in the classes that implement this port interface.
  - **update** indicates that for each class that implements this port interface, the method may access all the field of the class and change their values.

  - **eval** means that the method may access all the fields of the class but may not change their values. If a method is declared with the keyword
  - **constant** indicates the method may only access the constant and parameter fields of the class.[3]

- **methodName** is the identifier of this method.
- **type** is the return type.
- **args** is the list of arguments.

A port interface may have two methods with the same identifier, if their lists of arguments are not identical.

The following example declares two port interfaces:

```
public interface Writer extends Port {
  eval int querySpace();      // returns the number of available slots
  update void write(int[] data, int num);
  update void write(double[] data, int num);
}

public interface Reader extends Port {
  eval int queryData();      // returns the number of available data
  update void read(int[] data, int num);
  update void read(double[] data, int num);
}
```

2.3 Media

In the metamodel language a medium is a passive object that is used for communication and holding state. It can implement `port interfaces` and be connected to processes and other media.

A medium is declared with a keyword `medium`, and has the syntax below.

```
modifier medium MediumName extends SuperMediumName implements InterfaceName {
  // the body of this medium declaration
}
```

- **modifier** must be either unspecified, or one of the following keywords: `public`, `protected`, `private`, `abstract`, `static`, `final`. They are used in the same way as in process declarations.

[3]In fact, there is another keyword `elaborate` for effect. However, it is not for describing how the function affects the state of the object. When a function is annotated with `elaborate`, it means that it will be handled by elaboration phase, otherwise, the function will be ignored. More details in section 6.2.

- **MediumName** is the identifier of this medium.
- **SuperMediumName** is the identifier of a medium from which this medium is derived. `MediumName` and `SuperMediumName` must be different.
- **InterfaceName** is a list of the identifiers of interfaces, separated by comma. For each identifier in the list, it is required that this medium declares all the members of the interface in its body. We say that these interfaces are implemented in this medium.

The body of a medium consists of fields, constructors, and methods, as with the case of processes. Among them, fields and constructors are subject to the same rules described for processes in Section 2.1.

2.3.1 Methods

The methods are also declared in the same way as in processes, with the following two exceptions. First, it is not mandatory to declare a method with its identifier being `thread`. It is not illegal for a medium to declare a `thread` method, but it doesn't mean anything and isn't recommended. Secondly, when a member of a port interface is declared, the declaration has to be in the following syntax:

```
public effect type methodName(args) useport PortNames{
  // the body of this method
}
```

The `effect` must be the same as the one declared in the port interfaces implemented in this medium.

2.4 Quantities and Quantity Managers

Quantities can be physical numbers like time and power, or they can be used for something like the arbitration of a shared resource. Quantity managers handle a particular quantity, how it is modified, and who gets access to it. This section describes the syntax of declaring quantities and quantity managers, for an explanation of how they are used see chapter 4.

A quantity is declared with a keyword `quantity` and has the syntax below:

```
modifier quantity QuantityName extends SuperQuantityName
                                  implements InterfaceName {
  // the body of this quantity declaration
}
```

- **modifier** must be either unspecified, or one of the following keywords: `public`, `protected`, `private`, `abstract`, `static`, `final`. They are used in the same way as in process and medium declarations.
- **QuantityName** is the identifier of this quantity.
- **SuperQuantityName** is the identifier of a quantity from which this quantity is derived, and is optional. `QuantityName` and `SuperQuantityName` must be different.
- **InterfaceName** is a list of the identifiers of interfaces, separated by comma. For each identifier in the list, it is required that this quantity declares all the members of the interface in its body. We say that these interfaces are implemented in this quantity.

Note that quantities by default implement an interface called `QuantityManager`, which includes has the syntax below with four methods:

```
public interface QuantityManager extends Port {
    eval void request(event e, RequestClass rc);
    update void resolve();
    update void postcond();
    eval boolean stable();
}
```

The `request` function is to request a quantity annotation(rc) for a particular event (e). The type of rc is RequestClass, which should be the super class of all user defined request classes.

The `resolve` function is used to resolve the existing quantity annotation requests-made by request function or previous ungranted requests depending on the actual implementation of the quantity.

The `postcond` function is to clean up the states of the quantity and the quantity requests. The stable function returns whether or not the quantity resolution stabilizes. Section 4.3 talks about quantities in more detail.

The body of a quantity consists of fields, constructors, and methods. Among them, fields and constructors are subject to the same rules described for media in Section 2.3.

In Metropolis Metamodel, there is a built-in quantity, `GlobalTime`, that represents the global time of the system as a double precision floating point number.

2.5 Statemedia

A statemedium is a special type of medium used to communicate between processes and quantity managers. It passes the state of a process to the quantity manager, and

propagates the scheduling results back to the process. A statemedium is declared with a keyword `statemedium`, and the syntax below:

```
modifier statemedium StatemediumName extends SuperStatemediumName
                                          implements InterfaceName {
  // the body of this statemedium declaration
}
```

- **modifier** must be either unspecified, or one of the following keywords: `public`, `protected`, `private`, `abstract`, `static`, `final`. They are used in the same way as in process declarations.
- **StatemediumName** is the identifier of this statemedium.
- **SuperStatemediumName** is the identifier of a statemedium from which this statemedium is derived. `StatemediumName` and `SuperStatemediumName` must have different names.
- **InterfaceName** is the list of the interfaces implemented by the state medium.

A Statemedium by default implements two interfaces. One is `StateMediumSched`, the other is `StateMediumProc`. The interface StateMediumSched defines a set of functions for scheduling purpose as follows. An explanation of some of these functions, is present in the next chapter. StateMediumProc is an empty interface.

```
public interface StateMediumSched extends Port {
    eval process getProcess();
    eval SchedProgramCounter getProgramCounter();
    eval int getNumEnabledEvents();
    eval event getEnabledEvent(int i);
    eval boolean isEventEnabled(event e);
    eval event getMustDo();
    eval ArrayList getCanDo();
    eval int getSchedState();
    update boolean setSchedState(int newState);
    update boolean setMustDo(event e);
    update boolean setMustNotDo(event e);
}
```

The body of a statemedium consists of fields, constructors, and methods, as with the case of medium. Among them, fields and constructors are subject to the same rules described for medium in Section 2.3.

2.6 Netlists

Netlists are used to contain a set of objects and their connections. They can be used to build design hierarchies, to classify objects into different types of groups and to refine other objects. All metamodel designs have a top-level netlist.

A netlist is declared with a keyword `netlist`, and the syntax below:

```
modifier netlist NetlistName extends SuperNetlistName
                              implements InterfaceName {
  // the body of this netlist declaration
}
```

- **modifier** must be either unspecified, or one of the following keywords: `public`, `protected`, `private`, `abstract`, `static`, `final`. They are used in the same way as in process declarations.
- **NetlistName** is the identifier of this netlist.
- **SuperNetlistName** is the identifier of a netlist from which this statemedium is derived. `NetlistName` and `SuperNetlistName` must be different.
- **InterfaceName** is a list of the identifiers of interfaces, separated by comma. For each identifier in the list, it is required that this netlist declares all the members of the interface in its body. We say that these interfaces are implemented in this netlist.

2.6.1 Building Design Hierarchies

This is the most basic usage of netlists. They serve as containers of sets of objects. A typical flow of constructing one netlist is that during the execution of a constructor of the netlist, objects (including processes, media, statemedia, quantities, netlists, etc.) are instantiated or passed into the constructor as arguments and added to the netlist, then connections are made among these objects. If there exist refinements, objects and connections will be adjusted due to the refinement commands. When the constructor of the netlist finishes, the entire system structure is captured by this netlist. Another fact is that the netlist is fully decided at compile time, and metamodel does not support any dynamic changes to the netlist structure so far.

When instantiate an object, user must provide a so called instance name to the object. It has to be a unique name in order not to cause further problems in elaboration and simulation. Then, the object can be added into the netlist by calling

`addcomponent`(*NodeObject*, *NetlistObject* [, *ComponentName*])

This command will add *NodeObject* into *NetlistObject*. In metamodel, objects contained

in a netlist are called components of the netlist. Each component has a component name in a particular netlist, which is provided as an optional argument *ComponentName* when adding that component into the netlist. In case there is no *ComponentName* argument, a unique component name will be assigned to the component automatically. Note that it is possible that one object belongs to multiple netlists, therefore it has one instance name but multiple component names. The following constructs can be used to retrieve information about objects.

- `getinstname`(*NodeObject*) returns the instance name of the *NodeObject*.
- `getcompname`(*NodeObject*, *NetlistObject*) returns the component name of *NodeObject* in *NetlistObject*.
- `getcomponent`(*NetlistObject*, *ComponentName*) returns the object reference which is in *NetlistObject* and has its component name as *ComponentName*.
- `getprocess`(*Event*) return an object reference to a process which *Event* belongs to.
- `getthread`() returns an object reference to a process that is calling this construct.

Having all components in the netlist, we need to connect them together. A connection has three essential parts: a source object, a port and a destination object. A connection source could be a process, a medium, a statemedium or a quantity. The port to be connected must reside in the source object. It could be either the name of the port or a reference to that port, which is usually returned by other constructs like `getnthport` etc. The type of the port (a port interface) must be implemented by the destination object, which could be a medium, a statemedium or a quantity. In setting up the connect, we need to use the keyword

`connect`(*SourceObject*, *Port*, *DestObject*)

which connects *SourceObject* to *DestObject* through *Port*. In addition to the basic connection construct, there are a couple of other connection related constructs to help manipulate the netlist.

- `getnthport`(*NodeObject*, *InterfaceName*, *index*) returns the index'th port of type *InterfaceName* defined in *NodeObject*.
- `getportnum`(*NodeObject*, *InterfaceName*) returns the number of ports of type *InterfaceName* defined in *NodeObject*.
- `getconnectionnum`(*NodeObject*, *InterfaceName*) returns the number of objects connected to *NodeObject* through a port of type *InterfaceName*.

- `getconnectiondest`(*SrcObject*, *Port*) returns the destination object of a connection which is made from *SrcObject* and through *Port*. Here, *Port* could be either the name of the port or a reference to the port.
- `getnthconnectionsrc`(*NodeObject*, *InterfaceName*, *index*) returns the index'th object connected to the *NodeObject* through a port of type *InterfaceName*.
- `getnthconnectionport`(*NodeObject*, *InterfaceName*, *index*) returns the index'th port connected to the *NodeObject* through a *Port* of type interfaceName.

