Rapid Prototyping for an Assertional Specification Language

Jorge Cuéllar[*†]
Dieter Barnard[*]
Martin Huber[*]

ABSTRACT
Temporal Language of Transitions (TLT) is a framework for the specification and verification of reactive control systems. *Message Passing Interface* (MPI) is a library for message-passing in a distributed environment. In this paper we present a compiler from TLT to the language C and the library MPI. The development of the compiler raised a number of interesting issues like the implementation of deadlock-free synchronous communication and the scheduling of guarded-commands.

1 Introduction

Today there exists a plethora of formal methods for the specification and verification of distributed and embedded systems. However, in many cases they still lack the recognition as a serious software development technique in industry. One important reason is the perception of a large gap between a correct specification on the one hand and its implementation on the other. This deficiency can be addressed by *rapid prototyping*, which tries to deliver an executable system based on a given specification as early as possible. We present such a tool in the form of a compiler, which inputs (verified) TLT specifications and facilitates their distributed execution on a network of workstations using the MPI library.

2 Temporal Language of Transitions (TLT)

TLT is a compositional framework for the formal treatment of reactive control systems [CH94, BC95, CBH95]. A TLT specification consists of a

[*]Siemens AG, ZFE T SE 1, D-81730 Munich, Germany.
[†]E-Mail: Jorge.Cuellar@zfe.siemens.de, Tel. (089) 636-47585, Fax. (089) 636-42282.

number of modules. Each module consists of a number of interfaces and a module body. The interfaces and the body each contain a set of declarations of variables and actions, an initially predicate and a set of always-commands. In addition, the body also contains a set of guarded-commands.

Actions are used to express synchronous message-passing. They are typed, and declared as *input* or *output*, depending on whether they are under control of the environment or the given module. A module reacts to input actions using always-commands (thereby making it input-enabled). A module executes output actions using guarded-commands in the usual way.

3 A Compiler for TLT

3.1 The Message-Passing Interface (MPI)

MPI is a library standard for message-passing [MPI94], with support for the language C and SUN/SPARC workstations. We will concentrate here on only a small part of MPI necessary for our compiler: initialization, point-to-point communication, and termination. In MPI the code of all the modules are packed together and then distributed. During initialization, each processor node is assigned an *id* between 0 and $n-1$, from which it determines the appropriate part of the code to execute. A variety of asynchronous and synchronous send/receive primitives is provided (discussed below). A MPI application terminates if all participating nodes decide to do so.

3.2 Synchronous Input and Output

A central issue in the development of the compiler is the realization of synchronous communication. Even though MPI explicitly provides a blocking synchronous send primitive suitable for TLT output actions, it is easy to see that two processes wishing to synchronize simultaneously can deadlock. We therefore make use of the nonblocking synchronous send primitive, which includes the possibility of checking whether a started synchronization has completed. If not, care is taken not to initiate a new synchronization on the same channel (by temporarily disabling the appropriate instructions). TLT input actions are treated dually: the blocking receive primitive is used, but only after checking that there indeed exists a matching output action wishing to synchronize.

3.3 Scheduling

Consider the scheduling algorithm for each module, summarized in Figure 1. Every TLT variable is implemented as a unprimed and a primed C variable. The unprimed versions are assigned initial values where specified. At the the start of the first *execution phase*, their values are copied

to the primed versions. This implements the frame axiom for TLT, namely that variables which are not assigned new values retain their current values. Next the scheduler checks for the presence of any input actions; if another module wishes to communicate, the appropriate always command is executed.

After processing the inputs, a nondeterministic selection is made of any enabled instruction (if there exists one). Note that in addition to the explicit guard of an instruction, enabledness is also determined by the ability to perform all of the output actions in the command part (i.e., all previous synchronizations on those particular channels must have been completed). Finally, the primed values are copied back to the unprimed variables before the start of a new execution phase. The execution is repeated ad infinitum. If no inputs are present and no instructions are enabled, the module just stutters.

```
Assign initial values to (unprimed) variables.
Assign unprimed values to primed variables.
while (true) do {
    Perform always commands triggered by inputs.
    Select and execute an enabled instruction.
    Assign primed values to unprimed variables.
}
```

FIGURE 1. The Scheduler for a TLT Module

This scheduling algorithm renders a true implementation of the TLT semantics for a large class of TLT modules (that respect commutativity conditions similar to [Mis91]). In particular, it is not possible for consistently composed TLT modules to deadlock, since output actions are performed in a nonblocking manner and since all inputs are processed at the start of each phase. This covers the case where two modules wish to synchronize simultaneously, and where one module wants to synchronize twice but the other module has not reacted to the first yet. In this case the second synchronization is disabled temporarily until the first has completed.

