

A Critique of the Telecommunication Description Language (TeD)*

Brian J. Premore, David M. Nicol, Xiaowen Liu
Department of Computer Science
Dartmouth College

Abstract

TeD is an object-oriented description language designed to facilitate the modeling of large scale telecommunication networks, with simulation on parallel and distributed platforms. TeD models are mapped to the Georgia Tech Time Warp engine (GTW) for execution. In this paper we outline the features of TeD, pointing out its strengths and identifying characteristics that gave us trouble as we used TeD to model detailed networks. Our issues are motivated specifically by a model of TCP and a model of multicast resource allocation. Our intention is to illustrate by example what TeD can do, and characteristics that a potential TeD user should be aware of.

1 Introduction

The Telecommunications Description language (TeD) is a system under development at Georgia Tech to provide a simulation framework for large-scale network simulation. TeD is modular, and object-oriented. Its design reflects an overriding goal that TeD submodels be reusable, that they support a library approach to building up large complex systems. Its design reflects a goal for model extensibility as well. It achieves these goals by drawing heavily from VHDL (VSIc Hardware Description Language). It achieves a rich expressiveness by allowing the incorporation of general C or C++ code that manipulates TeD model constructs. TeD models are run in parallel, automatically, by being transformed into GTW (Georgia Tech Time Warp) models.

Programming interfaces for parallel discrete event simulation (GTW [1, 7], U.P.S. [4]) frequently leave low-level details in control of the programmer. While a low-level API can be advantageous in giving the programmer some control over the optimization of a simulation, it can at the same time be burdensome. TeD removes the option of such low-level control by making the description independent of the simulation. The programmer describes the model, and the rest is left to the TeD compiler. This approach has pluses and minuses. The automation is of course, a plus. The negatives

*This work was supported in part by by NSF grants CCR-9308667 and CCR-9625894, and DARPA Contract N66001-96-C-8530.

are that the goal of automation has seemingly forced the TeD designers into imposing restrictions on TeD models that impede model development and force us into unnatural and inefficient modes of expression. TeD also removes from programmer access to some critical performance-sensitive decisions.

We are part of a team that is funded to develop and use TeD on large-scale real-life simulation network modeling problems. In some sense we are representative of the intended user group for TeD: we are experienced modelers, but unexperienced in Time Warp simulation. While this paper is critical of aspects of the TeD design, it should be noted that TeD is a hugely ambitious project that has been under development for only a year. We see our role in the project as being prototypical users, providing feedback on TeD's suitability for its intended task. This paper should be viewed in that vein.

Section 2 briefly overviews modeling in TeD. Section 3 investigates and reports on aspects of TeD which characterize its behavior. Section 4 reviews the performance of a particular simulation modeled in TeD. Section 5 summarizes and concludes this examination of TeD.

2 A brief overview of modeling in TeD

A model in TeD consists of *entities* which communicate using *events* and *channels*. Each entity object represents some actual entity in the modeled network, and events are simply messages encapsulating information. A channel represents some means by which actual entities can send and receive events. Every channel is associated with a particular entity, and two entities can communicate if they have associated channels which have been *mapped* to each other. To illustrate, we shall introduce an example which simulates the TCP network protocol. Top level entities in this model are such things as `ROUTERS` and `SOURCE_NODES` (non-router nodes which generate and send information). Both types of entities may have any number of channels, and these channels could be mapped to define the user's network topology of choice. The object-oriented structure of TeD also allows for entities to contain components which are themselves entities. For example, `SOURCE_NODE` entities might have an array of `PORT` subentities. In fact, in every TeD model, there exists a root entity of which all others are subentities. `ROUTERS` and `SOURCE_NODES` are subentities of a root entity named `SYSTEM`.

For each entity, at least one *architecture* must be defined which specifies that entity's behavioral model. The architecture defines the *state* of the entity (variables to be used during the simulation), the *behavior* (which includes how it will respond to events that it receives), and the *result* (variables which describe the result of the simulation for that entity). In the TCP example, an architecture for a `ROUTER` might contain a routing table (state), a process which forwards datagrams when they are received (behavior), and variables to keep track of the number of datagrams forwarded (result).