The type of a port may be a special interface called Scope, which is used in the await statements. Such a port is always connected to a netlist object, using the following statement

`setscope`(*NodeObject*, *Port*, *NetlistObject*)

Here, *NetlistObject* is an object of netlist, *NodeObject* is an object, and *Port* is a port of the *NodeObject* with the type Scope. This statement sets the value of *Port* to *NetlistObject*. Further, it makes *NodeObject* a component of *NetlistObject*, if it has not been so already. Similarly,

`getscope`(*NodeObject*, *Port*)

returns the netlist to which the *Port* of *NodeObject* connects to.[4]

2.6.2 Classify Objects

In metamodel, there are concepts of scheduled netlists and scheduling netlists (see chapter 4). Fundamentally, there are no distinctions between these two sorts of netlists. The way to tell the difference is to check the interfaces they are implementing. A scheduling netlist must implement an interface called *SchedulingNetlistIntfc*.

```
public interface SchedulingNetlistIntfc extends Port {
    eval boolean ifTop();
    update void top();
    update void postcond();
    update void resolve();
}
```

Among these functions, *ifTop*() returns true if the netlist is the top most one or false if the netlist is a component of another netlist. The other three functions are related to doing scheduling. Their exact meaning will be described in chapter 4.

[4] `setscope` and `getscope` are not fully supported at this time.

Other than implementing SchedulingNetlistIntfc explicitly, metamodel provides a short hand syntax to define a scheduling netlist, i.e. define a netlist and let it extend the SchedulingNetlist. SchedulingNetlist as shown below is nothing but a normal netlist implementing SchedulingNetlistIntfc. However, user should provide the actual implementation of all the functions defined in SchedulingNetlistIntfc. There is no default implementation of them in SchedulingNetlist.

```
public netlist SchedulingNetlist extends Netlist
                                  implements SchedulingNetlistIntfc {
    private boolean _top;
    public SchedulingNetlist(String name, boolean top) {
        _top = top;
    }
    public eval boolean ifTop() { return _top; };
    public update void top() {};
    public update void postcond() {};
    public update void resolve() {};
}
```

2.6.3 Refining Objects

Refinement is one of the key concepts in metamodel. It is used to migrate from one level of abstraction to another or change from one implementation to another. The syntax of refinement is

`refine`(*NodeObject*, *NetlistObject*);

During refinement, *NodeObject* is refined to a set of objects, which are encapsulated by *NetlistObject*. Syntactically, the creation of a netlist used in a refinement is the same as in creating a basic netlist in building design hierarchies. The only difference is that in refinement `refine` command will automatically add the *NetlistObject* into the network, while in building hierarchies `addcomponent` has to be called explicitly to add *NetlistObject* into the network. More details about refinement will be discussed in section 2.7.

2.7 Refinement

During refinement, an object is usually decomposed into a netlist of objects. For example, in Figure 2.1 (a), a single medium is decomposed into multiple processes and media after the refinement shown in Figure 2.1 (b). The meta-model uses the netlist objects to define refinements. The methods of a netlist must be implemented in order to carry out the following tasks.

The inputs to the methods are in two fold. First, an object to be refined must be provided. We denote it by x in the sequel. The type of this object is either process or medium. Second, a set of objects that have been already created elsewhere and will be used to constitute the netlist may be provided. In some cases, a part of the objects used to define the refinement are already created for other refinements. In Figure 2.1 (b), the medium object lc might model a shared resource used in different kinds of communication, and it might have been employed in this refinement as well as others. Such objects are provided as input to the methods of the netlist. In general, a single netlist may define more than one refinement scenario, depending on which objects are provided as input. For this reason, the netlist may implement more than one method with the same name, e.g. the constructor, which take different sets of objects as input in order to implement corresponding refinement scenarios accordingly. The types of these objects must be process, medium, scheduler, or netlists.

Given these inputs, the set of tasks to be carried out is listed as follows.

- Register this netlist as a refinement of the object x. This is done by using the refine statement:

  `refine`(*NodeObject*, *NetlistObject*);

  This statement sets the internal data structures of *NetlistObject* and *NodeObject* respectively.

- Create instances of process, medium, scheduler, and netlist that are necessary to constitute this netlist. Exactly what objects are needed may depend on the set of objects provided as input.

- Register the objects that constitute this netlist as its components, by using **add-component** statement defined in the previous section.

- If a component has a port of type Scope, it may be set to this netlist using **setscope** statement.

- Define the internal connections, i.e. connections among the components. This is done by using the connect statements.

- For each object that originally connects to x, e.g. a process object whose port is connected to x, define a component to which this object should be connected after the refinement. This is done by using the following statement.

  `refineconnect`(*NetlistObject*, *SrcObject*, *Port*, *ComponentObject*);

  NetlistObject is the netlist defining the refinement of x. *SrcObject* connects to x through the port *Port*. *ComponentObject* is a component of *NetlistObject* to which this port should be connected after the refinement. *ComponentObject* must implement the interface used as the type of *Port*. Unlike the connect statement,

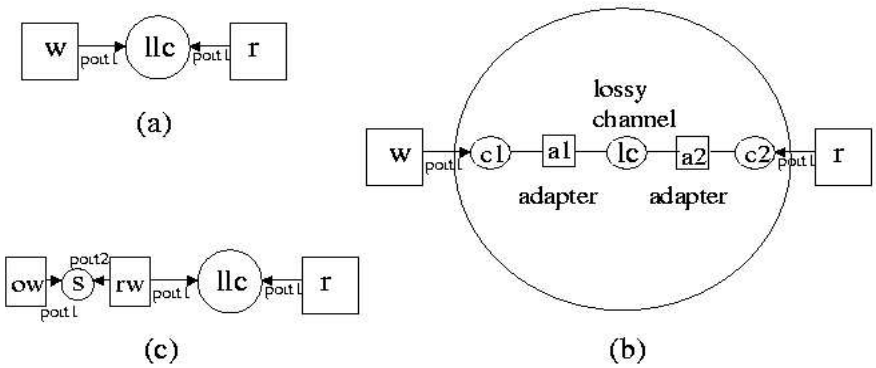

Figure 2.1: A Refinement Example

this connection is only logical. Namely, the value assumed by *Port* remains unchanged, designating the object being refined. This statement stores in an internal data structure of *NetlistObject* the information that *SrcObject* connects to *ComponentObject* through *Port*.

- For each object to which x connects, define a component and a port from which the connection should be made to the object after the refinement. This is done by using the following statement.

  `redirectconnect`(*NetlistObject*, *OrgObject*, *OrgPort*, *ComponentObject*, *NewPort*);

  NetlistObject is the netlist defining the refinement. *OrgObject* is the object being refined and *OrgPort* is a port of the object. *ComponentObject* is a component of the *NetlistObject* and *NewPort* is a port of *ComponentObject*. The type of *NewPort* must be identical with or a subclass of that of *OrgPort*.

  For example, in Figure 2.1 (c), suppose that the process object w in Figure 2.1 (a) is refined in to the three objects ow, s, and rw. If the original connection from the port1 of w to llc is represented by the connection from the port1 of rw to llc after the refinement, then this can be specified as

  `redirectconnect`(this, w, port1, rw, port1);

  where the first argument represents the netlist object that defines the refinement of w.

  This statement sets in an internal data structure of *NetlistObject* the information that the connection made from *OrgObject* through *OrgPort* is redirected so that it is made from *ComponentObject* through *NewPort*. It then sets the value of *NewPort* to the one assumed by *OrgPort*. The value assumed by *OrgPort* remains unchanged.

- Sometimes, it needs to know whether a connection has been refined or redirected. This information can be retrieved by calling the construct

  `isconnectionrefined`(*SrcObject*, *Port*, *DestObject*);

  Here, *SrcObject* and *DestObject* are the source object and the destination object of the connection. *Port* is the port through which the connection is made. These three component uniquely specify a connection. As its name suggests, this construct returns true if the connection is refined or redirected, otherwise false.

As an example, suppose that the loss-less channel llc in Figure 2.1 (a) is refined into a lossy channel lc, two adaptors a1 and a2, and two media c1 and c2, as shown in Figure 2.1 (b). Let us assume that llc implements two interfaces, Write and Read, and the process types of objects w and r are called W and R respectively. Similarly, the types of the adaptors, c1, c2, and lc are called Adaptor, C1, C2, LossyChannel, respectively, where Adaptor is a process and the rest are media. The medium objects c1

and c2 implement Write and Read interfaces used by the ports of W and R respectively. This refinement can be defined as follows.

Example 2.7.1

```
// declare a netlist type named Netlist1
netlist Netlist1 {
    Netlist1(LossLessChannel llc) {
        // Scenario 1: no object is provided.
        refine(llc, this);
        // create a lossy channel
        LossyChannel lc = new LossyChannel();
        create(llc, lc);
    }
    Netlist1(LossLessChannel llc, LossyChannel lc) {
        // Scenario 2: LossyChannel is provided.
        refine(llc, this);
        create(llc, lc);
    }
    void create(LossLessChannel llc, LossyChannel lc) {
        int i;
        C1[] c1;
        C2[] c2;
        Adaptor[] a1;
        Adaptor[] a2;
        addcomponent(lc, this);
        // instantiate local objects and connect them
        c1 = new C1[getconnectionnum(llc, Write)];
        a1 = new Adaptor[getconnectionnum(llc, Write)];
        for(i=0; i < getconnectionnum(llc, Write); i++) {
            c1[i] = new C1(); addcomponent(c1[i], this);
            a1[i] = new Adaptor(); addcomponent(a1[i], this);
            connect(a1[i], port1, c1[i]);
            connect(a1[i], port2, lc);
        }
        c2 = new C2[getconnectionnum(llc, Read)];
        a2 = new Adaptor[getconnectionnum(llc, Read)];
        for(i=0; i < getconnectionnum(llc, Read); i++) {
            c2[i] = new C2(); addcomponent(c2[i], this);
            a2[i] = new Adaptor(); addcomponent(a2[i], this);
            connect(a2[i], port1, c2[i]);
            connect(a2[i], port2, lc);
        }
        // establish external connections
        for(i=0; i < getconnectionnum(llc, Write); i++)
            refineconnect(this, getnthconnectionsrc(llc, Write, i),
                getnthconnectionport(llc, Write, i), c1[i]);
        for(i=0; i < getconnectionnum(llc, Read); i++)
            refineconnect(this, getnthconnectionsrc(llc, Read, i),
                getnthconnectionport(llc, Read, i), c2[i]);
    }
}
```

In this example, Netlist1 supports two kinds of refinement scenarios. One is the case where the object modeling the lossy channel has been already created elsewhere and is provided as input. The other case is that no object is provided and all the internal objects are created in this netlist. The netlist has two constructors for this reason. The method create() is the main body of defining the refinement. It is written so that this netlist can be used for an arbitrary number of writers and readers connecting to the original object llc being refined. For each writer w, it creates a pair of objects c1[i] and a1[i] and connect them. a1[i] also connects to the lossy channel lc. Then the connection from w to lc is refined so that it is connected to c1[i], using the refineconnect statement. The readers are similarly handled. The created objects are registered as components of the netlist.

