MetaTeD — A Meta Language for Modeling Telecommunication Networks

Kalyan S. Perumalla and Richard M. Fujimoto
(kalyan@cc.gatech.edu and fujimoto@cc.gatech.edu)
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Andrew T. Ogielski
(ato@dimacs.rutgers.edu)
WINLAB
Rutgers University
Piscataway, NJ 08855

GIT-CC-96-32
December 31, 1996
(Last Updated June 13, 1998)

ABSTRACT

TeD is a language designed mainly for modeling telecommunication networks. The **TeD** language specification is separated into two parts — (1) a *meta* language (2) an *external* language. The meta language specification is concerned with the high–level description of the structural and behavioral interfaces of various network elements. The external language specification is concerned with the detailed low-level description of the implementation of the structure and behavior of the network elements. The meta language, called **MetaTeD**, is described in this document (An external language specification, with $\mathbf{C}^{++}$ as the external language, is described in a separate related document.).

Contents

1 Introduction

TeD is a language designed mainly for modeling telecommunication networks. The **TeD** language specification is split into two distinct parts — **MetaTeD** and "external language." **MetaTeD** defines a set of concepts for modeling the dynamic interactions of *entities* and their compositions, in an application-independent manner. **MetaTeD** is an incomplete language — it is more appropriately called a "framework." When **MetaTeD** is appropriately combined with any regular general-purpose programming language, say $\mathcal{L}$, then a complete language is formed. Such a complete language is called an $\mathcal{L}$-instance or $\mathcal{L}$-flavor of **TeD**. For example, see [2] for the description of a **C**$^{++}$-instance of **TeD**.

In this document, only the **MetaTeD** concepts are described. Thus, the descriptions in this document apply to all *instances* of the **TeD** language irrespective of the external language used in the *instance*.

2 Background

Some of the main objectives in the design of the **TeD** language are:

- The language should be sufficiently general for modeling current as well as future telecommunication networks.
- The same language should provide for *specification*, *description* as well as *simulation* of models (implies simulation-independent description).
- The language should support object-orientation — encapsulation, inheritance and polymorphism — to facilitate hierarchical structure and behavior description.
- Models expressed in the language should be amenable to high-performance parallel/distributed simulation (through automatic or semi-automatic translation).
- The language should provide support for user-definable and extensible libraries.

Our basic approach towards the design of such a language was as follows. The analogous language for hardware/VLSI domain, namely, the VHSIC Hardware Description Language (VHDL), was examined[1]. Several concepts were borrowed from VHDL and suitably modified to suit the telecommunication domain. Some VHDL concepts that are specific to the hardware domain were ignored, and new concepts appropriate to the telecommunication domain were added. Also, support for object orientation was added to aid development of structural and behavioral heirarchies in the models. The result of this exercise is a new language, which is an object-oriented, simulation-independent language for modeling telecommunication networks. Throughout the language design, emphasis was placed on strong modularity, and the separation of structure from behavior, while at the same time retaining the ability to analyze the models through *parallel* simulation.

Orthogonality of Concepts

In the process of designing the language, it was observed that the set of concepts supported in this language — such as, elements of dynamic interaction among entities — is orthogonal to the set of concepts addressed in regular general-purpose programming languages — such as, data types and control flow. Most modeling languages (VHDL, for example) combine these two orthogonal sets of concepts into one specification language, even though it is often possible to mix and match them together.

Hence, it was decided to carefully separate the two sets of concepts. The concepts that specify the relationships and dynamic interactions of the entities in the modeling domain are termed the meta language **MetaTeD**. **MetaTeD** is datatype-unaware[1]. When **MetaTeD** is suitably combined with any given regular programming language (called the "external" language), a concrete *instance* of the **TeD** language is formed. For example, a $\mathbf{C^{++}}$ *instance* of **TeD** is described in [2].

It is interesting to note that the use of an external language is similar to the concept of "outsourcing"; in the context of the language specification, it is wise to utilize the well-researched and well-developed constructs of other languages, and their well-tested tools, for the purposes of datatypes and control flow, instead of inventing and standardizing yet another general purpose language.

In the rest of the document, we shall use the terms **TeD** and **MetaTeD** interchangeably where no ambiguity exists. Also, by default, we shall use $\mathbf{C^{++}}$ as the external language, following the interface described in [2], in all the examples.

3 A Tutorial

In the following tutorial, the information on the constructs and semantics may be incomplete in the interest of clarity of basic introduction to the various concepts supported in the language.

The following are some of the fundamental concepts supported in the **TeD** language:

1. Entity
2. Event
3. Channel
4. Architecture
   - State
   - Behavior
     - Process
     - Component
   - Result

In the following sections, the basic terminology and semantics are introduced.

3.1 Overview of Concepts

Figure 1 illustrates the basic framework underlying the constructs in **TeD**, and their relation to each other.

[1] With the exception of integers; it recognizes integers since it needs to "know" how to "count", in order to be able to define parametrized structures (varying number of entities and channels).