3 Characteristic aspects of TeD

3.1 Features

TeD provides many ways for programmers to precisely define the high-level aspects of their models. We have found the following to be general and useful.

3.1.1 External code blocks and macros

There are many contexts in TeD in which it is necessary to define detailed model behavior. An *external code block* is a stand-alone segment of the external language (C++, in this case) that allows such details to be programmed. TeD defines several macros for use inside external code blocks which allow access to model data and perform model-related functions, such as sending an

event. As an example, in defining the behavior of a `TCP_AGENT`—a subentity of a `PORT`—there are many different scenarios which must be dealt with (handling and making connect requests, sending and acknowledging data, monitoring and adjusting the transfer rate of data, handling and making close requests, and many more). Much data manipulation and the use of nested conditional statements are necessary to properly define its behavior so that it can keep track of and handle these situations. External code blocks provide a convenient means for this to be accomplished.

Overall, there are few restrictions on what is allowed in an external code block. However, some of them are significant, and are detailed in the sections to come.

3.1.2 The configuration language

While entities and their behavior can be described in detail in TeD, it can still be difficult to customize different instances of entities without resorting to extremely awkward techniques (e.g., passing in many parameters). In order to make such customization easier, the TeD system provides a *configuration language* in addition to the standard TeD language. The configuration language provides for such customization, in addition to allowing the programmer to specify the bindings of entities to architectures, all of which can be done in a very straightforward manner. Everything which is specified in the special configuration files can be changed to update the model without having to recompile.

Suppose that in the TCP model, we are implementing `SOURCE_NODES` as an array of subentities inside the `SYSTEM` root entity, and we wish to give out specific IP addresses to each one. Passing in the addresses as integer values, one for each `SOURCE_NODE`, would be non-trivial for two reasons. First, the values would have to be passed in as parameters to the `SYSTEM` entity, and all parameters are required to be integers (we would prefer to pass an array of integers). Second, the number of `SOURCE_NODES` itself might be variable. While configuring under these constraints could still be done, TeD's configuration language saves us the trouble. We can put the values in an array, as desired, and use a simple loop to assign the appropriate address to each `SOURCE_NODE`. Furthermore, if we wish to change just one address, we can edit the configuration file and avoid recompilation.

3.2 Restrictions

While TeD provides the programmer with much functionality and freedom, it also imposes some restrictions. The ones described below are those that caused us the greatest difficulty.

3.2.1 The wait statement

Like VHDL, TeD allows model behavior to be expressed in terms of processes that operate on model state. Typical of process-oriented simulation, the code expressing process behavior includes **wait** statements expressing that the process should suspend for a certain length of time, until some condition is met, or until it is prodded back into life by some external event. A TeD *process* is composed only of external code blocks (see Section 3.1.1) and special *wait statements*; it looks much like the canonical function or procedure. In fact, except for the **wait** statements they behave almost identically the same as C++ functions. Processes may be *event-driven*, or *arrival-driven*. If event-driven, a process is only executed when an event arrives on a specified channel or channels. If *self-driven* it executes continually throughout the simulation, restarting each time it finishes.

The **wait** statements used in processes control synchronization and timing of the model. A **wait** statement has three basic forms. It can **wait for** a given length of simulated time to pass, **wait on** a given channel for an event to arrive, or **wait until** a given condition becomes true. Certain combined forms are also allowed, in which case waiting ceases when any one of the statement's constituent parts is satisfied. (For example, **wait on *ch* for *t*** would cease waiting when either an event arrived on channel *ch*, or *t* time units passed, whichever occurred first.) Despite the added

```

begin APPLICATION behavioral process
begin external code block
  int x = a random integer;
  int n = network address of a randomly chosen SOURCE_NODE;
  int p = a randomly chosen PORT number;
  switch (state_of_execution)
  {
  case connected:
    if (x%2 == 0) // a 1/2 chance of attempting a disconnect
    {
      send TCP_AGENT msg requesting the connection be closed;
      wait for response msg from TCP_AGENT;
      if (response msg indicates a successful connection close)
        state_of_execution = unconnected;
    }
    else
      wait for my_wait_time time units;
    break;
  case unconnected:
    if (x%3 == 0) // a 1/3 chance of attempting a connect
    {
      send TCP_AGENT a msg requesting a connection with
        the APPLICATION on PORT p of SOURCE_NODE n;
      wait for response msg from TCP_AGENT;
      if (response msg indicates connection was established)
        state_of_execution = connected;
    }
    else
      wait for my_wait_time time units;
    break;
  default:
    error;
  }
end external code block
end process