Chapter 3

Specification of Network Executions

This section covers the meta-model constructs used to specify a set of executions of a single network. The execution of a model, is defined by the trace of events that each process executes. Each process executes under its own thread of control, which is formally sketched in the next section. In particular we also provide an overview of the statements listed here distinguish metropolis from the Java multithreaded programming model of concurrent sequential threads. The statements covered are: await, bounded-loop, nondeterminism, and blackbox. The complete semantics is given in [1].

3.1 Semantics

The semantic domain we use to interpret executions of meta-model netlists 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. A precise definition of an action is given in the next section.

The execution of netlists evolves through a sequence of *state transitions*, where each transition consists of a current state, a set of observable actions (at most one for each process) and the next state, which is also the current state of the subsequent transition. A state of the program consists of two parts. The first is the state of the *memory* which consists of assignments to *state variables*. The second part of the state corresponds intuitively to the program counters and stack of all the processes in the network.

3.2 await

This construct is used to specify an execution under a guarantee that a certain condition holds. The `await` statement can only be used in methods of `process` and `medium` objects.

The **await** statement is used to describe a situation that a process object waits for a given condition to become true, and once the condition holds, it continues its execution. This semantics is different from that of the usual **if** statement, because the latter does not wait until the condition becomes true and there is no guarantee in concurrent programming that the condition still holds when the execution of its statements begins. Further, one can specify in the await statement a set of activities that cannot be carried out by other process objects during the execution of the statements.

The syntax of the await statement is the following:

Syntax 3.2.1

await {
< (*guard*; << *PortName.TestList* >> ; << *PortName.SetList* >>) [*statements*;] >
<< (**default**; << *PortName.TestList* >> ; << *PortName.SetList* >>) [*statements*;] >>
}

guard is an expression to be evaluated as **true** or **false**. *PortName.TestList* is either the keyword `all`, or else a list of elements separated by commas, where each element is in the form of *PortName.IfName*. Here, *PortName* is either a port of the object in which the await resides, or else a keyword `this`. *IfName* is either a port interface, or else a keyword `all`. As a special case, *PortName.TestList* may be empty. The syntax of *PortName.SetList* is same as that of *PortName.TestList*. [*statements*;] refers to a block of statements. These blocks are also *called critical sections*. There may exist more than one list of (*guard*; *PortName.TestList*; *PortName.SetList*) {*statements*;}. The last line with **default** is optional.

The semantics of `await` is defined only if *guard*'s are pure state predicates, i.e. executing a guard should not change the value of any variable, nor generate any observable events . If this is not the case, the semantics is undefined. With each *PortName.TestList* and *PortName.SetList* we associate a set of actions of other process denoted by ⟦*PortName.TestList*⟧ and ⟦*PortName.SetList*⟧ respectively. If *PortName.TestList* is empty, then so is ⟦*PortName.TestList*⟧. If *PortName.TestList* is the keyword `all`, then ⟦*PortName.TestList*⟧ contains all the actions of all other processes. Finally, if *PortName.TestList* is a list, ⟦*PortName.TestList*⟧ is the union of sets ⟦*PortName.IfName*⟧, one for each element of the list. To determine ⟦*PortName.IfName*⟧, we first determine the port ⟦*PortName*⟧ associated with *PortName*, as follows;

- if *PortName* is the keyword `this` and the object it appears in is a medium, then

⟦*PortName*⟧ is that medium, if *PortName* is the keyword `this` and the object it appears in is a process, then ⟦*PortName.IfName*⟧ is empty,

- if *PortName* is a port name, then ⟦*PortName*⟧ the medium connected to that port.

Now, if *IfName* is the keyword `all`, then ⟦*PortName.IfName*⟧ contains calls to any interface function of medium ⟦*PortName*⟧, on any interface, and *IfName* is an interface name, then ⟦*PortName.IfName*⟧ is restricted to the functions in the interface *IfName*. The set ⟦*PortName.SetList*⟧ is determined according to the same rules.

We say that a critical section is *enabled* if its *guard* is **true**, and no actions in ⟦*PortName.TestList*⟧ are currently being executed. If an `await` statement has no enabled critical sections, then it blocks until at least one becomes enabled. If some critical sections are enabled, then one is chosen (non-deterministically) to begin its execution. During this execution no actions in ⟦*PortName.SetList*⟧ can start.

3.3 boundedloop

Another important construct in the meta-model is `boundedloop`. It has following syntax:

Syntax 3.3.1

boundedloop(*iterVar*, *iterCount*) {
  [statement;]
}

Its semantics is equivalent to:

for(*iterVar*=0; *iterVar* < *iterCount*; *iterVar*++) {
  [statement;]
}

However, while the latter may iterate infinitely, **boundedloop** can be used (under the user's responsibility) only with a guarantee that the iteration stops within the specified steps. Such an explicit guarantee eases the analysis task for a compiler. Note that *iterVar* cannot be modified by any of the statements inside the construct.

3.4 nondeterminism

Processes may have non-deterministic characteristics. Such non-determinism comes from (1) incomplete specification (i.e. a system described in a relatively high level) or (2) fragmentary information, i.e. a system too complicated to describe completely. Modeling the environment can be such an example. In the meta-model, an operator nondeterminism is introduced to describe such non-deterministic behavior. Its syntax is:

Syntax 3.4.1

data_type **nondeterminism**(*data_type*);

where nondeterminism(data_type) can return any value of data_type; and data_type can be of any supported primitive type in the meta-model. In the following example,

Example 3.4.1

tk = **nondeterminism**(int) % 2;

nondeterminism(int) can return any value of int type.

3.5 label

The meta-model supports labels. As in C, a label is an identifier followed by a colon, and may appear anywhere in a method. Unlike C, the scope of a label is the whole class where it appears, in order to make constraint specification easier. In addition, it is also possible to attach a label to a block of statements. This is useful mostly for naming the blocks of statements such as **if**, **else**, **for** and so on. It is done by using the **block** construct:

Syntax 3.5.1

block(*label*) {
    [*type VarName*;]
    [statements;]
}

block(*label*) can precede any block of statements defined in Java, i.e. a contiguous region of code that consists of statements, where the block is surrounded by { and }. It only annotates the block so that constraints can refer to it. Of course, prefixing the block with a label is semantically equivalent:

Example 3.5.1

l1: while (i < 3) **block**(*l2*) { ... }
l1: while (i < 3) *l2*: { ... }

l3: if (j < 3) **block**(*l4*) { ... } else **block**(*l5*) { ... }
l3: if (j < 3) *l4*: { ... } else *l5*: { ... }

***label*{@ ... @}** is a general way to specify labeled statement(s). In addition, it is more powerful than labels and blocks. It can label an expression inside a statement.

For example,

Example 3.5.2

*l1***{@** while (i < 3) *l2***{@** ... **@} @}**
*l1***{@** while (*l2* **{@**i < 3**@}**) { ... }**@}**
*l1***{@** i = *l2***{@**i + 1 **@} @}**

3.6 blackbox

This construct is used to specify a section of code that is not analyzed by the meta-model compiler. This is useful to insert texts such that their syntax or semantics is not supported by the meta-model but the texts are meaningful to some Metropolis tools that take the meta-model specification as input. For example, one may want to write a text that can be printed on the display during a simulation of the meta-model specification conducted by a SystemC-based Metropolis simulator. If the simulator has a feature that it can execute SystemC code directly written in the meta-model, as far as it is specified using the blackbox construct with "SystemCSim" identifier, then one can write in the meta-model:

blackbox(SystemCSim)%%
  count¡¡"The write command executed at port0."¡¡endl;
%%

Such a Java statement is not supported in the meta-model, but the simulator understands and executes it during the simulation. The syntax of the construct is the following.

Syntax 3.6.1

blackbox(*identifier*)%% *text* %%

Here *identifier* is an arbitrary sequence of characters, and *text* is any text not including %%. This construct can be placed anywhere in a file that corresponds to type declaration, member declaration (i.e. methods and fields), or statements. As with the constraint clause, the control flow points of the meta-model do not enter inside this construct. Therefore, its semantics is undefined at the meta-model level; it is simply provided as a convenient mechanism to provide tools with specific capabilities, as defined by *identifier*, with data in their own format to be interpreted in specific positions in the meta-model. Intuitively, it is analogous to allowing uninterpreted macros inside the meta-model code, each annotated with its expansion in a given context such as simulation or synthesis. Right now, *identifier* can take "SystemCSim" and "elaborator" to

include native codes for the SystemC-based simulator and the elaborator respectively.

Chapter 4

Annotating and Restricting Network Executions

This chapter describes meta-model mechanisms to annotate and restrict (i.e. declare invalid) executions of a single network. Both annotation and restriction is necessary to model design refinement. Annotations are typically used to represent cost of performing certain computations on a given architecture, e.g. delay or energy information. In the metamodel, annotations are the responsibility of objects called *quantity managers*. Models of architecture can include the cost information through annotation requests. For example, to specify that certain operation o requires d time units to execute, we would generate a request that the difference in time stamps between the beginning and the end of o is exactly d. Quantity managers collect requests from all the processes and try to satisfy them. If there is an event for which the annotation request cannot be granted, that event needs to be prevented from occurring. In other words, the quantity managers not only annotate events, but also determine which events should occur, i.e. they schedule the execution of the model. That is why the collection of quantity managers (and some other related objects) is called the *scheduling network*, and the system model, which we so far have referred to just as network, is also sometimes referred to as the *scheduled* network.

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, as described in the following sections, determine 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 metamodel 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 metamodel restriction if and only if:

- it can be generated by the execution of the scheduled network restricted by the scheduling network,
- it satisfies all the constraints specified by logic formulas.

In the rest of this section we explained both of these mechanisms in more detail.

4.1 Actions, Events and Event References

Actions are executions of the pieces of code in the scheduled network. We have already mentioned that function calls to media through ports are observable actions, but the set of all actions is much richer, and it includes each statement in the code of the scheduled network.

