TeD — A Language for Modeling Telecommunication Networks

Kalyan Perumalla Richard Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{*kalyan,fujimoto*}*@cc.gatech.edu*

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

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. In this document, we present an introduction to the* TeD *language, along with a brief tutorial using an example model of a simple ATM multiplexer.*

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 [3] for the description of a C^{++}-instance of TeD.

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.

The result of the design exercise towards the preceding goals 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. The TeD modeling framework has some analogies with VHDL [1] and similar hardware description languages, which have been very successful in aiding in the construction of complex systems of systems.

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 C^{++} *instance* of TeD is described in [3].

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 C^{++} as the external language, following the interface described in [3], in all the examples. The external language expressions, declarations and other code segments appear between pairs of $ signs, or between `\{` and `\}`.

2 A Tutorial

In the following tutorial, the information on the language constructs and semantics is not necessarily comprehensive, in the interest of clarity of a brief introduction to the language. Detailed documentation with examples can be found in [2] and [3]. Figure 1 illustrates the basic framework underlying the constructs in TeD, and their relation to each other. Figure 2 is an illustration of some of the basic elements of a TeD model and their relationship with respect to the physical object being modeled.

Entity

The 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.

[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).

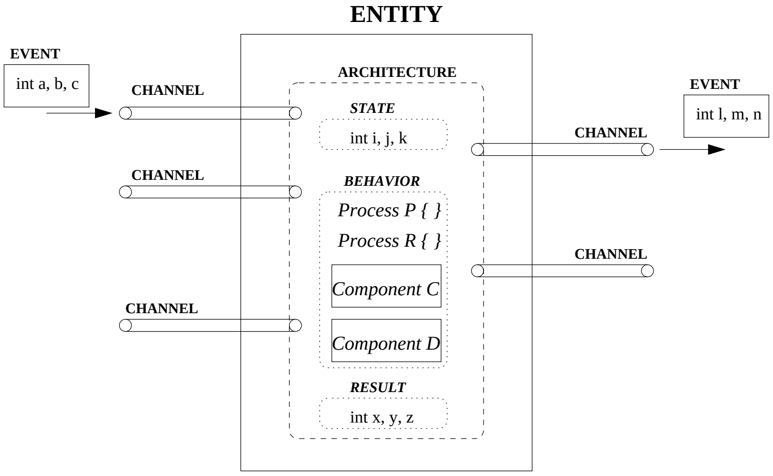

Figure 1: Basic framework of an Entity

An output channel of one entity can be connected to the input channel of another entity. The entity construct for the simple ATM multiplexer shown in figure 2 is as follows:

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

In the preceding 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 later in this section.

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, and it does *not* define any specific behavior of the object. Entities can have zero or more channels (called the interface channels), each of **in**, **out** or **inout** modes. are typed. 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.

An entity, `E2`, can inherit the structure of another entity, `E1`. `E2` will contain all the interface channels of `E1`, in addition to any interface channels defined in `E2`. Entity `E2` may "enhance" its interface by expanding the channel types of some or all of the inherited interface channels. Another type of enhancement is the promotion of the mode of an interface channel from either **in** or **out** to **inout**.

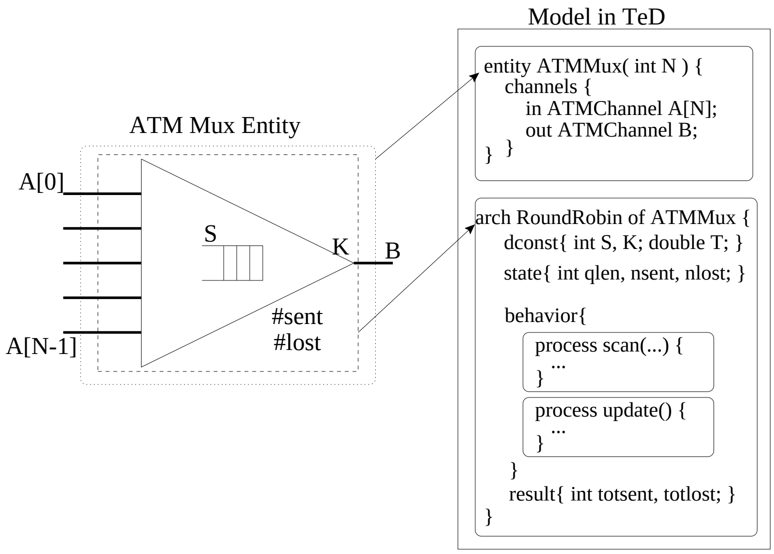

Figure 2: Illustration of a physical entity and its model in TeD

Events and Channels

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. 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.