```

(a)

```

This is badcode.tex, the one before this was code.tex.

begin APPLICATION behavioral process
begin external code block
  int x = a random integer;
  int n = network address of a randomly chosen SOURCE_NODE;
  int p = a randomly chosen PORT number;
  boolean sent_dummy_msg = FALSE;
  switch (state_of_execution)
  {
  case connected:
    if (x%2 == 0) // a 1/2 chance of attempting a disconnect
      send TCP_AGENT msg requesting the connection be closed;
    else
      send TCP_AGENT a dummy msg; // DUMMY MESSAGE
      sent_dummy_msg = TRUE;
    }
    break;
  case unconnected:
    if (x%3 == 0) // a 1/3 chance of attempting a connect
      send TCP_AGENT a msg requesting a connection with
        the APPLICATION on PORT p of SOURCE_NODE n;
    else
      send TCP_AGENT a dummy msg; // DUMMY MESSAGE
      sent_dummy_msg = TRUE;
    }
    break;
  default:
    error;
  }
end external code block
wait for response msg from TCP_AGENT; // UNEMBEDDED WAIT
begin external code block
  if (!sent_dummy_msg)
  {
  switch (state_of_execution)
  {
  case connected:
    if (response msg indicates a successful connection close)
      state_of_execution = unconnected;
    break;
  case unconnected:
    if (response msg indicates connection was established)
      state_of_execution = connected;
    break;
  default:
    error;
  }
  }
end external code block
end process

```

(b)

Figure 1: The effect of restricting the use of the **wait** statement. In (a), the code is written in a very natural way, but uses **wait** statements illegally by embedding them in an external code block. The code in (b) is legal, but notice that without using embedded **wait** statements, we must use *two* external code blocks. Not only that, but wasteful dummy messages must be sent to make the code work as desired.

functionality from these more complex forms of **wait** statements (which were not yet implemented at the time of this writing), there are still some behaviors which cannot adequately be described in TeD. It is for this reason that we can focus on an example which uses only the first two types of **waits** (**wait on** and **wait for**) and still make valid conclusions about TeD. In fact, the problem in TeD is not that the **wait** statements themselves are not powerful enough, but that there is no mechanism to embed those **wait** statements inside an external code block. The repercussion of this restriction is that certain simple behaviors of a process cannot possibly be described in TeD. To illustrate these consequences, let us look at a concrete instance that arose in our TCP model.

Recall that for each SOURCE_NODE entity, there is an array of PORT entities. Each PORT entity in turn contains an APPLICATION entity and a TCP_AGENT entity. An APPLICATION communicates with its associated TCP_AGENT, which does the communicating with the “outside world,” talking across the network to TCP_AGENTS on other SOURCE_NODES. We now focus on the communication between an APPLICATION and a TCP_AGENT, which are connected by channels.