With each action a we associate two *events*, a^+ indicating the start of an execution of a, and a^- indicating the end. For each process P we define the set of events Σ_P that contains a^+ and a^- for each action a of P, and a special symbol $P : nop$, indicating that no events are occurring in P. When no ambiguity can arise, we will abbreviate $P : nop$ to nop. Cross-product of all the sets of events in the system is called the set of *event vectors*. Occasionally, we will treat an event vector $\sigma = (\sigma_1, \ldots, \sigma_n)$ as the set $\{\sigma_1, \ldots, \sigma_n\}$, and write expressions like $\sigma_2 \in \sigma$. Two representations of σ are equivalent, because Σ_P's are disjoint, and we will make no notational distinction between them.

In several places in the meta-model, there is a need to refer to an event. This can be done with an expression of type `event`. To specify such an expression, we use expressions of types `process`, `medium`, and `action`, which we define next.

Expressions of type `process` are used to specify processes. Given some expression e of type `process`, we use $[\![e]\!]$ to denote its semantic value, i.e. the process that it specifies. An expression e of type `process` can be one of the following:

- within a process declaration, it can be the keyword `this`, in which case $[\![e]\!]$ is that process,
- within a netlist declaration, it can be a process name defined in that scope, in which case $[\![e]\!]$ is the process identified by that name,
- within a medium declaration, it can be a `getnthconnectionsrc` expression, in which case $[\![e]\!]$ is the process that is the value of that expression,
- within a netlist declaration, it can be a `getnthconnectionsrc` or `getcomponent` expression, in which case $[\![e]\!]$ is the process that is the value of that expression,
- within a method definition in a medium or a process it can be `getthread()` in which case $[\![e]\!]$ is the process currently executing that code.

The semantic value of an expression e of type `medium` is a medium denoted by $[\![e]\!]$. Such an expression can be one of the following:

- within a process or a medium declaration, it can be a port name, in which case $[\![e]\!]$ is the medium connected to that port,
- within a netlist declaration, it can be a medium name defined in that scope, in which case $[\![e]\!]$ is the medium identified by that name,
- within a medium declaration, it can be a `getnthconnectionsrc` or `getconnectiondest` expression, in which case $[\![e]\!]$ is the process that is the value of that expression,
- within a netlist declaration, it can be a `getnthconnectionsrc`, `getconnectiondest` or `getcomponent` expression, in which case $[\![e]\!]$ is the process that is the value of that expression.

Since connections can be made from a `process` to a `medium` or from a `medium` to another `medium`, it is an error to use `getnthconnectionsrc` and `getconnectiondest` as an expression of type `process` (respectively `medium`) identifier, if the object returned by these two constructs is a `medium` (`process`). The same applies to `getcomponent`, which could return a `process` or a `medium` in a netlist. This kind of error can be discovered after netlist elaboration.

Expressions of type `action` are specified as follows:

Syntax 4.1.1

`all`
⟨*object*⟩.⟨*name*⟩

where `all` is a keyword, ⟨*object*⟩ is an expression of type `process` or `medium`, and ⟨*name*⟩ is either a label, or a function name. In either case, ⟨*name*⟩ must be in scope at the declaration of the process or medium specified by ⟨*object*⟩.

The semantic value of an expression e of type `action` is a set of actions denoted by $[\![e]\!]$. If e is the keyword `all`, then $[\![e]\!]$ contains all the actions in the system. If e is of the form ⟨*object*⟩.⟨*action*⟩, then its semantic value is determined as follows:

- if ⟨*action*⟩ is a function name, then $[\![$⟨*object*⟩.⟨*action*⟩$]\!]$ contains all the calls to the function with that name that is a member of the object (a process or a medium) specified by ⟨*object*⟩,
- if ⟨*action*⟩ is a label name, then $[\![$⟨*object*⟩.⟨*action*⟩$]\!]$ contains all the statements or blocks of statements labeled with ⟨*action*⟩ that appear in a member function of the object specified by ⟨*object*⟩.

Event reference is an expression of type `event`, which can be specified as follows:

Syntax 4.1.2

`beg`(⟨*process*⟩, ⟨*action*⟩)
`end`(⟨*process*⟩, ⟨*action*⟩)
`none(`⟨*process*⟩`)`
`other(`⟨*process*⟩`)`

where ⟨*process*⟩ is an expression of type `process`, or the keyword `all`, and ⟨*action*⟩ is an expression of type `action`.

If ⟨*process*⟩ is the keyword `all`, then the semantic value of an event reference `beg`(⟨*process*⟩, ⟨*action*⟩) is the set of beginnings of all actions in ⟦⟨*action*⟩⟧, and the semantic value of `end`(⟨*process*⟩, ⟨*action*⟩) is the set of endings of all actions in ⟦⟨*action*⟩⟧. If ⟨*process*⟩ is an expression of type `process`, then these sets of events are restricted to events executed by the process ⟦⟨*process*⟩⟧.

If ⟨*process*⟩ is the keyword `all`, then the semantic value of `none(`⟨*process*⟩`)` is the set containing events $P : nop$ for all processes P. If ⟨*process*⟩ specifies a single process P, then the semantic value of `none(`⟨*process*⟩`)` is the singleton $\{P : nop\}$. Finally, the semantic value of `other(`⟨*process*⟩`)` contains all events of process(es) specified by ⟨*process*⟩, that cannot be specified by any of the other three constructs (i.e. all events associated with actions that are *not* function calls or labeled statements or blocks).

We say that event references `beg`(⟨*process*⟩, ⟨*action*⟩), and `end`(⟨*process*⟩, ⟨*action*⟩) are *scoped* if neither ⟨*process*⟩ nor ⟨*action*⟩ are the keyword `all`. With each scoped event reference we associate the set *Scope* which contains names of all the variables that are in the scope when the corresponding events occur.

More precisely, if ⟨*action*⟩ is a statement label, then *Scope*(`beg`(⟨*process*⟩, ⟨*object*⟩.⟨*action*⟩)) and *Scope*(`end`(⟨*process*⟩, ⟨*object*⟩.⟨*action*⟩)) are the same, and they contain the names of all variables that are in the scope at that point of code.

If ⟨*action*⟩ is a block label, then *Scope*(`beg`⟨*process*⟩, ⟨*object*⟩.⟨*action*⟩)) contains names of all the variables that are in scope just after the left brace starting the block (excluding thus any variable defined inside the block), and *Scope*(`end`(⟨*process*⟩, ⟨*object*⟩.⟨*action*⟩)) contains names of all the variables that are in scope just before the right brace ending the block.

If ⟨*action*⟩ is a function name, then *Scope*(`beg`(⟨*process*⟩, ⟨*object*⟩.⟨*action*⟩)) contains the names of all variables that are in scope just after the left brace starting the function body (including thus all the formal arguments from its declaration, but excluding any variable declared inside the body), *Scope*(`end`(⟨*process*⟩, ⟨*object*⟩.⟨*action*⟩)) contains names of all the variables that are in scope just before the right brace ending the function body, and the keyword `retval`, which is used as a name for the state variable containing the return value of the function.

In general, there may be more than one state variable with the same name. In particular, return value of all the functions are named `retval`. However, since a given scoped reference (say r) determines a unique code location executed by a unique process, there is a straightforward mapping from names in *Scope*(r) to state variables.

4.2 Annotations with quantities

To specify performance constraints in the meta-model, behaviors must first be annotated with a quantity, such as time, power, or Quality of Service. In the meta-model, such annotations are defined with a concept called *quantity*. 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 the rest of this section we describe the structure of the scheduling network, the way it executes, and the objects it contains: quantity managers, statemedia (communication channels between processes and quantity managers), and requests (messages between processes and quantity managers).

4.3 Scheduling network

By default, the scheduling network consists of:

- a statemedium for each process in the scheduled network,
- a manager for each quantity in the scope, where a quantity is in the scope if:
  - it is defined in the netlist
  - it is passed as argument in the netlist constructor
- any additional user-defined processes and media (You may ignore this for now. It is explained in Section 4.3.5.)

While these objects are defined by default, it is the user's responsibility to interconnect them.

In contrast to ordinary ("scheduled") networks, scheduling networks have some methods associated with them. Who calls these methods and when, is described in Section 4.3.4, but for now we only describe the functions themselves.

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.

4.3.1 Statemedia

Each process in the scheduled network has a representative in the scheduling network. This representative is an object of type `statemedium`. The primary purpose of statemedia is to exchange information between processes in the scheduled netlist and quantity managers in the scheduling netlist. Through statemedia, processes can request annotations for their enabled events. Quantity managers can examine which events are enabled and which requests are made, and they disable events by calling `setMustDo` or `setMustNotDo`.

The function `eval process getProcess();` returns the process associated with the statemedium.

The functions:

```
eval int getNumEnabledEvents();
eval event getEnabledEvent(int i);
eval event getMustDo();
eval ArrayList getCanDo();
```

let the quantity managers examine currently enabled events of the process associated with the statemedium. Function `getCanDo` returns an array of enabled events. The user can examine the array element by element, or equivalently, she can use functions `getNumEnabledEvents` and `getEnabledEvent` to iterate over enabled events. If there is a unique enabled event, function `getMustDo` returns it. Otherwise, it returns NULL.

Once an enabled event is accessed it can be stored in a variable, but its status may change. To check if an event is still enabled, the user may call `eval boolean isEventEnabled(event e);`

Finally, the quantity managers may change the status of event by calling `update boolean setMustDo(event e);` which disables all events but `e`, or by calling `update boolean setMustNotDo(event e);` which disables `e`.

4.3.2 Quantity managers

Quantity managers implement a function called `A`, which is used to refer to annotations. The type of `A` is the type of annotation. This function takes two arguments: `e` of type `event`, and `i` of type `int`. Intuitively, `Q.A(e,i)` denotes the value of `Q` annotation for the `i`-th occurrence of event `e`. In an executable piece of code (e.g. making a request is executable code, but a constraint is not) it is legal to substitute `i` with the keyword

`LAST`. The value of `LAST` is basically the number of occurrences of event `e` up to that point in the execution.

The user may or may not define `A`. Typically, it is defined only after the design is mapped to a particular platform. The user must ensure that `A` is not used in constraints, or otherwise, if it is not defined.

The process may request a particular annotation for an enabled event. The statemedium is the first to receive the request (exactly how will be described in the following sections). The statemedium should forward the request to the appropriate quantity manager by calling:

```
eval void request(event e, RequestClass rc);
```

which the designer of the quantity manager should define.

`RequestClass` is a container used to package requests for various quantity managers. With each quantity manager, there should be one or more associated sub-classes of this class that correspond to various types of requests. Some examples might be:

- a request for a specific value,
- a request that specifies minimum and maximum acceptable values,
- a request for an annotation that is the same as the annotation of some other event,
- a request that does not specify a particular value, but specifies a priority level.