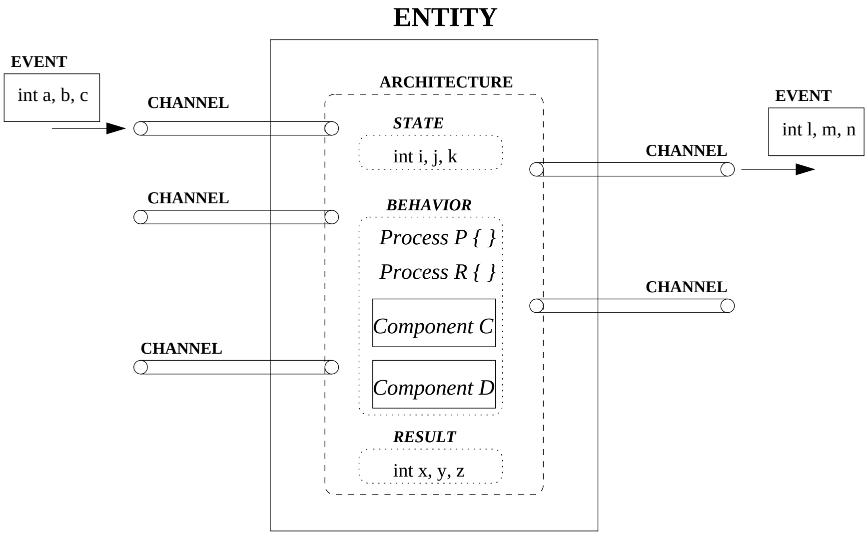

Figure 1: Basic framework of an Entity

In **TeD**, physical and conceptual objects in the telecommunication domain are modeled in terms of *entity* descriptions. Entities are connected to- and interact with each other via *channels*. Channels are the only means of dynamic interaction between entities. Information units, called *events*, flow through channels. An output channel of one entity can be connected to the input channel of another entity. This allows for modeling of an arbitrary structure of connected entities.

The dynamic behavior of each entity is described by its *architecture*. The architecture of an entity is expressed using concurrent process semantics. It is essentially described in terms of the actions of the entity upon event arrivals on its input channels, and the production of events on its output channels. One or more processes can be defined to act upon events arriving on the input channels of the entity. The processes may generate events on output channels as part of their computation/action. An event generated on an output channel arrives as input on the input channel to which the output channel is *mapped*. The architecture of an entity can also be expressed as as a composition of interacting *components*. Components are nothing but entities themselves (see figure 2) "logically enclosed" inside the bigger entity. The channels of the components can be arbitrarily mapped among each other or to the channels of the enclosing entity. Thus, the behavior of an entity can be described in terms of its structure (components) or dynamics (processes), or a combination of both.

Entities are highly modular. Except for the interaction through their channels, entities' state and behavior are completely encapsulated and invisible to other entities. The entity construct describes the entity's external input/output view, while the architecture construct describes the internal behavioral part of the object being modeled. More than

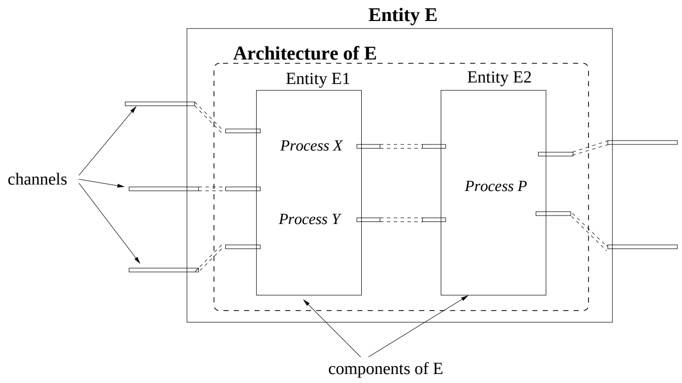

Figure 2: Entity with sub-entities

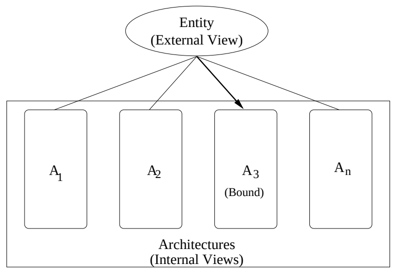

Figure 3: Single Entity – Multiple Architectures

one architecture can be defined on/for an entity; however, exactly one of them must be chosen and bound to the entity for simulation-based analysis (see figure 3).

Structural and behavior descriptions of an entity can be inherited by other similar entities, thus allowing object-oriented hierarchy-based design and development. Inherited properties can be specialized, redefined or overridden by the inheriting entity (sections 5.1 and 5.4 describe structure and behavior inheritance semantics).

Figure 4 is an illustration of some of the basic elements of a model and their relationship with respect to the physical object being modeled.

3.2 Entity Declaration

The entity construct is used to define the external view of a physical or conceptual object. It essentially defines the input and output specifications for the entity. In particular, it does *not* define any internal behavior.

The entity construct for the simple ATM multiplexer shown in figure 4 is as follows:

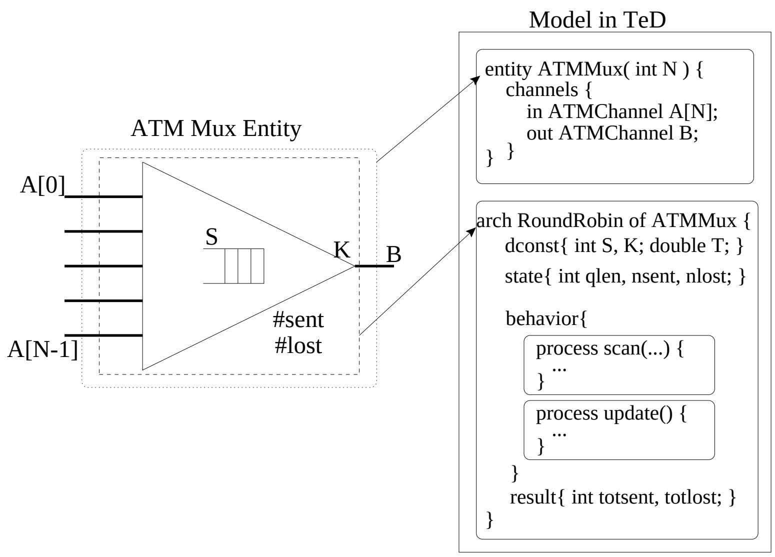

Figure 4: Illustration of a physical entity and its model in **TeD**

```
entity ATMMux ( int N )
{
        channels
        {
                in ATMChannel A[ $PARAM(N)$ ];
                out ATMChannel B;
        }
}
```

In the above entity declaration, `N` is a parameter that defines the number of inputs. The multiplexer is defined to have `N` input channels, `A[N]`, and one output channel, `B`. Each of the channels is of type `ATMChannel`, as defined in the next section.