Since an APPLICATION should be the controlling party (the TCP_AGENT exists to handle its requests), it initiates all communication between the two. (This method also avoids the complication of simultaneous messages between them.) The behavior of an APPLICATION is such that at any moment, it is in one of several predefined states, which we will refer to as *states of execution*;¹ for this example we shall simplify the situation so that there are only two such states—*connected* (a communication link is open with another APPLICATION somewhere on the network) and *unconnected*. We define an APPLICATION to behave according to its state of execution, some arbitrarily chosen probabilities, and a wait time which helps control how “quickly” it executes. Throughout any simulation run, an APPLICATION’s behavior is defined by a looping self-driven process. (The behavior of the APPLICATION is said to be *process-driven*.) Each time through the loop, the APPLICATION follows well-defined steps:

- 1 Check the current state of execution and skip to the code associated with that state.
- 2 Probabilistically determine what action will be taken (if any), and whether or not that action requires that a request message be sent to the TCP_AGENT.
- 3 Execute the appropriate choice of the following two, depending upon whether or not a request needs to be sent to the TCP_AGENT:

Request Necessary: Send the request to the TCP_AGENT and wait for a response. When the response is received, make any necessary changes in the state of execution and other state variables.

Request Unnecessary: Wait for *my_wait_time* units of (simulated) time.

The most straightforward way to code the previous sequence in C++ is with a **switch** statement (switching on the different possible states of execution). Then within each case, parts 2 and 3 of the list are executed using conditional, assignment, and **wait** statements. The end of the **switch** statement would also be the end of the loop. The code might look something like that of Figure 1 (a).

Though TeD macros for sending and interpreting messages can appear in an external code block, **wait** statements cannot (see Figure 1). This is unfortunate, as that functionality is required to code the behavior of an APPLICATION in a natural way. Furthermore, legal alternatives with no embedded **waits** can become large and unwieldy, and in some cases, no alternative exists at all! Specifically, *there is no way to code a choice between the two standard kinds of wait statements*.² Coping with this obstacle leads to very unnatural and complicated code, as was the case with the APPLICATION code. We chose to wait on a channel, since we could not use both types of **waits** in the desired manner. For the times when we did not want to wait for an event, we had to send a dummy message to the TCP_AGENT requesting a dummy response simply to get past that line of code. Ugly as it is, this was cleanest workaround which we found (see Figure 1 (b)).

Three unwanted characteristics that result from the **wait** statement restriction are as follows:

- 1 **The desired model behavior may not be possible.** If the desired behavior of a process requires using different types of waiting from one execution to the next (e.g., it sometimes waits on a channel, sometimes waits for t time units), there is no way that the model can be coded within TeD.
- 2 **Modeling becomes more cumbersome.** When using any language, one would like to produce code which expresses the properties and behaviors of the resulting executable as clearly, concisely, and naturally as possible. Not being able to do so complicates, and in turn slows, the coding process.
- 3 **Model performance is slowed.** As Figure 1 illustrates, lengthier code and even wasteful communication between entities may result. Entities can

¹This terminology is used simply to avoid confusion with other similar terms. One of the other defining characteristics of an entity is its *state*, which is simply a collection of *state variables* (see Section 2). In fact, the state of execution itself is one of an APPLICATION’s many state variables.

²The “two standard kinds” of wait statements we refer to here are the **wait for** and **wait on** forms. Other forms seem as if they might help achieve the desired behavior, but on careful inspection, that is not in fact the case.

end up sending messages constantly—effectively ping-ponging other entities—as a side-effect to achieving (or just approximating) the desired overall behavior.

While it is clear that forbidding **wait** statements from being embedded in external code blocks is restrictive to the modeler, it is important to understand why TeD makes this restriction. In GTW, all logical processes are event-driven. It is a feature of TeD that the modeler is allowed to define the behavior of an entity as being process-driven, and it is the translation from the event-driven to the process-driven type of behavior that is at the root of the **wait** statement restriction. Each process in a TeD model is represented internally as a C++ class of which one instance is used during a simulation. That class contains a vector of “process items,” which are defined as sections of the process code, with **wait** statements serving as the boundaries between them. For example, a process with just one **wait** statement is divided into two process items—the code before the **wait**, and the code after the **wait**. Generalizing, a process with n **wait** statements has $n + 1$ process items. This design supports a very simple solution to the problem of re-animating suspended code. When a TeD process is suspended at a **wait** statement, no context information (e.g., stack) is stored, only the index of the re-entry point, which is stored as part of the C++ class describing processes. To return control to a process following the last **wait** statement it executed, the TeD run-time system needs only to look up the index and jump to the “next” segment of code.