A quantity manager designer also needs to define functions `update void resolve();` and `update void postcond();`. Typically, in the `resolve`, a quantity manager looks at the pending requests, and decides if they can or cannot be granted. If the request cannot be granted, the managers disables the corresponding events. For example, if several events request a time-stamp, the time manager must set the current time to the lowest of all requests, and it must disable all the events requesting a higher time-stamp.

Repeatedly calling the `resolve` function of all the quantity managers will decrease the number of enabled events. Eventually, there will be a single enabled event for each process. The details of how we get to this point are described in Section 3.1. At this point, the vector of events that will occur has been set, and the quantity managers can assign annotations to these events. This is the primary use of the function `update void postcond();`, which the quantity manager designer should define. Other typical uses of this function is to clean-up data used in the resolution process.

Since, many managers need to cooperate in selecting an event vector that can be annotated consistently with all made requests, the function `resolve` is often called many times. If during a particular call, the quantity manager disables some event, or makes some annotation, then a subsequent call to function `eval boolean stable()` returns `false`, and otherwise it returns `true`. In other words, `stable` is useful to decide if the resolution process has converged, i.e. it has reached the point at which further calls to `resolve` will not change enabled events. Currently, the user must define `stable`, but in the future it may be provided by default.

As a part of a definition of a new quantity, the user may specify some *axioms*, i.e. properties that corresponding annotations must always satisfy. Axioms are specified with a `constraint{}` construct that may contain LOC and ELOC formulas.

The meta-model provides a pre-defined integer-valued quantity called *global execution index*, denoted with `GXI`. For a given sequence of event vectors $v_1, v_2, \ldots$, the value of `GXI` for all events in the vector v_i is i. Informally, and event has `GXI` annotation of i, if it occurs in the i-th "step" of system execution.

4.3.3 Annotation requests

Code executed by a process can have inserts with the following syntax:

```
{$
[ beg{ ⟨begin_code⟩ }]
[ end{ ⟨end_code⟩ }]
$}
```

By their position in the code, these inserts are always associated with an action. More precisely, ⟨*begin_code*⟩ is associated with the beginning of that action, and ⟨*end_code*⟩ is associated with its end. These two pieces of code are also called *request making code*, or *RM code* for short. The primary purpose of RM code is to generate an annotation request for the event it is associated with. To do so, it typically involves some computation, and then a call to the `request` function of some quantity manager.

For example in the following code fragment:

```
while(j>0){
  labela{@
    {$
        end{
          currentTime=pgt.A(beg(getthread(), this.labela), LAST);
          pgt.request(end(getthread(), this.labela),
                    new GlobalTimeRequestClass(currentTime+cycle));
        }
    $}
    j--;
  @}
  blackbox(SystemCSim) %%
    cout<<"In process "<<(pc->p->name())<<": cycle="<<cycle<<" j="<<j<<endl;
    cout<<"GlobalTime @ beg ="<<currentTime<<endl;
    cout<<"GlobalTime @ end ="<<currentTime+cycle<<endl;
  %%
}
```

the RM code is associated with the block with label `labela`. It makes a request for the difference in time stamps between beginning and the end of execution of the the block `labela` to be exactly `cycle` time units.

The RM code must be written in a way that no two pieces of RM code interfere, i.e. interleaving their execution in an arbitrary way should not change the requests that they make. It is the user's responsibility to satisfy this requirement. If it is not satisfied, the semantics of the scheduling network is not defined.

We give precise rules for execution of RM in the next section.

4.3.4 Scheduling network execution

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 RM code. A scheduling network consists of the following stages:

1. **request making:** In this phase RM codes of all enabled events are executed. This typically generates request for annotation of enabled events.

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.

4.3.5 Recursive scheduling network

Scheduling networks can be extended by media and process. Quantity managers and statemedia can be extended with additional ports and these ports can be connected to additional media. We require strict layering: media and processes that can be reached through ports of quantity managers and statemedia must be distinct from media and processes in the scheduled network. Therefore, these extra media and processes form a lower-layer scheduled network. This lower-layer scheduled network can have a scheduling network of its own, but again, we require that quantities in this lower-layer scheduling network are distinct from the quantities in the higher-layer scheduling network. This layering can extend to any level of depth. The execution rules for these networks remain essentially the same, except that for the negotiation (respectively annotation) phase

to terminate, its is not sufficient for topmost `top` (respectively `postcond`) function to terminate, but also the network of processes and media at that level must reach a steady state, i.e. a state in which all processes are blocked in `await` statements.

4.4 Constraints

Constraints in the meta-model can generally be divided into two classes: *coordination constraints*, and *performance constraints*. Coordination constraints are used to rule out certain sequences of event vectors that would otherwise be legal according to the execution semantics. Thus it is a quick and convenient way to restrict concurrency, possibly to model restrictions of the intended implementation platform. This effect can be obtained using schedulers, but we also provide an alternative way, using *linear temporal logic (LTL)*. LTL [5] is a well known and well studied logic used to reason about behaviors of concurrent systems. We do not change any of the standard LTL definitions. However, to integrate it into the meta-model, we have to define precisely the notion of *atomic formulas* which are usually dealt with quite abstractly in the classical literature.

Performance constraints deal with quantities like time and power, that are usually known only at later stages of the design, when many implementation decisions have been made. To describe such attributes, the meta-model provides a special class called `Quantity`. To specify performance constraints, the meta-model uses *logic of constraints (LOC)*. LOC formulas are used to specify constraints that a design must meet, but in addition to these constraints, the meta-model also enables expressing properties or quantities that hold always, and not just for a specific design (e.g. time cannot decrease). These properties are called *axioms* of a property. To express them, the meta-model uses a generalization of LOC called *extended logic of constraints (ELOC)*.

Formulas of various logics are specified in the meta-model as follows:

Syntax 4.4.1

`ltl` ⟨*name*⟩(⟨*args*⟩) ⟨*ltl-wff*⟩;
`loc` ⟨*name*⟩(⟨*args*⟩) ⟨*loc-wff*⟩;
`eloc` ⟨*name*⟩(⟨*args*⟩) ⟨*eloc-wff*⟩;

where ⟨*ltl-wff*⟩, ⟨*loc-wff*⟩ and ⟨*eloc-wff*⟩ are *well-formed formulas*, to be defined shortly. If the formula appears inside a `constraint{}` construct, then ⟨*name*⟩ and arguments must be omitted. Only these formulas define actual constraints. Named formulas defined outside a `constraint{}`, provide a convenient way for constraints to share sub-formulas, increasing thus readability, re-usability and compactness of the code (but not its expressive power). All variables appearing in well-formed formulas must be in scope of their declaration. The scope of variables appearing in the argument list is the well-formed formula following it. In addition to standard meta-model types, arguments can also be of type `event` and `action`, defined in the next section.

In the subsequent sections, we define syntax and semantics of well-formed formulas of the three logics. We start by defining the part that they all share.

4.4.1 Meta-model LTL syntax

Well-formed LTL formulas ⟨*ltl-wff*⟩ can be one of the following:

1. an event reference, as described in Section 4.1,
2. an expression of type `bool` involving member variables and functions of type `eval` of the object in which the formula appears,
3. an expression of the form ⟨*eventRef*⟩`=>`⟨*localExpr*⟩ where ⟨*eventRef*⟩ is a scoped event reference, and ⟨*localExpr*⟩, is an expression that may involve variables in *Scope*(⟨*eventRef*⟩) in addition to variables and functions mentioned in the previous item,
4. an expression of the form ⟨*name*⟩(⟨*args*⟩), where ⟨*name*⟩ is the name of a previously defined LTL formula (as usual, the number and type of arguments must match the declaration)
5. expressions of the form `F` f, `G` f, `X` f, f `U` g, (f), `!`f, f`&&`g, f`||`g, f `->` g, and f `<->` g, where f and g are well-formed LTL formulas

The first three items above are meta-model specific. They define formulas that correspond to atomic propositions in the classical definition. The fourth item provides a simple way to re-use parameterized formulas, and the fifth item defines the standard way of building up LTL formulas from atomic propositions.

4.4.2 Meta-model LTL semantics

Given a meta-model netlist, and its execution:

$$s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_i} s_i \xrightarrow{\sigma_i+1} \ldots \quad ,$$

we interpret LTL formulas over the sequence:

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

If α is such a sequence, then we say that:

1. α satisfies an event reference r if one of the events in $[\![r]\!]$ appears in σ_1,
2. α satisfies a meta-model expression e if s_0 is in $True(e)$, (where $True$ is defined as in Section 3.2),

3. α satisfies an expression of the form r`=>`e, where r is a scoped event reference and e is an expression, if either no events in $[\![r]\!]$ appears in σ_1, or if s_0 is in $True(e)$,

4. if f a name of some LTL formula, and $y_1, \ldots, y_n$ are variables appearing in the argument lists of the definition of f, then α satisfies $f(x_1, \ldots, x_n)$, if it satisfies the formula obtained from the definition of f by substituting y_i with x_i, for all $i = 1, \ldots, n$,

5. α satisfies other types of LTL formulas as in the standard definition [5].

4.4.3 Meta-model LOC and ELOC Syntax

In the meta-model, the syntactic elements of LOC and ELOC are defined as follows:

Annotations: For each quantity and variable name (including `retval`) in a given netlist, we create an LOC annotation. Syntactically, we use `F.A(`$\langle e \rangle$`,` $\langle t \rangle$`)` to represent $f(e[\tau])$, where `F` is quantity or variable name that corresponds to annotation f, $\langle e \rangle$ is a reference to e, and $\langle t \rangle$ is a meta-model integer expression that corresponds to τ.

Event names: The set of event names is exactly the set of scoped event references described in Section 4.1.

Algebra: LOC and ELOC are defined relative to a multi-sorted algebra given by the type system, including both built-in and user-defined types.

Terms: The type system also defines all the operators and relations, i.e. any expression allowed by the type system is a legal term.

Well-formed formula: An ELOC well-formed formula can be one of the following:

- a term of type `bool`
- an expression of the form $\langle name \rangle(\langle args \rangle)$, where $\langle name \rangle$ is the name of a previously defined ELOC formula (as usual, the number and type of arguments must match the declaration),
- an expression of the form `forall(`$\langle args \rangle$`)`f or `exists(`$\langle args \rangle$`)` f, where f is an ELOC well-formed formula, and $\langle args \rangle$ is a list of arguments as in function declaration,
- an expression of the form `!`f, f`||`g, f`&&`g, f`->`g, or f`<->`g, where f and g are ELOC well-formed formulas.