3.3 Event and Channel Declarations

A channel is a port of input or output for an entity. An output channel of an entity can be mapped to an input channel of another entity.

An event is a unit of information that flows through channels. An event that is sent on an output channel, say, A in entity X, will appear as an arrival on the input channel, say, B in entity Y, if channel A is mapped to channel B. Each event type definition consists of the name of the event and the data associated with the event. Any number of event types can be defined per application.

A *channel type* is defined as a set of event types. The channel type essentially defines that only those events belonging to the defined set of event types are allowed to "flow" through a channel of the given type. Also, two channels can be mapped (connected) to each other if and only if they are of the same channel type (or *compatible* channel types — see section 5.3).

In the preceding ATM multiplexer example, the `ATMChannel` type can be defined as follows:

```
event ATMCell {$ char data[ 53 ]; $}
channel ATMChannel { ATMCell }
```

The `ATMCell` declaration specifies an event type and its associated data. The `ATMChannel` declaration defines that any channel of type `ATMChannel` allows only `ATMCell` events to flow through it. Any number of event types can be specified in the list of event types for defining a channel type.

The same event and channel types can be used in the definition of any number of entity types. In fact, sharing the same event and channel definitions for several entity definitions aids in ensuring that those entities can be interconnected.

3.4 Architecture Declaration

The architecture of an entity defines the internal behavioral model of an entity. The architecture description mainly consists of three parts — the *state*, the *behavior*, and the *result*. The state is the set of variables that are used to denote the current status of the entity. The result consists of the set of variables that abstract the result of simulation of that entity. The behavior describes how the entity reacts on information arrival on its input channels, and how the entity generates information onto its output channels. Two concepts — *process* and *component* — are available for the purpose of expressing the behavior. A process is a set of statements that are executed upon event arrival on an input channel. A component is just another entity, also called a sub-entity; thus, part of the behavior of the (enclosing) entity is expressed in terms of this (enclosed) entity. Any number of processes and components can be used in the architecture. Also, processes and components can both be used together in the same architecture.

To illustrate, figure 4 shows an internal view of the multiplexer of figure 4. In this view, the multiplexer uses a finite buffer of size `S`, and the output link of the multiplexer is capable of transferring `K` cells in one cell arrival time on any of the input links. Since the current occupancy of the buffer is the only state information required, a variable `qlen` is used to represent the state. Additional measures, such as the cumulative number of cells transferred and the number of cells dropped due to buffer overflow, can also be added to the state. In such a case, this architecture may include those statistics as part of its result. For the description of the multiplexer's dynamic behavior, two processes are defined — one process, `scan`, acts on cell arrivals on the input channels and enqueues them into the buffer, while another process, `update`, models the transfer of cells from the buffer onto the output link. The architecture construct for this behavior of the simple ATM multiplexer is as follows:

```
architecture RoundRobin of ATMMux ( int N )
{
    dconst{$ int S, K; double T; $}
    state{$ int qlen, nsent, nlost; $}
    behavior
    {
        process #1 scan( A );
        process #2 update;
    }
    result{$ int totsent, totlost; $}
}
```

The initialization of the state and result variables, and the definition of the code for the two processes `scan` and `update` are not defined here in the interest of brevity. Concrete examples, with complete source code and explanations, are given in [2].

3.5 External Code Block

In **MetaTeD** models, external language expressions and/or declarations are used for certain purposes. Such inserted code segments are called external code blocks. External code blocks are any valid external language expressions (or declarations, depending on the context) enclosed between a pair of $ signs, or between `\{` and `\}`[2]. The interface provided by the external language environment to the external code blocks in **MetaTeD** models depends on the model development support provided for the particular external language.

[2]Linebreaks are not allowed to appear between $ signs.

4 Syntax Reference

Keywords are set in **bold**. User-defined items are denoted in *italics*. Optional items are enclosed in [square brackets]. Alternative items are separated by |.

---

use "*model-unitname*"

---

import interface | **implementation** "*ext-code-unitname*"

[**sure**] **event** *event-type* { *ext-var-decls* }

---

channel *channel-type* [**inherit** *base-channel-type*] { *event-types* }

---

entity *entity-type* [(*param-decls*)] [**inherit** *base-entity-type* [(*param-decls*)]]
{
    channels { *channel-decls* }
}

where, *param-decls* is comma-separated list of one or more of:
    int *var-name*
and, *channel-decls* is zero or more of:
    [**overload**] **in** | **out** | **inout** *channel-type channel-var-name* [[*ext-expr*]] **;**

---

architecture *arch-type* [(*param-decls*)] **of** *entity-type* [(*param-decls*)]
    [**inherit** *base-arch-type* [(*param-decls*)]]
{
    dconst { *ext-var-decls* }
    state { *ext-var-decls* }
    lstate { *ext-var-decls* }
    channels { *channel-decls* }
    behavior
    {
        procedure #*N procedure-name* [(*ext-arg-decls*)] **;**
        process #*N process-name* [(*channel-vars*)] [**overload**] **;**
        component *component-block-name* **;**
        [**abstract**] **function** *function-name* [(*ext-arg-decls*)]
            [**returns** *ext-type*] [**overload**] **;**
    }
    result { *ext-var-decls* }
}