This is a simple solution for the TeD designers, but the restriction that process code must be straight-line (with respect to wait statements) greatly restricts us as modelers. Real process-oriented simulators suspend and restore processes by saving and restoring contexts, and we believe TeD might be able to do the same. The Standard C library contains functions **setjump** and **longjmp** precisely for saving stack context and returning to it. With some re-design of TeD, these calls could provide a way of greatly generalizing our ability to express TeD process behavior.

3.2.2 Complex data structures

While there are relatively few restrictions on coding in external code blocks, restrictions that do exist can impose severe limits. For example, no memory can be allocated or deallocated (except in a few limited initialization contexts). This in turn makes it practically impossible to use many kinds of common complex data structures, such as linked lists and trees. Although structures such as classes and structs are not restricted anywhere, they do lose some of their usefulness as a result of the memory allocation/deallocation restriction. Dynamic memory allocation in optimistic parallel simulation was a problem solved by the Time Warp Operating System project [3] (TWOS), the solutions developed there ought to be provided in TeD and GTW.

One context in which the use of pointers seems simple, however, is in the external variable declarations of an event/message. Allowing pointers in this context would be very useful to the TCP model. We model a TCP datagram as an event, and the specification for a datagram calls for optional headers which can be added to provide additional information. Without pointers, such optional parts are difficult to include in an event description without setting hard limits and wasting memory (e.g., a fixed-size array).

One important characteristic of an event variable which makes it different from a state variable is that its value never changes. Furthermore, events rarely exist for more than a small fraction of the total simulation time. An event is created and sent by one process, and received and interpreted by another. After the receiver is done reading it, it is not needed. Since event variable values never change, there is no reason to checkpoint them. In a shared memory scheme (such as GTW), it would be reasonable for them to reside in the same memory location throughout their existence, thus making the use of pointers within them safe. In a distributed memory scheme, a slightly more complicated (though equally effective) system would be necessary. The burden would lie on the programmer, who would need to build appropriate methods for transferring event variables intact. These methods would deal with packing a dynamic structure—a tree, for example—in such a way

that if the packed version were sent to another memory bank, a similarly defined unpacking method would be able to restore its structure correctly. The methods required would be such things as a copy constructor, a transfer constructor, and a destructor.

3.2.3 Global variables and scoping

The current TeD design prohibits the use of global variables. This is reasonable in that any values which are not constant throughout the simulation should be considered part of the simulation state. For global variables which are read-only, however, such a constraint is unnecessary.

We introduce now another example—simulating a multicast on an internet—to illustrate the usefulness of read-only global variables. To simulate multicast sessions, we’ve designed a model composed of N NODE entities and M LINK entities which can be interconnected to form any desired network topology. As a simulation executes, NODEs attempt, at random times, to set up a multicast session. In order to determine the feasibility of setting up such a session, the NODE needs bandwidth information from each involved LINK. Before getting such data, the NODE must determine which LINKs those are. Once this information is acquired, a NODE can calculate whether or not the given session can be admitted. In order to carry out these tasks, each NODE needs to know the topology of the entire network (unlike the TCP example, where only local connections had to be known). For a realistic simulation, the size of the network could be very large. Keeping copies of the entire topology of a network of this size at each NODE would be highly inefficient.

The network topology is defined during the setup phase of the simulation, and does not change thereafter. Clearly, keeping only one global copy of it would be ideal. Because of the hierarchical design of TeD, changing the scoping rules to allow subentities read access to the state variables of their parent entities would solve the problem. Thus, we have defined a global variable as any state variable in the root entity of a model. To simplify the multicast example, then, the root entity would keep the network topology in its state, having defined it using the configuration language (see Section 3.1.2) in the setup stage of the simulation. Since every other entity is somewhere below the root in the hierarchy, each NODE would have the desired access to the topology.