LOC well-formed formulas are ELOC well-formed formulas that do not contain any quantifiers, either directly, or indirectly, through calls to previously defined ELOC formulas.

Formulas: An ELOC formula is an ELOC well-formed formula in which every variable is in scope in one of the following ways:

- the formula is named, and the variable appears in the argument list associated with the name,
- the formula is in scope of a quantifier, and the variable appears in the argument list of the quantifier.

LOC formulas are un-named ELOC formulas of the form:

$$\texttt{forall(int } \langle var \rangle)\langle wff \rangle \ ,$$

where $\langle var \rangle$ is an arbitrary variable name (not necessarily i), and $\langle wff \rangle$ is an LOC well-formed formula.

Notice that there is no way to refer to the value of events, only their annotation. This is because, we do not define any value for events in the meta-model, and store all relevant information in the annotations.

For example, if in some network contains producer processes `P0, P1, P2, P3` and consumer processes `C0, C1`, all of type `IntX`, and `Rd` and `Wr` are labels inside `IntX` marking the reading and writing of some token, then the latency constraint on time between a token is produced (written) and consumed (read) can be specified as follows:

```
loc latency(IntX p, IntX r, int C)
  GTime.A(end(r, r.Rd), i) - GTime.A(beg(p, p.Wr), i) < C;

constraint {
  loc latency(P0, C0, 10);
  loc latency(P1, C0, 10);
  loc latency(P2, C1, 10);
  loc latency(P3, C1, 10);
}
```

4.4.4 Meta-model semantics of LOC and ELOC

Meta-model executions, as defined in Chapter 3, together with defined quantities, contain all the information needed to evaluate LOC and ELOC formulas. However, we still need to put this information in a form consistent with the definition of annotated behaviors.

To do this, we first define a transformation from sequences of event vectors (as defined by the execution semantics) to annotated behaviors. Since the meta-model does not use event values, we omit defining value in basic behaviors, and focus on defining annotations, which corresponds quantity and variable names. In general we will use X to denote the annotation associated with the quantity or variable name `X`. We introduce the slight difference in fonts to maintain the consistent use of fonts for

meta-model constructs on one side, and LOC concepts on the other. However, the distinction between annotations and quantity and variable names is better inferred from the context, rather than the difference in fonts.

The semantics of LOC require that values and annotations be defined for the n-th occurrence of any event, where n can be arbitrary large. On the other hand, in a behavior of a netlist, a particular event may occur only a finite number of times, even if the behavior is infinite. To bridge this difference, we add the special symbol $\perp$ to value domains of all types, as well as the value domain of LOC and ELOC formulas (i.e. the value domain of formulas is now $\{true, false, \perp\}$). Intuitively, this symbol indicates that a formula or a term has undefined value.

Now, we can define annotations for all quantity and variable names. Given an event e, integer n, and a meta-model netlist execution:

$$s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_i} s_i \xrightarrow{\sigma_{i+1}} \ldots \tag{4.1}$$

we first set $GXI(e, n)$ to be i such that e occurs in σ_i for the n'th time. More formally:

$$GXI(e,n) = \begin{cases} i \text{ s.t. } e \in \sigma_i \text{ and } |\{j \leq i | e \in \sigma_j\}| = n & \text{if such } i \text{ exists,} \\ \perp & \text{otherwise.} \end{cases}$$

Given event e, integer n, execution (4.1), quantity name `Q`, and variable name `X`, we define annotations Q and X as follows:

$$Q(e,n) = \begin{cases} \text{the value of } \texttt{Q.A(}e\texttt{,}n\texttt{)} & \text{if } \texttt{Q.A} \text{ is defined and } GXI(e,n) \neq \perp \\ \perp & \text{otherwise.} \end{cases}$$

$$X(e,n) = \begin{cases} \text{the value of } \texttt{X} \text{ in state } s_{GXI(e,n)-1} & \text{if } \texttt{X} \in Scope(e) \text{ and } GXI(e,n) \neq \perp \\ \perp & \text{otherwise.} \end{cases}$$

To deal with the new "undefined value" $\perp$, we add the following semantic rules for determining a value of a formula:

- evaluating any operator, except `&&`, `||`, `->` and `<->` gives $\perp$ if any of the operands have value $\perp$; the rule for the remaining four operators are given in Table 4.1
- the value of `exists`($\langle args \rangle$) $\langle wff \rangle$ is:
  - $true$, if there exists an assignment of values to the variables in list $\langle args \rangle$ such that the value of $\langle wff \rangle$ is $true$,
  - $false$, if the value of $\langle wff \rangle$ is $false$ for all assignments to variables in list $\langle args \rangle$,
  - $\perp$, otherwise,
- the value of `forall`($\langle args \rangle$) $\langle wff \rangle$ is:

Table 4.1: Extensions of Boolean operators to $\perp$.

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

- $true$, if the value of $\langle wff \rangle$ is $false$ for all assignments to variables in list $\langle args \rangle$,
- $false$, if there exists an assignment of values to the variables in list $\langle args \rangle$ such that the value of $\langle wff \rangle$ is $true$,
- $\perp$, otherwise,

Finally, we say that execution (4.1) satisfies ELOC formula ϕ, if the value of ϕ at (4.1) is not $false$.

4.5 Relating Executions of Networks

This section describes meta-model mechanisms to relate executions of two networks. The ability to do so is crucial to keeping functional and architectural models orthogonal, because then they can execute simultaneously due to the relations. This resembles the functional model being implemented or mapped to the architectural model.

In the meta-model, the relation is specified at event level by using ltl synch constraints. Its syntax is

```
  ltl synch (e1, e2, ..., en [: v1@(e1,i)==v2@(e2,i), ...]);
or
  ltl synch (e11 || e12 || ... || e1n => e21 || e22 || e2m
             [: v1@(e11,i)==v2@(e21,i), ...]);
```

In the first syntax, it says that all events from e1 to en must execute simultaneously, or none of them can execute. If the optional equality comparison part is given, additional conditions must also be satisfied in order to let all events execute. These conditions are specified by the equality comparison result made on two variables in the scope of two events respectively. This synch syntax can be used to model a group of simultaneous events, such as is often the case when a functional event is mapped to an architectural event. For example,

ltl synch (e1, e2 : b@(e1,i) == a@(e2,i));

This relation requires that the variable b in the scope of event e1 must be equal to

the variable a in the scope of event e2; at the same time, e1 and e2 must be enabled. If both conditions are satisfied, then these two events can occur simultaneously. In the equality comparison parts, the variable i is the global execution index. In synch, it should always be i, which means the current occurrence of the event.

Comparing with the first syntax, the second synch syntax offers more expressive power in specifying event relations. Now, it is not necessary that all events appearing in synch must execute simultaneously. The condition relaxes to that if any event on the left hand side executes, at least one event on the right hand side must execute. This event occurrence implication is usually used to model multiple functional events being mapped to a single architectural event, for instance the architecture provides a shared resource that will be used by more than one function. The semantics of the variable comparison part are the same as in the previous synch syntax, except that if an event is already disabled, then the equality comparisons involving variables in the scope of that event are considered satisfied. For example, in
ltl synch (e1 || e2 => e3 : b@(e1,i) == a@(e3,i), b@(e2,i) == a@(e3,i));
The following occurrence of events are all consistent with the semantics.

e1	e2	e3	b@(e1,i) == a@(e3,i)	b@(e2,i) == a@(e3,i)
T	T	T	T	T
T	F	T	T	-
F	T	T	-	T
F	F	T	-	-
F	F	F	-	-

In addition to relating events from functional and architectural networks, sometimes it is necessary to pass values from one side to the other in order to capture more precisely the system behavior. For example, the execution time of a hardware divider depends on the operands. In this case, it is desirable to pass operands to the divider. In order to achieve that, the metamodel provides a non-deterministic variable approach.

In the metamodel.lang package, there is a class called Nondet. It provides a mechanism to model a variable that could have nondeterministic values. The following is its definition.[1]

```
public class Nondet extends Object {
    int data;
    boolean nondet;

    public Nondet();
    public Nondet(int i);
    public void set(int i);
    public void setAny();
```

[1]Ideally, Nondet should be defined with template type data, not hard coded with integer type data. However, since the limitation of the current backend tools, we keep this temporarily.

```
    public int get();
    public boolean isNondet();
}
```

When a variable, say *a*, in the equality comparison part is defined as Nondet, the semantics of the equality comparison part becomes different. Instead of simply comparing the two variables, it will first check whether *a.isNondet*() returns true. If so, and the other variable, *b*, is either a regular variable or a Nondet variable but *b.isNondet*() is false, then the value of *b* (or *b.get*()) will be passed to variable *a* by calling *a.set(b)* (or *a.set(b.get())*). Note that in the expression v@(e,i), v could be not only a variable but also a constant, such as 123@(e,i).

Chapter 5

The Compilation

5.1 The Metropolis Infrastructure

The overall Metropolis infrastructure is shown in figure 5.1. It captures the Metropolis design methodology and tool infrastructure. For Metropolis design methodology and design guidelines, please refer to the document [4]. In this chapter, the focus will be put on the Metropolis tool infrastructure.

5.2 Compilation Flow

The Metropolis compilation flow starts from the user's design entry, which is written in Metropolis Metamodel language. See figure 5.2. The design entry is then passed to the Metropolis compiler frontend to perform syntax checking and semantics checking, and at the same time, generate abstract syntax tree (AST), the internal data representation. Then, based on user's need, one or more backend tools can be invoked. During the execution of the different tools, the AST is decorated or modified for later analysis purposes, such as simulation and verification.

The different steps described above can be invoked automatically by script commands provided by Metropolis. Right now, there are such script commands for metamodel file compilation (metacomp) and systemc based simulation (systemc). An equivalent but interactive way to perform the function of these scripts is to use the Metropolis shell (metroshell). With metroshell, users can invoke each step interactively, examine the result of each step, and have more freedom to invoke different backend tools without going through the compilation and possibly elaboration phase, etc.

In the following sections, we will introduce each of the key components in the compilation flow.