---

dconst *entity-type* **:** *arch-type* **: init**
{
    ext-statement
}

---

state *entity-type* **:** *arch-type* **: init**
{
    ext-statement
}

---

lstate *entity-type* **:** *arch-type* **: init**
{
    ext-statement
}

component *entity-type* **:** *arch-type* **:** *component-block-name*
{
  entities { *entity-instantiations* }
  map { *mapping-statement* }
  wrapup { *result-computation* }
}

---

where, *entity-instantiations* is zero or more of:
entity-name **=>** *entity-type* [*param-vals*] **:** *arch-type* [*param-vals*] [[*ext-expr*]] **;**
mapping-statement and *result-computation* are *ext-statements*, and
param-vals is **%***n***(** *ext-args* **)**

---

process #*N* *entity-name* **:** *arch-name* **:** *process-name* [**(** *channel-vars* **)**]
  [**waits** { *wait-var-decls* }]
  [**calls** { *call-var-decls* }]
  [**local** { *ext-var-decls* }]
{
  *wait-statement*s and/or *compute-statement*s
}

---

where, *channel-vars* is comma-separated list of one or more of:
  channel-var [**[** *ext-expr* **to** *ext-expr* **]**]
compute-statement is:
  ext-statement
wait-statement is:
  wait *wait-clause* **;**
wait-var-decls are zero or more of:
  var-name1 **,** *var-name2* **,** ... **:** *wait-clause* **;**
wait-clause is:
  on *channel-vars* **until** *condition* [**surely**] **for** *expression*
condition and *expression* are:
  *ext-expr*s
and, *call-var-decls* are zero or more of:
  var-name1 **,** *var-name2* **,** ... **:** *procedure-name* **;**

---

procedure: exactly same syntax as for **process**, except that *channel-vars* must be replaced with *ext-arg-decls*.

---

function *entity-type* **:** *arch-type* **:** *function-name* [**(** *ext-arg-decls* **)**]
  [**returns** *ext-type*]
{
  ext-statement
}

---

result *entity-type* **:** *arch-type* **: init**
{
  ext-statement
}

---

result *entity-type* **:** *arch-type* **: wrapup**
{
  ext-statement
}

ext-code-block: Any external language source code enclosed between two $ signs, or between `\{` and `\}`; newlines are *not* allowed between $ signs.
ext-var-decls: *ext-code-block* containing variable declarations
ext-expr: *ext-code-block* containing a valid expression
ext-type: *ext-code-block* specifying a single data-type
ext-arg-decls: *ext-code-block* specifying "formal argument list"
ext-args: *ext-code-block* containing "actual argument list"
ext-statement: *ext-code-block* containing executable statement(s)

5 Semantics

The following sections describe the detailed semantics of each of the concepts in the **TeD** language.

5.1 Entity

The *entity* declaration is used to define a black-box view of an entity. The entity could be a direct model of a real-life object, such as a multiplexer, or of an abstract object, such as a protocol. The entity declaration only presents an external structural view of the object. It does *not* define any behavior of the object. The syntax of the **entity** specification is as follows:

entity *entity-type* [(*param-decls*)] [**inherit** *base-entity-type* [(*param-decls*)]]
{
  channels { *channel-decls* }
}

---

where *param-decls* is comma-separated list of one or more of:
  int *var-name*
and, *channel-decls* is zero or more of:
  [**overload**] **in** | **out** | **inout** *channel-type channel-var-name* [[*ext-expr*]] **;**

Entities can have zero or more channels (called the interface channels), each of **in**, **out** or **inout** modes. Channels are typed (see Section 5.3 for a description of channel type declarations). The behavior of an entity is defined solely in terms of the entity's dynamic reactions to events on its input-mode channels, and the production of events on its output-mode channels.

5.1.1 Inheritance

An entity, `E2`, can inherit the structure of another entity, `E1`, by using the `inherit` clause as follows:

entity E2 **inherit** E1
{
  ...
}

`E2` will contain all the interface channels of `E1`, in addition to any interface channels defined in `E2`.

5.1.2 Interface Enhancement

Entity `E2` may "enhance" its interface by expanding the channel types of some or all of the inherited interface channels. For example, consider the following declarations.

```
event T1 { ··· }
event T2 { ··· }
channel C1 { T1, T2 }
entity E1
{
      channels { in C1 A; }
}
```

```
event T3 { ··· }
channel C2 inherit C1 { T3 }
entity E2 inherit E1
{
      channels { overload in C2 A; }
}
```

`E2` enhanced its interface by including acceptance of event type `T3` on its inherited input channel `A`. Note that any behavior defined for `E1` will still be valid for `E2`. The behavior can only be extended additionally to act on events of type `T3` arriving on `A`.

Another type of enhancement is the promotion of the mode of an interface channel from either **in** or **out** to **inout**. Both the mode and the type of an inherited channel can be enhanced by the inheriting entity.

Note: If it is desired to enhance a single interface channel to an array of channels, it is necessary to define the original channel to be an array of length 1. Thus, the following is invalid:

```
entity E1
{
      channels { in C A; }
}
entity E2 inherit E1
{
      channels { overload in C A[ $100$ ]; }
}
```

Instead, `E1` must be defined as follows to achieve the desired effect.

```
entity E1
{
      channels { in C A[ $1$ ] ; }
}
```

5.1.3 Parametrized Structures

The parameter values can be used in the specification of sizes of channel arrays, thus effectively defining parametrized structures of entities. For example, a multiplexer entity parametrized by an integral value `N` can be defined as follows.

```
entity E( int N )
{
      channels { in C A[ $ PARAM(N) $ ]; }
}
```

5.1.4 Implicit Parameter

An implicit parameter, **I**, is predefined for every entity. It denotes the index of the entity instance if the entity happens to be an element in an array of entities at the time of model execution. If the entity instance is not an element of an array, then the value of **I** is -1.

No declared parameter of any entity must be named **I**.