3.4 Execution and Logging

Compiled TLT systems can be executed on SUN/SPARC workstations connected by a LAN, with one TLT module executing on each computer. Each module contains a scheduler as explained above. There are two ways in which a user can visualize the execution of a systems. Firstly, provision has been made for ASCII-based input (keyboard) and output (screen). If only one of the modules performs I/O, then the compiler starts this module on the default console. If more than one module wishes to perform I/O, then the compiler adds an I/O manager, i.e. a separate MPI process dedicated handling all I/O requests.

The second possibility is to make use of the MPE (Message Passing Environment) library, which offers extensive logging facilities. If activated, each MPI process creates a local log-file. After (proper) termination, the log-files are collected and integrated, and displayed by the MPE tool `upshot`. For this purpose, there is a special keyword HALT in TLT, which can be used in instructions to terminate execution for logging puposes. An example is provided in Figure 2, where we compiled and executed a small master/slave system. The top bar represents the master module, the lower two represent the slaves, and the arrows represent MPI communication.

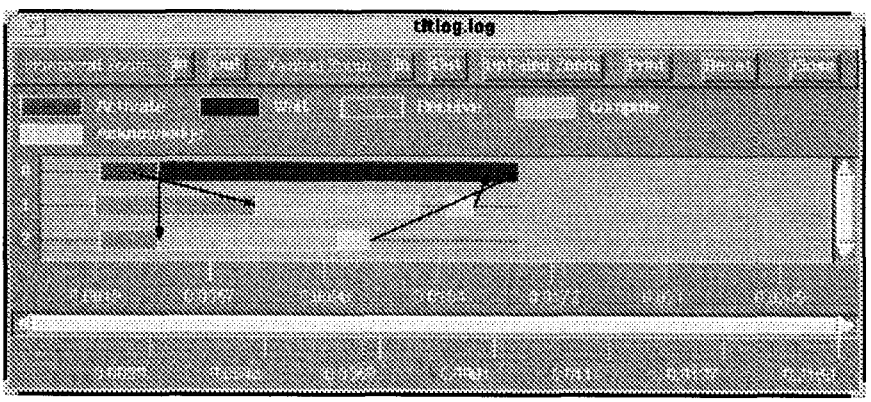

FIGURE 2. MPI/MPE Logging of Master/Slave

4 Conclusion

We have presented a compiler which allows rapid prototyping of TLT specifications. Our goal was to narrow the gap between formal specifications and their implementation. We maintain that the compiler preserves the original (formal) semantics of TLT, even though the synchronous input and output actions of TLT have been replaced by MPI communication. Such prototypes not only serve the role of simulations, but can significantly aid in the understanding of a specification, by visualizing their distributed execution and communication. In addition to numerous small examples, the compiler was used to create a real-time, distributed controller of the FZI production cell simulation [CH94].

Acknowledgements: Christine Roeckl and Dagmar Proell contributed significantly towards the implemenation of the TLT compiler.

5 REFERENCES

[BC95] Dieter Barnard and Simon Crosby. The Specification and Verification of an ATM Signalling Protocol. In *Proc. of 15th IFIP PSTV'95*, Warsaw, June 1995.

[CBH95] Jorge Cuéllar, Dieter Barnard, and Martin Huber. A Solution relying on the Model Checking of Boolean Transition Systems. Submitted as Final Solution to Dagstuhl Seminar of Broy/Lamport, 1994, Siemens Corporate Research and Development, ZFE T SE 1, D-81730 Munich, Germany, 1995.

[CH94] Jorge Cuéllar and Martin Huber. The FZI Production Cell Case Study: A distributed solution using TLT. In *Proc. of the FZI*, volume 891 of *LNCS*. Springer-Verlag, 1994.

[Mis91] Jayadev Misra. Loosely-Coupled Processes. In Springer Verlag, editor, *PARLE'91, Vol. 2*, pages 1–26, 1991. LNCS 506.

[MPI94] MPIF (MPI Forum). MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, May 1994.