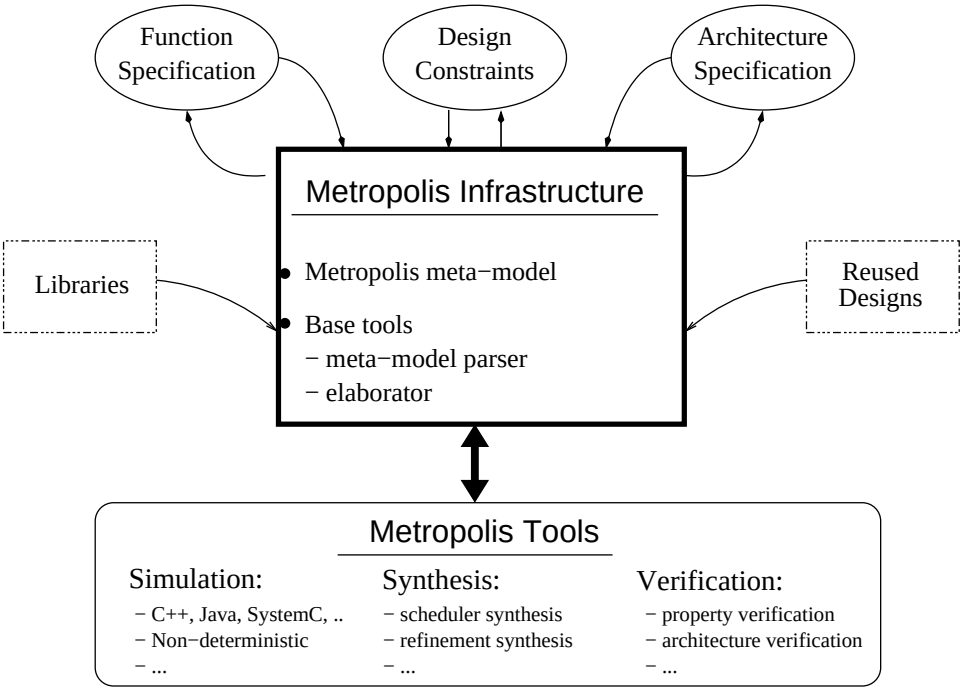

Figure 5.1: The Metropolis Infrastructure

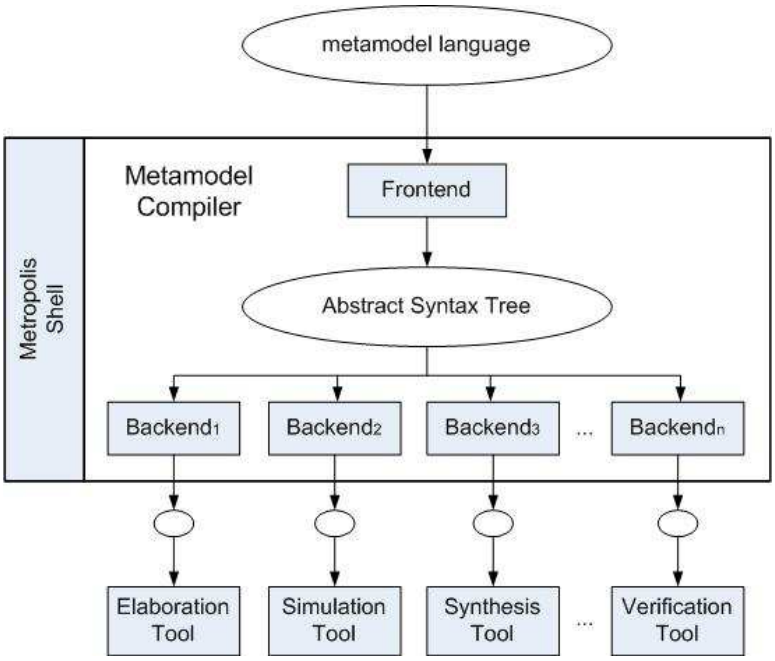

Figure 5.2: Metropolis Compilation Flow

5.3 The Frontend

The Metropolis frontend is in charge of syntax checking, template elimination and metamodel semantics checking. Portions of the code in the compiler frontend were derived from sources developed under the auspices of the Ptolemy II project. In the Metropolis frontend, there are several passes:

- Pass 0: Package resolution
  - Initialize the environments for semantic analysis (Scope, Decl)
  - Resolve import statements
  - Resolve type names
  - Additional classes might be loaded during this pass
- Pass 1: Class resolution
  - Add class members (fields, methods, ports, parameters) to the environments
  - Add inherited members to the environments
  - Additional classes might be loaded during this pass

- Pass 2: Template elimination
  - Locate all instantiations of templates in the source file. Identify the set of type instantiations of each template.
  - For each template and each type instantiation of the template, generate an "instantiated template" class file.
  - Replace all references to the template by references to the "instantiated template" class.
  - Of course, "instantiated template" class must be loaded and undergo at least pass 0 and pass 1.
- Pass 3: Name resolution
  - Resolve references to local variables, parameters and labels
  - Resolve references to class members (fields, methods)
- Pass 4: Meta-model checks
  - Check that the program is valid. At this point, we have enough information to proceed to the back-end. This pass can be time-consuming, and so it should be optional. It should be possible to skip this pass using compiler flags.
  - Perform thorough type-checking of all statements, expressions and variable initializations. This includes type-checking of meta-model constructs like await, non-determinism, pc, beg, etc.
  - Perform non type-related checks of meta-model statements.

For more information about each pass, please refer to the detailed description file metro/src/metropolis/metamodel/VISITORS.txt.

5.4 Abstract Syntax Tree

The abstract syntax tree (AST) is the standard way to represent programs in compiler technology. In the metamodel compiler, the AST is also used to represent the meta-model design entries. During compilation, the frontend will generate an AST for each metamodel file. Each node in the AST corresponds to a particular language construct. It could be a keyword, an operator, a user defined name such as a process name or variable name, or a higher-level construct such as an if-else structure.

Each node in the AST has the following attributes:

- an identifier of this class of nodes

- a list with a fixed number of children, e.g. a metamodel process declaration node will have children representing its modifiers, name, super process name and its body.
- a list of annotations, e.g. again a metamodel process declaration node will have annotations of its package, its own properties and a list of its members.

To make the manipulation of AST easy, a set of functions are defined. The basic operations that can be performed on an AST node include

- Retrieve its parent – getParent()
- Retrieve a child – getChild(index)
- Replace a child – setChild(index, val)
- Set/get methods for each child – e.g. getName()/ setName(val)
- Get the unique ID of the class – classID()
- Get a copy of a subtree – clone()
- Traverse the AST – accept(visitor, args)

The operations that can be used to decorate an AST node include

- Annotate the node – setProperty(index, val)
- Read an annotation – getProperty(index)
- Check existence of an annotation – hasProperty(index)
- Remove an annotation – removeProperty(index)

The Metropolis compiler (metacomp) can take a `-dumpast` argument, with which a textual view of the AST will be printed out to standard output. This is especially helpful to backend tool developers for understanding the program and AST structure.

Note: AST nodes are defined in file metro/src/metropolis/metamodel/NodeTypes.def. The corresponding Javadoc description is at metro/doc/codeDoc.

5.5 The Backend

Once the design entry goes through the frontend and the AST is generated, based on user's need, one or more backend tools can be invoked. Backend tools have a standard interface. Every one of them takes an AST and specific switches as input. They usually perform analysis first on the AST, and then output data of another sort in order to do analysis. For instance, the SystemC-based simulator will traverse the AST to collect the

built-in synchronization constraints (await and interface interactions) and user-specified explicit constraints (synch statements), and then optimize away unnecessary ones. This optimization result is annotated to the AST. In a later simulation code generation step, the backend tool will output optimized SystemC code.

For more details about how to develop a new backend tool or the description of existing Metropolis backend tools, please refer to the next section.

5.6 The Metropolis Interactive Shell

Metropolis shell (metroshell) gives users a textual interface to interact with the Metropolis compiler. With it, users can have finer control of the compilation process. Based on the need, they can invoke the compiler frontend and any other backend tools whenever they want. This is especially helpful to examine the result after applying an individual tool. For instance, there is a backend tool called elaborator. It will extract the system structure of the design entry. With metroshell, a user can invoke elaborator after feeding the design entry into the frontend. Then, the elaboration result is available for examination. This is very helpful when the system becomes very big or involves many refinements.

In the following, let us show the basic usage of metroshell by using the example in metro/examples/producers_consumer. Remember that whenever you need, you can type 'help' in metroshell to get help messages.

- Invoke metroshell
  > `metroshell` ↩

```
"$JAVAHOME/bin/java"    -Xms1g -Xmx1g
 -classpath "/users/berkeley/metro/src:/users/berkeley/metro/lib/ptjacl.jar::
   metropolis.metamodel.shell.Shell
 -classpath "/users/berkeley/metro/lib"
 /-------------------------------------------------------------------------\
 |         Metropolis: Design Environment for Heterogeneous Systems        |
 |                                                                         |
 |  Copyright (c) 1998-2004 The Regents of the University of California.   |
 |                          All rights reserved                            |
 \-------------------------------------------------------------------------/

 New to Metropolis? Type 'help' for information on the available commands.



 <> Loaded script '/users/berkeley/.metroshrc'
```

```
metropolis> _
```

- Set up classpath
  The classpath is used by the compiler frontend to look for user specified packages. It should include the parent directory of the user's top level package. In this example, classpath should have /users/berkeley/metro/examples in it. There are several ways to set up classpath.
  - In metroshell
    metropolis> `classpath show` ↩

    ```
    1 dirs in classpath
    /users/berkeley/metro/lib
    ```

    metropolis> `classpath add /users/berkeley/metro/examples` ↩

    ```
    Added '/users/berkeley/metro/examples' to classpath.
    ```

  - In environment variable METRO_CLASSPATH
    This environment variable should be set before invoking metroshell. It is a convenient way to specify classpath if metroshell will be run many times.
  - In metroshell script file
    Every time metroshell is invoked, it will look in the user's home directory for a metroshell script file called '.metroshrc'. If there is one, then it will execute it as if the commands in the file had been typed in by the user. Obviously, the user can add classpath into the script file.

- Read in user's design
  metropolis> `metroload pkg -semantics producers_consumer` ↩
  This command loads the user's package (pkg) producers_consumer into memory and performs semantics checks. If the design passes the checks of the frontend, nothing will be printed out after issuing this command, otherwise, the user will see error messages. Instead of loading in a package, there are other options like loading a class or a file. The user can also choose to perform checks other than semantics check. Please use help to find out more.

- Elaborate the design
  metropolis> `elaborate producers_consumer.IwIr` ↩

  ```
  Choosing a temporary directory...
  Finding java compiler and interpreter...
  Generating Java elaboration code...
  ```

```
Compiling elaboration code...
Running elaboration code...
```

In this command, the argument after `elaborate` is the top level netlist name. It must be a fully qualified name. After issuing this command, the elaboration backend is invoked. The response reflects the internal steps run by the elaborator. For details, please read the next section.