5.1.5 Parameter Lists

The parameter lists are concatenated when refering to inherited entities. The following declarations illustrate the concatenation order.

```
entity E1( int p1, int p2 )
{
      ...
}
entity E2( int p3, int p4 ) inherit E1( int p1, int p2 )
{
      ...
}
entity E3( int p5 ) inherit E2( int p3, int p4, int p1, int p2 )
{
      ...
}
```

5.2 Event

Events are units of information exchanged between two different entities or between processes and components within an architecture of an entity. Events are "sent" through the interface channels between entities, or through the internal channels within the architecture of an entity. The following is the syntax for declaring an event type:

```
[sure] event event-type { ext-var-decls }
```

The external language shall provide a way to instantiate an event and to initialize the member variables of the event. No system-dependent information is provided to distinguish between any two instances of the same event type. If such a distinction is required, then distinguishing information must be included as event variables in an application-dependent fashion.

The event may be sent and received in entities which are in two different address spaces; thus no assumptions may be made about shared address space. Thus, use of pointer values in events may not be supported by the external language.

Upon receipt of events on the channels of an entity, one or more actions can be performed by the **process**es of that entity's architecture(s) (see Section 5.4.5). These actions, in general, could potentially be executed more than once *for the same event* during the simulation of the model (for example, if optimistic parallel simulators are used). In certain cases (such as, if input/output operations, or any other non-idempotent operations, are to be performed as part of the event processing), it is desirable for some events to be tagged so that the actions associated with their arrival are executed *exactly once* per event arrival. Thus, if an event type is tagged with the **sure** keyword, then it is guaranteed that the actions upon arrival of any instance of the given event type are executed exactly once per instance during the simulation.

5.3 Channel

Channels form the medium for inter-entity or intra-architecture communication. Events flow through channels. The syntax for declaring a channel type is as follows:

channel *channel-type* [**inherit** *base-channel-type*] { *event-types* }

The channel type declaration specifies the types of events that are permitted to "flow" through any channel of that channel type. Thus, a channel type is the definition of a subset of the set of all event types. Only instances of event types belonging to that subset are allowed to be sent and received on channels of the given channel type.

All channels are of **inout**-mode by default. It is an error to send events on an **in**-mode channel, or listen on an **out**-mode channel.

No more than one event can exist at any given moment in simulated time on a channel. If an event "arrives" on an end of a channel while another event is "active," then the new event destroys the old event.

5.3.1 Inheritance

When a channel type, `C2`, is declared to inherit another channel type, `C1` (called the *base-channel-type* of `C2`), it implies that `C2` includes all the event types of `C1` in addition to those defined for `C2`.

A channel type, `C2`, is said to be a *superset* of another channel type, `C1`, if and only if `C2` is declared to inherit `C1`, either directly or by an inheritance-chain.

5.3.2 Mapping

A channel can be *mapped* to another channel[3]. By default, external (interface) channels remain unmapped to any channels, while internal channels map to themselves. It is valid to send events on unmapped external channels; such events go into oblivion. The default mapping (to themselves) of internal channels can be changed by mapping them to channels of component entities. However, the internal channels cannot be reverted to unmapped condition.

Two channels can be mapped to each other only if they are *compatible* with each other. Mapping compatibility of channels is defined as follows.

[3] Current specification allows for at most one channel to be mapped to any given channel.

```
mode1 C1 ch1;
mode2 C2 ch2;
```

Given the preceding declarations, channels `ch1` and `ch2` are mapping-compatible if all the following conditions are satisfied:

- Either *C1* and *C2* must be the same type, or one must be a superset of the other.
- If *C1* is the same as *C2* then:
  mode1 and *mode2* cannot be the same, unless both are **inout**.
- If *C2* is a superset of *C1*, then
  mode2 must be **in**, and *mode1* must be **out** or **inout**.

  Similarly, if *C1* is a superset of *C2*, then
  mode1 must be **in**, and *mode2* must be **out** or **inout**.

5.4 Architecture

The behavior of an entity is defined in terms of its *architecture*. While the entity declaration is analogous to an "interface" specification, the architecture declaration is analogous to an "implementation" specification. More than one architecture can be defined for the same entity type. For example, one architecture may be used to model the entity at at very fine level of granularity of behavior using several *process*es and *component*s, while another architecture for the same entity may be defined for a coarser specification of behavior, perhaps using just one process.

The following is the syntax for declaring an architecture of an entity:

```
architecture arch-type [( param-decls )] of entity-type [( param-decls )]
      [inherit base-arch-type [( param-decls )]]
{
      dconst { ext-var-decls }
      state { ext-var-decls }
      lstate { ext-var-decls }
      channels { channel-decls }
      behavior
      {
            procedure #N procedure-name [( ext-arg-decls )] ;
            process #N process-name [( channel-vars )] [overload] ;
            component component-block-name ;
            [abstract] function function-name [( ext-arg-decls )]
                  [returns ext-type] [overload] ;
      }
      result { ext-var-decls }
}
```

The architecture of an entity is divided into the following sections:

- **Parameters** (*param-decls*): A set of integer variables that are used in defining parametrized strucutures/templates of entities and architectures.

- **Deferred Constants (dconst)**: A set of items whose values could be different for different instances of entity, even if the instances are of the same entity and architecture types.
- **State (state)**: A set of variables that together form a part of the abstract state of the modeled entity.
- **Large State (lstate)**: A set of variables that are similar to the state variables, with the difference that the memory size of these variables could be significant.
- **Internal Channels (channels)**: A set of channels that are used for communication among the internal processes and components of an architecture.
- **Behavior Processes (behavior-process)**: A set of threads of computation that act on the events arriving on the interface and internal channels.
- **Behavior Components (behavior-component)**: A set of entities that *logically* form subentities of an entity's behavior.
- **Result (result)**: A set of values that are an abstraction of the "result" of "execution" of the model.