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

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. 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. Two channels can be mapped to each other only if they are *compatible* with each other.

Architecture

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.

The architecture of an entity can also be expressed as a composition of interacting *components*. Components are nothing but entities themselves "logically enclosed" inside the bigger entity. The entity format of Figure 1 is thus recursive in the components. 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.

While the entity construct describes the entity's external input/output view, the architecture construct describes the internal behavioral part of the object being modeled. More than 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.

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

- **Parameters**: A set of integer variables that are used in defining parametrized strucutures/templates of entities and architectures.
- **Deferred Constants**: 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**: A set of variables that together form a part of the abstract state of the modeled entity.
- **Large State**: 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**: A set of channels that are used for communication among the internal processes and components of an architecture.
- **Processes**: A set of threads of computation that act on the events arriving on the interface and internal channels, or that act upon time advances.
- **Components**: A set of entities that *logically* form subentities of an entity's behavior.
- **Result**: A set of values that are an abstraction of the "result" of "execution" of the model.

The two concepts — *process* and *component* — facilitate 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 delegated 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.

Structural and behavior descriptions of an entity can be inherited by other similar entities, thus allowing object-oriented hierarchy-based design and development. 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 — parameters, deferred constants, state, large state and result variables, processes and component blocks. Inherited properties can be specialized, redefined or overridden by the inheriting entity. Thus, channels are shared across inherited entities; state is shared across inherited architectures; and, processes can be overridden by inheriting architectures.

To illustrate the concept of an architecture, consider the multiplexer shown in figure 2. 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 an input link. Since the current buffer occupancy is the only state information required, a variable `qlen` is used to represent the state. Additional measures, such as the number of cells dropped due to buffer overflow, can also be added to the state. To model the multiplexer's 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 dconst, state and result variables are not defined here in the interest of brevity (consult [3] for complete source code and documentation).

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.

Computation actions are specified using action statements in the external language. Synchronization actions 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.

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. As examples of processes, the `scan` and `update` processes of the simple ATM multiplexer are as follows:

```
process #1 ATMMux : RoundRobin : scan( A[ $0$ to $PARAM(N)-1$ ] )
{\{
    for( int i = 0, n = ASETSZ(A); i < n; i++ )
    {
        if( STATE(qlen) < DCONST(S) ) STATE(qlen)++;
        else                          STATE(nlost)++;
    }
\}}
```

The first process `scan` is executed whenever at least one of the input channels has a data arrival on it.

```
process #2 ATMMux : RoundRobin : update
{
    \{ if( STATE(qlen) > 0 ) {
           CHANNEL(B) << EVENT(ATMCell,());
           STATE(qlen)--;
           STATE(nsent)++;
       }
    \}
    wait for $DCONST(T)/DCONST(K)$;
}
```

The second process `update` is scheduled to be executed every one cell emission time period. This process calculates and updates the state information defining the queue length and the number of cells lost.

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 non-overlapping *component blocks* according to the logical relationships among them. The components in a block can be interconnected among each other, and may also be connected via internal channels to the processes of the enclosing architecture. Any number of component blocks can be defined for a given architecture.

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 using a wild-card specification.

Temporal Dependencies

When an entity is created, its (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, and (7) component channel mappings are performed. During the model execution, in any given entity, processes are executed according to the timeline of activity (any simultaneity is resolved using declaration order).

3 Remarks

The TeD language has been successfully used in the modeling and *parallel* simulation of some large and complex network configurations and protocols, such as ATM Private Network to Network Interface (PNNI) signaling networks [4], multi-cast protocols [5] and wireless networks [6]. The language is being refined and developed further. Problems that are being addressed include the difficulties of coping with two different hierarchies (of TeD and the external language), and the proliferation of macros in the external language to interface with TeD.

References

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

[2] "MetaTeD — A Meta Language for Modeling Telecommunication Networks," K. S. Perumalla, A. T. Ogielski, R. M. Fujimoto, Technical Report GIT-CC-96-32, College of Computing, Georgia Institute of Technology, 1996.

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

[4] "A Virtual PNNI Network Testbed," K. S. Perumalla, M. Andrews, S. Bhatt, Proceedings of the Winter Simulation Conference, December 1997.

[5] "Optimistic Parallel Simulation of Reliable Multicast Protocols," D. Rubenstein, J. Kurose, D. Towsley, Dept. of Computer Science, University of Massachusetts, November 1997.

[6] "WiPPET, A Virtual Testbed for Parallel Simulations of Wireless Networks," J. Panchal, et al, WINLAB, Rutgers Univesity, 1997.