There are, however, important consequences to be considered when extending the scope of variables.³ Currently, TeD is built on the GTW engine for shared memory multiprocessors. To implement global variables of this type on a distributed memory platform, each processor could have its own instance of the variable. However, this is reminiscent of the original problem of multiple copies that we had hoped to solve. A more space-efficient solution would be to implement a software cache for global variables, at each processor. A cache miss would trigger a fetch from the processor owning the global variable.

3.2.4 Mapping channels

In a TeD model, each channel can be mapped to at most one other—a constraint which seems unnecessary and has proven to be burdensome. Although TeD does allow a process to wait on any number of channels simultaneously, extracting the active set (the set of channels with pending events) may not be efficient. In the multicast model, for example, each LINK must maintain a large number of channels, each of which comes from a different NODE. It must listen to all of these channels simultaneously by looping, which impacts performance negatively when the number of channels is large. Furthermore, management of all these channels is tedious, when it is obvious that a many-to-one channel mapping would ensure that all incoming events arrived on the same channel. Likewise, if an event were sent out on a one-to-many channel, multiple copies of the event could automatically be generated to go to the multiple recipients.

³Calling them ‘variables’ here is somewhat of a misnomer. The only globals we are interested in are those which are set once at start-up and remain constant from that point on. TeD allows entities to have “deferred constants,” which fits our definition exactly, except that they are not global.

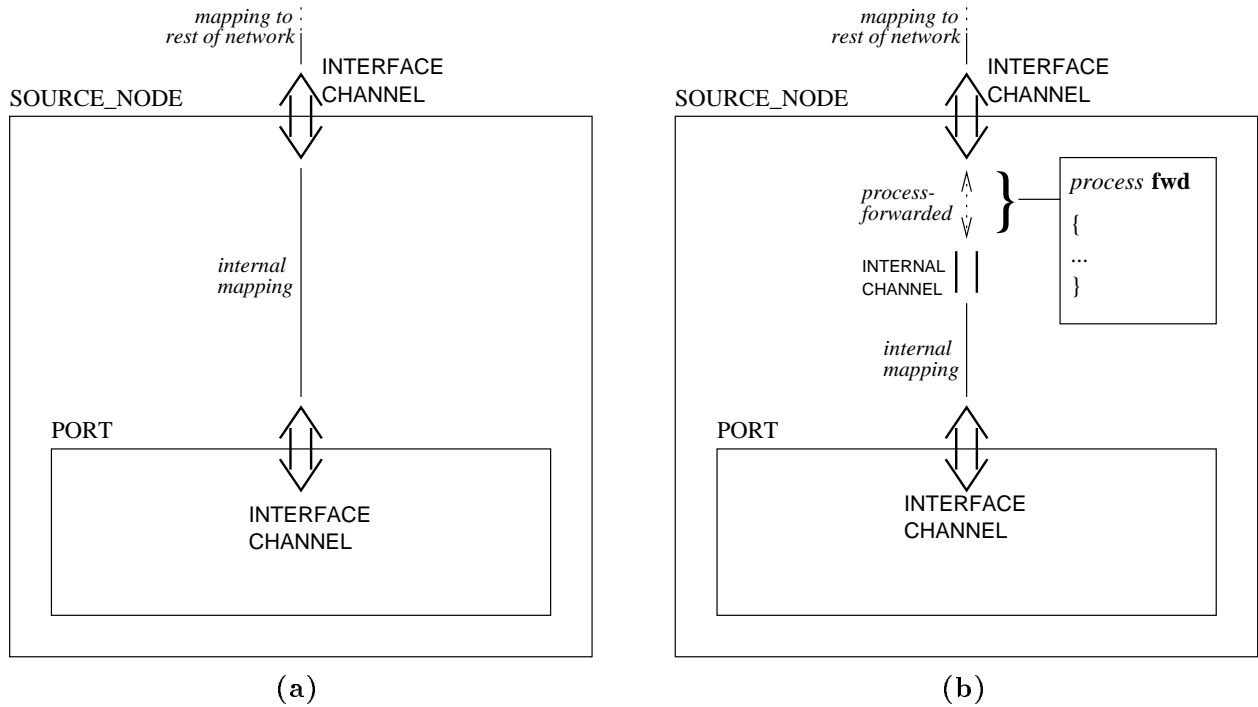


Figure 2: The difficulty with internal mapping. (a) How one would expect to be able to implement an internal mapping. (b) How it must be done in TeD.