Each of these concepts is described in the following sections.

Inheritance

An architecture of an entity can *inherit* from another architecture of the same entity type or from an architecture of an entity type that is inherited by the given entity type.

An inheriting architecture inherits all the items of the inherited architecture. Thus, Parameters, Deferred constants, State, Large State and Result variables, processes and component blocks are all inherited. All inherited items are *visible* in all inheriting architectures. There is no concept of limiting the scope of visibility in the inheritance of architectures.

5.4.1 Parameters

Parameters are variables of type **int**. They are mainly designed to be used in the definition of parametrized structures. The parameters of entities can be used in the definition of external interface structure (external channels), while the parameters of architectures can be used in the definition of internal communication interface (internal channels).

The external language shall provide some way of accessing the values of the entity parameters in the declaration of external channels. Similarly, architecture parameters shall be available in the declaration of internal channels. Both types of parameters shall be accessible inside the **process**es and **function**s of the architecture.

When specifying parameters for an entity-architecture pair, the parameter lists of the architecture and the entity are concatenated, with architecture parameters first. If an architecture inherits another architecture, then their parameter lists are concatenated, with the inheriting architecture's parameters first.

5.4.2 Deferred Constants

Deferred Constants are those that can be set to different values for different instances of entities. The external language interface shall provide ways to initialize the constants to user-specified values, either at compile-time or "later than compile time". Facilities shall be provided to set different values to the deferred constants of different entity instances even if the instances are of the *same* entity and architecture types.

The names and types of the deferred constants are declared in the **dconst** section of the architecture declaration. The syntax for the declaration of deferred constants inside an architecture is given in section 5.4. The setting of instance-dependent values to these constants is performed in the **dconst-init** construct outside of the architecture declaration, with the following syntax.

dconst *entity-type* **:** *arch-type* **: init**
{
    ext-statement
}

Once the values of the **dconst** variables are set, they remain constant throughout the execution of the model; the external language may enforce this by providing readonly access to the **dconst** variables, thus preventing their modification.

In the initialization of **dconst** variables, the values of entity and architecture parameters of the object are available for use (parameter values are set before deferred constants are initialized).

If the architecture inherits another architecture, the deferred constants of the inherited architecture are initialized before those of the inheriting architecture are initialized.

The external language shall provide some way of accessing the values of the deferred constants in the declaration of internal channels, and inside the **process**es and **function**s of the architecture.

5.4.3 State

State variables are those that together constitute the internal data representation of the object's behavior. Given any instant in time, the values of the state variables together uniquely and sufficiently define the object. However, these values vary with time — this variation constitutes a part of the dynamic behavior of the object.

The syntax for the declaration of state variables is included in the architecture syntax. The syntax for the definition of initialization of the state variables is as follows.

state *entity-type* **:** *arch-type* **: init**
{
    ext-statement
}

In the state initialization, the values of entity and architecture parameters of the object, and the values of its deferred constants are available for use (deferred constants are initialized before state variables are initialized). If the architecture inherits another architecture, the state variables of the inherited architecture are initialized before the state variables of the inheriting architecture are initialized.

The external language shall provide some way of accessing the state variables inside the **process**es and **function**s of the object.

5.4.4 Large State

```
lstate entity-type : arch-type : init
{
        ext-statement
}
```

The external language interface is free to implement this set of variables in an application-dependent fashion. On one extreme, these variables could be merged with the **state** variables. On the other extreme, severe restrictions on the datatypes, operations and access of these variables may be placed. The **lstate** construct is considered a "non-standard" part of the language, and not all external languages and all **TeD** modeling tools may support it.

Similar to state variables, in the large state initialization, the values of entity and architecture parameters of the object, and the values of its deferred constants are available for use (deferred constants are initialized before state variables are initialized). If the architecture inherits another architecture, the large state variables of the inherited architecture are initialized before the large state variables of the inheriting architecture are initialized.

5.4.5 Process

Processes are the leaf elements in the behavior tree defined by a hierarchy of entities. These are threads of computation acting on behalf of the entities that *own* them. The *owner* entity's channels (internal and external) and state variables are accessible by the entity's processes. The basic functionality of processes consists of combinations of two types of actions: *computation* (using the state variables), and *synchronization* (using channels or Time). Computation is some sequence of operations performed on the state variables. Synchronization is some sequence of actions on the channels or Time. Processes can be categorized into two types based on their channel-synchronization method.

- *Arrival-driven* processes are those that wait for activity on a set of channels, and perform a single set of computation actions upon arrival of events on those channels. Such processes are said to *occupy* zero Time.

- *Self-driven* processes are those that contain combinations of one or more computation and synchronization actions. Such processes are said to *occupy* positive Time.

The syntax of **process** definition is as follows:

process **#***N* *entity-name* **:** *arch-name* **:** *process-name* [(*channel-vars*)]
    [**waits** { *wait-var-decls* }]
    [**calls** { *call-var-decls* }]
    [**local** { *ext-var-decls* }]
{
    *wait-statement*s and/or *compute-statement*s
}

---

where, *channel-vars* is comma-separated list of one or more of:
    channel-var [[*ext-expr* **to** *ext-expr*]]
compute-statement is:
    ext-statement
wait-statement is:
    wait *wait-clause* **;**
wait-var-decls are zero or more of:
    var-name1 , *var-name2* , ... **:** *wait-clause* **;**
wait-clause is:
    on *channel-vars* **until** *condition* [**surely**] **for** *expression*
condition and *expression* are:
    *ext-expr*s
and, *call-var-decls* are zero or more of:
    var-name1 , *var-name2* , ... **:** *procedure-name* **;**

Computation actions are specified using action statements in the external language. The external language interface shall provide means to access the parameter, deferred constant, state and large state variables in such computation statements. The external language interface shall also provide means to call procedures and functions of the entity from the computation statements of the process.