- Examine the system structure
  metropolis> `network show` ↩

```
Top-level netlist:
netlist producers_consumer.IwIr {
  o Instance name: top_level_netlist
  o Component name:
  o Components:
       - MEDIUM  (instance name: InstIntM)
       - DummyReader  (instance name: DummyR)
       - DummyWriter  (instance name: DummyW)
       - Producer0  (instance name: Producer0)
       - Producer1  (instance name: Producer1)
       - Consumer  (instance name: Consumer)
  o Not refined by a netlist
  o Does not refine any node
  o No constraints
}
```

After elaboration, the system structure is available. Issuing `network show` lists the components of the top level netlist. Recall that each component has one component name in each netlist and only one instance name. The names listed under 'Components:' are component names in the current netlist, and the names in parentheses are instance names. The user can use the same command to examine individual components by giving instance names, e.g. `network show InstIntM`.

- Simulate the design
  metropolis> `simulate systemc` ↩

```
Generating SystemC code...
```

We choose to do SystemC-based simulation. After entering this command, the special makefile 'systemc_sim.mk' is generated. Note that the user can also give the top level netlist after `simulate systemc`, in which case the SystemC backend will automatically invoke the elaborator.

Now, we can compile and run the SystemC simulation code in either a regular shell or within metroshell using `exec` command. e.g.
metropolis> `exec make -f systemc_sim.mk` ↩
Once the compilation finishes, we can run the executable, which is by default called 'run.x'.
metropolis> `exec run.x` ↩

Chapter 6

The Backends

6.1 Write a Backend Tool

The Metropolis tool infrastructure is open. It allows adding backend tools easily into the infrastructure by using the standard interface and by following the six steps below.

- Write a class that implements the Backend interface
  The Backend interface defines a single abstract function *invoke()*. This function takes two arguments. One is a list of ASTs, the other is a list of arguments that are passed from the metamodel frontend.

- Write the method *invoke* in that class
  The abstract function *invoke* must be implemented. It is the starting point of executing the backend tool. It needs to define what the arguments mean, and also control the execution of the backend.

- Write visitors to traverse the ASTs
  Visitor functions must be defined for the kinds of AST nodes that this backend tool is interested in. In case no visitor functions are defined for some kinds of AST nodes, the default visitors provided for them do nothing.

- Add flags to the compiler command-line
  After having the backend implemented, it must be linked into the Metropolis infrastructure. The compiler frontend needs to know the existence of this backend and set up appropriate command line arguments when invoking it. Note that the backend and command line arguments registration point is in metro/src/metropolis/metamodel/Compiler.java.

- Add the backend in makefile
  Another registration point for the new backend is the makefile in metro/src/metropolis/metamodel/backends directory. The new backend must be

residing in a subdirectory of the backends directory. In the makefile, the new subdirectory must be added to the DIRS variable, otherwise when building metropolis, this backend will be omitted.

- Add the package to the ALLPACKAGES variable in metro/doc/makefile
  It is always helpful to provide clear documentation for programs. In Metropolis, since most of the files are written in Java, we rely on javadoc to generate nice HTML documentation from the standard comments in the Java files. So, backend tool developers should keep in mind to comment their code as well as possible.

For details of how to write a backend, please refer to
metro/src/metropolis/metamodel/backends/metamodel_backend_howto.ppt.

6.2 Elaboration Backend

As described in section 2.6, netlists are used to represent the system structure. The construction of a netlist goes through four phases: object instantiation, adding objects as components, connecting components and refinements. The former three phases often mingle together. If we view all these phases in the entire compilation flow (see section 5 for details), they all belong to an elaboration backend. Since most of the backend tools need the elaboration results, therefore they invoke elaboration backend before doing their own work, sometimes we also say there is an elaboration phase in the compilation flow.

The goal of elaboration is to capture the system structure as well as to get constraints associated with each object. The reasons for introducing elaboration are these:

- Solely by static analysis, it is very hard if not impossible to fully capture the system structure. For example, the number of instances of a particular object depends on a variable passed in from another netlist, which can be decided only at run time. For another example, the constraints associated with an object can involve events defined in the object which depend on the object reference. Again, this information is not available until the instantiation of the object at run time.

- In the Metropolis infrastructure, there exist several analysis tools organized as backend tools. Not all of these tools can handle the construction of the network at run time. Some of the tools do not even have the corresponding concepts of construction. Therefore, having an elaboration phase reduces the efforts for other backend tools.

However, the separation of the elaboration phase also imposes a limitation, which is requiring the network to be fixed after the construction. This implies that no dynamic changes to the network are possible.

In Metropolis, elaboration is done by translating into Java code all metamodel code related to network construction, which includes constructors and all functions called

by constructors. Note that every function that will be called by a constructor must be declared with the keyword `elaborate`. Constructors themselves do not need the `elaborate` keyword. This keyword tells the elaborator to generate elaboration code for the function. Otherwise, for reasons of efficiency and easy handling, the function will be ignored by the elaborator. After the translation, the Java code will be compiled and executed, constructing the network as described in metamodel code. Upon finishing the construction, the elaborator will run functions called postElaborate if there are any defined in metamodel objects. This gives users a chance to look at the network structure and if necessary set up the post-elaboration information accordingly.

6.3 Elaboration Testing Backend

RuntimeTestBackend is essentially a testing backend added for testing runtime library, constraint elaboration and other elaborator features. The only thing it does is to elaborate the design, manipulate the elaborated network and print out the results. Users are supposed to modify this backend and use it to test their own added features of runtime library and elaborator.

For more details, please refer to
metro/src/metropolis/metamodel/backends/runtimetest/README.txt

6.4 Compilation Backend

This backend is usually used for checking syntax at the early stage of design cycles. It simply parses the metamodel input files and then re-generate metamodel code from the ASTs. It is also able to generate a textual view of the AST, which is extremely helpful for backend tool developers to understand the AST structure.

For more details, please refer to
metro/src/metropolis/metamodel/backends/metamodel/README.txt

6.5 Formal Verification Backend

The promela backend is essentially a translator from Metropolis designs to Promela, a formal verification language of the model checker Spin. Spin is a formal verification tool for asynchronous software systems and is chosen as a backend verification engine.

The metamodel description is automatically translated into Promela description, and the properties are checked using SPIN model checker. The designer may perform any synthesis step (e.g. composition, decomposition, constraint addition, scheduler assignment) and a new Promela code can be automatically generated to verify the property. If it does not pass, the error trace may be used to help designers figure out whether the design needs to be altered. If the verification session runs too long, approximate verification can be used to explore a subset of the state space and report the probability that

the property will pass. Obviously, a partial exploration can not prove that a property holds.

For more details, please refer to
metro/src/metropolis/metamodel/backends/promela/README.txt

6.6 Simulation Backend

SystemC-based simulator is currently the main validation tool used in Metropolis infrastructure. It starts with the abstract syntax tree (AST) generated by the Metropolis compiler frontend. After performing all applicable optimization techniques, it generates SystemC code for the metamodel description while maintaining the metamodel semantics. During code generation, it also produces a make file systemc_sim.mk for compilation purpose. Once the SystemC code is compiled and linked with the simulation libraries provided by both SystemC and this particular backend tool, the executable will show the metamodel simulation result.

For more details, please refer to
metro/src/metropolis/metamodel/backends/systemc/README.txt

6.7 Debugging Backend

mgdb is a debugger for executable simulations built with the systemc backend. It is Gnu's "gdb" debugger with custom commands that make use of special code added to the C++ code generated by the systemc backend. It allows debugging the metamodel (.mmm) source code directly.

For more details, please refer to
metro/src/metropolis/metamodel/backends/mgdb/README.txt

Bibliography

[1] Balarin F, Lavagno L, Passerone C, Sangiovanni-Vincentelli A, Sgroi M, and Watanabe Y. *Concurrency and hardware design. Advances in Petri nets*, chapter Modeling and designing heterogeneous systems, pages 228–73. Springer-Verlag, 2002.

[2] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[3] Sun Microsystems. *Java 2 Platform, Standard Edition v1.3.1 API Specification*, 1999. http://java.sun.com/j2se/1.3/docs/api/index.html.

[4] Alessandro Pinto. Metropolis design guidelines. Technical report, Technical Memorandum UCB/ERL M04/40, University of California Berkeley, CA 94720, 2004.

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

Acknowledgements

This work was supported in part by the following corporations:
* Cadence,
* General Motors,
* Intel,
* Semiconductor Research Corporation (SRC),
* Sony,
* STMicroelectronics;
and the following research projects:
* NSF Award Number CCR-0225610 and the Center for Hybrid and Embedded Systems (CHESS, http://chess.eecs.berkeley.edu),
* The MARCO/DARPA Gigascale Systems Research Center (GSRC, http://www.gigascale.org),

The Metropolis project would also like to acknowledge the research contributions by:
* The Project for Advanced Research of Architecture and Design of Electronic Systems (PARADES, http://www.parades.rm.cnr.it/) (in particular Alberto Ferrari), 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 and University of Pisa.

Metropolis contains the following software that has additional copyrights. See the README.txt files in each directory for details

examples/yapi_cpus/arm/arm_sim
   arm_sim is an ARM processor simulator that was originally released under the GNU Public License.
   The ARM Simulator is only necessary if you would like to create your own trace files. Most users need not build the ARM Simulator.

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

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

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

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

Appendix A

Keywords

A.1 Primitive Data Types

void boolean char byte short int float long double event

A.2 Literals

true false null all super this LAST retval

A.3 Modifiers

abstract final private protected public static

A.4 Effects

constant eval update elaborate

A.5 Object Declaration

package import process medium interface netlist statemedium quantity class template extends implements port parameter useport

A.6 Control Flow Statements

do while for boundedloop switch case default if else break continue return await

A.7 Other Keywords

beg end none other blackbox label block nondeterminism new

A.8 Constraints

constraint ltl loc eloc forall exists GXI F G U X lfo excl synch mutex simul priority mindelta maxdelta minrate maxrate

A.9 Network

addcomponent getcomponent getcompname getinstname getprocess getthread connect getnthport getnthconnectionsrc getnthconnectionport getconnectiondest getportnum getconnectionnum refine refineconnect redirectconnect isconnectionrefined getscope setscope

A.10 Reserved Keywords

pc pcval gettype scheduler instanceof

A.11 Illegal Java Keywords

catch const finally goto native synchronized throw throws transient try volatile

A.12 Arithmetic and Logical Operations

+, −, *, /, ++, −−, +=, −=, *=, /=, <<, >>, >>>, <<=, >>=, >>>=, &=, ∧=, |=, =>, − >, < − >, &&, ||, ==, !=, <=, >=