A second type of channel called an *internal channel* differs only slightly from the *interface channels* that have been discussed so far. While interface channels are used for inter-entity communication, internal channels are used for intra-architecture communication, and are not bound to an entity, per se. For example, in describing the behavior of a SOURCE_NODE in the TCP model, we need to specify a method for forwarding datagrams internally to the proper PORT subentity. To simplify matters, let us consider a SOURCE_NODE with only one PORT, so that forwarding should be trivial. What we would like to do is map the interface channel of the SOURCE_NODE to the interface channel of its PORT, as in Figure 2 (a). However, such a mapping cannot be done in TeD. In order to achieve the desired effect, we must create an internal channel in the SOURCE_NODE, map it to the PORT’s channel, and create an event-driven process in the SOURCE_NODE which simply forwards events between the SOURCE_NODE’s internal and interface channels (see Figure 2 (b)). This is overly complicated for such a simple operation. The problem is that mapping directly would result in the SOURCE_NODE’s interface channel being mapped more than once—which is not legal in TeD—since it is also mapped to the rest of the network external to the SOURCE_NODE.

To make matters even worse, TeD provides no way to refer to “the current entity” in code (compare with the C++ **this** pointer). Every time we needed to implement an internal mapping, we had to write our own macro specific to the particular environment.

3.2.5 Mapping processes to processors

The performance of any parallel or distributed simulation is closely related to how the logical processes in the simulation are mapped to the physical processors. TeD does all mapping transparently, doing little more than doling out the logical processes cyclically to the processors. While this removes the burden of mapping from the modeler, it also prevents the modeler from being able to implement different, possibly better mappings (see Section 4 for some unsettling performance statistics). Since finding the optimal mapping for any given model is non-trivial, the modeler

should have some discretion over it. To make such low-level details configurable as part of the language would be to taint the high-level interface that TeD strives to provide. However, it would be reasonable to allow the user to edit some kind of optional mapping setup file to be layered over the TeD language, much the same way as the configuration language is (see Section 3.1.2).

4 Performance

We ran our implementation of the TCP model in TeD using a network composed of 10 ROUTERS and 70 SOURCE_NODES. The ROUTERS were connected in a ring, and each one had seven SOURCE_NODES connected to it. Each SOURCE_NODE contained four PORTS, and each PORT contained one APPLICATION and one TCP_AGENT. Along with the root entity (SYSTEM), that made for a grand total of 911 entities. From the unconnected state, an APPLICATION requests connections at a rate of 5 per simulation time unit. They chose which SOURCE_NODE and PORT to connect uniformly at random. When connected, data packets are sent at a rate of 1.5 per unit simulation time; the mean connection duration is 20 units of simulation time. For each link in the network (a channel mapping) that a TCP packet event crossed, there was a random delay between 1.0 and 1.3 time units, inclusive. A delay of 0.0 (no delay) was used for events passed between an APPLICATION and its associated TCP_AGENT.

In TeD, each entity is represented as one logical process, so there were also 911 of those. The model was run on an SGI Onyx with four processors. The average event rate for one processor was 6670 events/second, whereas for four processors it was 35279 events/second. This provides us with an acceleration of nearly 5.3. Indeed, increasingly inappropriate accelerations were also observed for two and three processors. Such behavior has been explained in the past as being due to use of non-scalable data structures, e.g. a linearly linked event-list with linear searching for insertion. Our model code contains no such constructs; it appears probable that GTW or the TeD run-time system do. We are investigating this further.