Synchronization in Processes

Synchronization actions in a process are specified using the **wait** statement. The **wait on** *channel-vars* causes the process to wait until at least one event arrives on at least one channel in the list of channels given by *channel-vars*. The **wait for** *expression* statement causes the process to wait for the amount of time given by the *expression*. The **wait until** *condition* statement specifies that the process shall remain waiting until such time that the *condition* evaluates to "true". The **for** clause can be used along with the **on** clause to achieve a *timeout* period on channel activity. The **until** clause can be used in conjunction with the **on** clause to achieve a wait for a parametrized instant in time for synchronization.

Waits and Calls

Synchronization can also occur inside the external language computation statements using implementation-dependent constructs that are equivalent in behavior to the external wait statements. The **waits** and **calls** declarations can be used in such synchronization.

The **waits** construct is used to name distinct points inside external language statements where time synchronization occurs. The same wait-clause can be used for more than one wait-point, by declaring (binding) those comma-separated wait variables to the same wait-clause, and using those variables inside the external language code.

Procedure invocations can occur inside the external language statements. Each such invocation can be declared using the **calls** construct. Multiple invocations of the same procedure in the same process (or procedure) can be specified by declaring (binding) multiple comma-separated variable names to the same procedure name, and using those variables inside the external language code.

Notes:

- The **waits** and **calls** declarations can be used only with procedures (see Section 5.4.9) and self-driven processes. It is illegal to use them with arrival-driven processes.
- The **waits** and **calls** declarations should only name the wait-points and call-points that occur immediately in the given process or procedure. Thus, the declarations must not include the transitive (indirect) synchronizations and invocations (for example, those that occur in the procedures invoked by the given process or procedure).

Local Variables

Variables that are used locally in a procedure or process may be declared using the **local** clause. Implementations may enforce other rules on the types of the variables that can be used for local variables. Also, typical implementations may require that those variables that need to retain their value across **wait** and **call** statements must be declared using the **local** clause.

Other

The **surely** tag for the **for** clause can be used for achieving timing semantics in a manner analogous to the **sure** tag for **event**s (see Section 5.2).

When a process is active due to activity on the channels, then all the events at that instant of Time are available at the same time on all the channels that have arrivals on them at that moment of Time. In other words, all simultaneous events are presented on the channels simultaneously; hence, the process must act on all its active channels instead of just one of them.

In a given instance of an entity+architecture, the state variables of the architecture are “shared” by all the architecture’s processes. Read/write conflicts are resolved using the implicit Time instant of access. Simultaneous accesses are serialized using the order of declaration of the processes in the architecture declaration. The rank of a process in this ordering is based on initial point of declaration of the process, and the rank remains unchanged even if the process is overloaded in an inheritance chain. Note that orderings other than the default can be achieved using appropriately designed synchronization via internal channels.

5.4.6 Process Numbering

Any number of processes can be defined for a given architecture. The process number is the key in distinguishing it from other processes within the same entity. Hence, the process name can be treated merely as a mnemonic. Two different processes in the same architecture cannot have the same number; however, process numbers of two processes in two different architectures of the same entity can be the same. Moreover, if one of the architectures inherits from the other, and a process of the inheriting architecture has the same number as the number of a process of the inherited architecture, then the process in the inheriting architecture must be explicity declared to overload the corresponding process of the inherited architecture using the **overload** flag in the process declaration. Such an overloading process in the inheriting architecture completely replaces the overloaded process in the inherited architecture.

If two processes are logically distinct, then they must have distinct process numbers, even across inherited architectures. Thus, process numbers in an inheriting architecture must *not* overlap with the process numbers in the inherited architecture, unless the overlap is intended (for overloading the process, as previously explained).

5.4.7 Component

An entity that is used as a means of defining the behavior of another entity is called a *component*. An entity's architecture can use one or more components. The components in an architecture may be grouped into one or more non-overlapping blocks according to the logical relationships among them. Each such group is called a *component block*. The components in a component block can be interconnected to each other. The components may also be connected to the processes of the enclosing architecture via internal channels.

In other words, the basic behavior of an entity's architecture can be defined in terms of processes and component blocks, where the channels of the entites within each component block are connected to the channels of other entities in the same component block, or to the internal channels of the architecture, while the processes act on the entity's interface channels as well as internal channels.

Any number of component blocks can be defined for a given architecture. The syntax for the declaration of a component block is included in the syntax for architecture. The syntax for the definition of a component block is as follows:

component *entity-type* **:** *arch-type* **:** *component-block-name*
{
  entities { *entity-instantiations* }
  map { *mapping-statement* }
  wrapup { *result-computation* }
}

---

where, *entity-instantiations* is zero or more of:
entity-name => *entity-type* [*param-vals*] **:** *arch-type* [*param-vals*] [[*ext-expr*]] ;
mapping-statement and *result-computation* are *ext-statement*s, and
param-vals is %*n*(*ext-args*)

The **entities** section declares the entity and architecture types of one or more component entities comprising a component block of the architecture. The **map** section defines the mapping pattern among the channels of the enclosing entity and the component entities. The **wrapup** section is used to extract the results of the component entities at the end of the model-exection.

The component block definition may specify a given entity+architecture type for its component entity instances, thus, effectively binding the instance to that entity type and architecture at compile time. Alternatively, the binding can be postponed for later-than-compile-time by specifying a `?` in the place of the *entity-type* or *arch-type*. If `?` is used to postpone the specification of the entity type, then no architecture type must be specified. However, if an entity type is specified, then the binding of architecture type can be postponed by using `?` for *arch-type*. The runtime component of the external language interface shall provide means of binding the entity and/or architecture types in a later-than-compile-time fashion (for example, using a *Configuration Language*).