The model whose performance is give above is perfectly homogeneous. TeD's strategy of cyclically assigning LPs to processors is bound to be effective. It is of some interest to observe what happens when the simulation workload is not so homogeneous. We modified the TCP model so that for every SOURCE_NODE, there is one APPLICATION whose connection rate is so slow that it virtually never connects. Because of the problem with the **wait** statement described earlier, what actually happens is that the workload associated with this application actually *increases* over applications that achieve connections, due to the incessant polling of the TCP_AGENT. After studying how TeD assigns workload to processes, we place these modified APPLICATIONS in such a way to be all assigned to the same processor in a four-processor system. Serial execution of this modified system yielded an event rate of 7190 events/second. Simulation by four processors yielded highly variant results; most runs produced on the order of 4800 events/second, but occasionally a faster run—e.g., 10700 events/second—was achieved. While artificially constructed, the point is made that TeD's load-mapping strategy is insufficient for non-homogeneous models. The modified model is actually homogeneous when viewed from a slightly higher level of abstraction. Given suitable control we could easily cause workload to be perfectly balanced, and could exploit known communication affinities (e.g., between SOURCE_NODES and TCP_AGENT). In fact, we dove into GTW internals to force a balanced mapping of this particular problem and were rewarded with an average event rate of 33600 events/second. We feel strongly that TeD ought to provide one with an ability to control or influence the workload mapping.

5 Conclusions

TeD is a high-level modeling language which provides a good abstraction from the performance details of simulation. Its object-oriented nature and separation of structure from behavior make

it a modular system, its components reusable and interchangeable. Though TeD is a step toward Fujimoto's "Holy Grail" [2] of a completely transparent interface, our experience is that it currently makes some implementation decisions that negatively impact our ability to model, and that in the special case of load management it takes away control only to handle the problem naively while leaving the modeler helpless.

TeD does provide the modeler with considerable high-level functionality, but it also has some weaknesses and restrictions which make describing certain characteristics of a model difficult. Some of these restrictions are due to fundamental difficulties in parallel/distributed simulation. Limited use of memory allocation/deallocation and pointers are an example of such. Others are not as deeply-rooted, such as forbidding global read-only variables and one-to-many or many-to-one channel mappings. Perhaps the most notable and costly limitation in TeD is its failure to provide complete process-driven functionality. We understand that providing such functionality is non-trivial, given that TeD is built on a system which is completely event-driven (GTW). Having full process-driven functionality, however, is essential for building natural, high-level models, as well as for further promoting reusability. As so, we are anxious to see this matter investigated more carefully.

TeD has the potential to be a building block toward ideal high-level modeling with optional low-level control. We hope that our experience will influence the TeD designers to reconsider some of their design decisions.

References

- [1] S. Das, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp system for Shared Memory Multiprocessors. In *1994 Winter Simulation Conference Proceedings*, pp. 1332–1339, December 1994.
- [2] R. Fujimoto. Parallel Discrete-Event Simulation: Will the Field Survive?. In *ORSA Journal on Computing*, 5(3):213–230, 1993.
- [3] D. R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLorenzo, P. Hontalas, P. Reiher, K. Sturdevant, J. Tupman, J. Wedel, and H. Younger. The Time Warp Operating System. *11th Symposium on Operating Systems Principles* 21(5):77-93, November 1987.
- [4] D. Nicol and P. Heidelberger. On Extending Parallelism to Serial Simulations. *Proceedings of the 1995 Workshop on Parallel and Distributed Simulation*, pp. 60–67, June, 1995.
- [5] K. Perumalla, R. Fujimoto, A. Ogielski. MetaTeD—A Meta Language for Modeling Telecommunication Networks. College of Computing, Georgia Institute of Technology, and Bell Communications Research, 1996.
- [6] K. Perumalla, R. Fujimoto. A C++ Instance of TeD. College of Computing, Georgia Institute of Technology, 1996.
- [7] K. Perumalla, R. Fujimoto. GTW++—An Object-oriented Interface in C++ to the Georgia Tech Time Warp System. GIT-CC-96-09, College of Computing, Georgia Institute of Technology, 1996.