If an entity type or architecture type is specified, then the integer parameter values required for instantiating such an entity must also be specified. If the entity requires no arguments, then the parameters can be omitted. Otherwise, the arguments must be specified within parentheses, and the count of arguments in the parameter lists must be specified following a `%` preceding the parentheses.

5.4.8 Instance Naming

All entity instances in a given model execution are assigned unique string names. These names are independent of the entity and architecture types of the instances, and the naming scheme is uniform and independent of the instance types. It is assumed that there is exactly one entity that is initially instantiated per model-execution. This entity, called the *root* entity, may be of any entity type and architecture type, and it may specify one or more entities as its components in its architecture, as is the normal capability available in any entity's architecture.

The naming scheme is as follows:

- The name of the root entity is always `root`.
- If a given entity is not the root entity, then it must be a component entity of another (parent) entity. Let the parent entity's instance name be denoted as `PARENT`. Suppose the component block of the architecture of the parent entity names the given component entity instance as `comp` (as specified using the *entity-name* clause in the component block definition — see Section 5.4.7). Then, the name of a component of an entity is given by `PARENT.comp`. If the component is an element in an array of entities (again, as given by the component block definition), then, the component's instance name is given by `PARENT.comp[`n`]`, where n is the index of the element in the array.

5.4.9 Procedure

A Procedure is almost identical in syntax and semantics to a Process. They differ as follows:

- Processes can call procedures, but procedures cannot call processes. Procedures, however, can call procedures and functions, just as processes can.
- Processes are spontaneously triggered upon initialization of the owning entity; procedures are not activated until invoked by some process or procedure of the entity.
- Upon executing the last statement, processes loop back to the first statement, thus running for ever, but procedures return immediately to their caller.
- Processes take channel variables as arguments, but procedures (like functions) take general external variables as arguments. Note that, unlike processes, the presence or absence of arguments does not affect the semantics of procedures (the semantics of processes, on the other hand, is different depending on presence or absence of channel arguments).

The same procedure can be invoked by more than one process at the same time. Procedures are re-entrant, and use the stack of the corresponding calling process for each invocation.

Procedures can take arguments, just like functions (see Section 5.4.10). However, unlike functions, they cannot evaluate to a return value. If the procedure needs to return a result value, it can do so using its arguments.

The syntax of **procedure** definition is as follows:

procedure: exactly same syntax as for **process**, except that *channel-vars* must be replaced with *ext-arg-decls*.

5.4.10 Function

Functions are units of external language code, defined for convenience in model development. The external language shall provide means for invoking functions from the statements of **process**es. Functions are scoped similar to processes and state variables.

The syntax for the declaration of functions is included in the syntax for the architecture declaration. The syntax for the definition of functions is as follows:

function *entity-type* **:** *arch-type* **:** *function-name* [**(** *ext-arg-decls* **)**]
    [**returns** *ext-type*]
{
    ext-statement
}

5.4.11 Result

The syntax for the initialization of result variables is as follows:

```
result entity-type : arch-type : init
{
        ext-statement
}
```

The syntax for using the result variables at the end of model-execution is as follows:

```
result entity-type : arch-type : wrapup
{
        ext-statement
}
```

5.4.12 Temporal Dependencies

The following is the set sequence of actions performed for each entity upon creation, in the specified order of precedence:

1. Parameter values are set.
2. Deferred constants are initialized.
3. State variables are initialized.
4. Large State variables are initialized.
5. Result variables are initialized.
6. Component entities are created.
7. Component channel mappings are performed.

During the model execution, in any given entity, processes are executed according to the timeline of activity. If more than one process is active at the same point on the timeline, then processes are executed in the order in which the processes are declared in their architecture declarations.

At the end of "model execution," the following sequence of actions is performed for each entity, bottom-most entities first to the root entity last.

1. **component-wrapup** is performed for each component block.
2. **result-wrapup** is performed.

The preceding sequence is performed for all component entities before the same is performed for the entity that encloses those component entities.

5.5 File Inclusion

To aid in modular development and reuse of models, compiler directives are provided that permit inclusion of references of other models in the definition of a given model. The units of model development may vary with the target system — such as files in a file-based system, or table names in a database-based system.

Two forms of directives are supported:

- **Model Inclusion Directives**: These are used to reference **TeD** model units in the definition of other **TeD** model units. The syntax for the model inclusion directive is as follows:

use "*model-unitname*"

- **External Code Inclusion Directives**: These are used in **TeD** model units to reference units containing external language code. The external code may be distinguished into two types: *interface* and *implementation*. The semantics associated with each type is dependent on the target system. The syntax for the external code inclusion directive is as follows:

import interface | **implementation** "*ext-code-unitname*"

5.6 Configuration Specification

MetaTeD is only concerned with the description of the models. The way models are assembled, customized, instantiated or used is dependent on the external language interface. The external language interface may provide means for configuration specification in a later-than-compile-time fashion.

6 Interoperability

In the spirit of the **TeD** language, all models must be developed with interoperability in view. Whenever possible, event and channel types must be declared separately from the entity and architecture declarations. Moreover, efforts must be made to reuse a standard set of event and channel types for a given application domain. Use of such standard types will aid in the development of a large library of models, that automatically become interoperable due to the fact that any meaningful interconnection of the instances of different event and architecture types is possible by way of compatible channel types.

References

[1] "A VHDL Primer," J. Bhasker, Prentice Hall, 1995

[2] "A $\mathbf{C}^{++}$ Instance of **TeD**," K. S. Perumalla, R. M. Fujimoto, Technical Report GIT-CC-96-33, College of Computing, Georgia Institute of Technology, 1996.