AMEDS-Tool: An Automatic Tool to Model and Simulate Large Scale Systems

Veronica Gil-Costa and Jair Lobos
Universidad Nacional de San Luis, Argentina.
{gvcosta,jlobos}@unsl.edu.ar

Roberto Solar and Mauricio Marin
Yahoo! Research Latin America, Santiago, Chile.
{rsolar,mmarin}@yahoo-inc.com

Keywords: XML specification, Automatic code generator, Discrete event simulation.

Abstract
Simulating the cost of applications running on large clusters of processors poses difficulties in model definition and simulation. In this paper we propose a methodology to ease this burden. The user specifies a model describing it as a timed coloured Petri Net in a graphical manner by using a tool like CPN tool. The model is automatically converted into a XML specification. Then a code generator converts the XML file into C++ code which is linked to a simulation kernel. The result is an efficient and scalable simulation program that can be executed sequentially or in parallel on either single multi-core processors or cluster of multi-core processors. We illustrate the suitability of our proposal by modelling and simulating the cost of message passing in a Fat-tree network which is commonly used to support communication in cluster of processors.

1. INTRODUCTION
The ability to manage large and complex sets of data processed with applications run on clusters of processors, also requires the ability to deploy and coordinate large amounts of hardware resources. Every day a new technique or application is introduced that pushes the edges of the speed-size envelope even further. However, evaluating the effectiveness and efficiency of these new techniques or applications has a high economical cost, especially if they do not succeed.

Modeling and simulation provide an approach to solve this problem. In particular, in this work we propose an automatic modeling tool tailored to perform parallel discrete-event simulation as described in [9]. Model definition is realized by defining Timed Colored Petri Nets (TCPN) elements and modules. Each CPN-element has one single transition and one or more places as defined in the Petri Net scheme. Each module has one or more CPN-elements that communicate to each other through I/O ports. Modules also communicate to each other through I/O ports. Therefore, we propose a layered approach to integrate model components into a parallel simulation kernel. We have developed an Extensible Markup Language (XML) schema definition to specify CPN elements and modules that defines a component syntax tailored for sequential and parallel simulation purposes. The XML document is automatically translated into C++ [6] code to be linked to the simulation kernel. We evaluate the accuracy and performance of our tool by modeling and testing different communication benchmarks on a Fat-Tree network[1].

This paper is organized as follows: In Section 2 we review Timed Colored Petri Nets concepts and the parallel simulation approach. Section 3 presents our proposed automatic tool to model and evaluate discrete systems. Section 4 shows the experimental results. Conclusions follow in Section 5.

2. BACKGROUND
XML has become a standard to represent data exchanged among applications and metadata which allows categorizing data and information through the use of tags. These tags can be used to represent elements of a particular programming language like data structures, programming classes, etc.. XML documents can be parsed to generate source code.

There is a large number of works presented in the literature which use XML documents to generate source code [3]. Most of these works focus on the parsing phase efficiency. The work in [2] presents an scheme which uses XML to generate C++ code by means of a generalized automaton. The work presented in [7] and extended in [8] presents a prototype to parse XML documents and a validation scheme to prove the robustness of XML-based communication. The Elvior XML-Simulator 1 is used to design and evaluate web services, P2P applications, distributed embedded applications. XML have also been used to customize system configurations and to pass parameters as in the stream computing platform S4 [11].

In our context, the XML format can be used to save or load user-defined models. With the CPN tool 2 users can define the system model by means of a graphical interface which is saved in XML format.

Our approach is similar to [5] and [12] in the sense of exploiting the flexibility of XML. Both approaches suggest to transform XML representations into platform specific models. The first one focuses on high level architecture compliance. The last one focuses on interface definitions close to UML for the parallel DEVS formalism. The approach we propose in this paper puts a strong emphasis on the interaction of basic CPN-elements, triggering conditions and inter-

1http://www.elvior.com/xml-simulator/general
2http://cpntools.org/
face definitions according to the CPN approach to modelling. More precisely, XML forms the basis for representing interfaces, CPN-elements and modules. Data binding is used to execute transformations between XML documents and object instances, which are used by the simulation kernel.

2.1. Timed Colored Petri Nets (TCPN)

A TCPN [14, 10] is a high-level Petri net formalism extending standard Petri Nets to provide basic primitives to model concurrency, communication and synchronization. The notation of TCPNs introduces the notion of token types, namely tokens differentiated by colors, which may be arbitrary data values. This aims to practical use because they enable the construction of compact and parameterized models. TCPNs are bipartite directed graphs comprising places, timed-transitions and arcs, with inscriptions and types allowing tokens to be distinguishable. Places are represented by ovals and transitions are rectangles. Places have a type and may have an initial set of tokens (initial marking). Arcs may contain tokens expressions types/values used to indicate the condition by which one or more tokens may move for one transition to one place, and vice-versa. There can also be expressions signaling tokens movement from several places (transitions) to several transitions (places). An expression is evaluated to true when all input places contain the required tokens. An un-timed transition is fired right after its firing condition is enabled, but a timed-transition waits for the pre-defined time delay to be fired after enabled. This allows us to cause cost ($\Omega$) of different operations in the simulation time.

Figure 1.(a) shows a simple TCPN model. The model is composed of a set of threads represented by the place named $T$. This place is initialized with five tokens ($5'$[$]). Each one of these tokens can solve a different request concurrently. The place named Input receives requests and sends them to place $B$. Transition function is triggered when there is at least one token inside the place $T$ and inside the place $B$. When transition function is triggered, simulation time is advanced according to $@+\Omega$. After that, a token is placed inside the place $T$ and another token is placed inside the place Out.

Modeling with TCPN provides some advantages like a diagrammatic tool to model concurrency and synchronization in distributed systems, it is used as a visual communication aid to model the system behavior and it is based on strong mathematical foundation. Then, modeling with TCPN provides a graphical view of the system behavior and a better undressing of how elements flow though different parts of the system.

2.2. Parallel Simulation

A discrete event simulation consists of a set of time-stamped events, which must be executed in ascending chronological order. On a single processor, these events can be placed in a central queue so that the global event list algorithm can correctly order them. However, simulation of large-scale system requires long execution times, it can easily take weeks to simulate networks with thousands of nodes. When run in parallel, not only is the event list distributed so that each processor has a portion of it, but events may also arrive asynchronously from other processors.

Parallel simulation models provide an improvement in terms of execution times. Under this paradigm the simulation model is partitioned into different physical processors that communicate exchanging messages (events). The main goal of traditional parallel discrete simulation (PDES) models is to provide the exact same results as with the sequential simulation models. The parallel simulation approach presented in [9] is more relaxed than traditional PDES approaches since it prioritizes fast execution of the simulation rather than having accurate results. No rollback mechanisms are used to correct the chronological order of event execution.

There are several models of parallel computing, one of them is the Bulk Synchronous Parallel (BSP) computing model [13]. Under the BSP model, computation is organized as a sequence of supersteps. During a superstep, processors may perform computations on local data and/or send messages to other processors. At the end of a superstep there is always a synchronization barrier. It permits that messages sent during the current superstep are available for processing at their destinations at the next superstep. The underlying communication library ensures that all messages will be available at their destinations before starting the next superstep.

As described in [9], the implementation of the parallel simulator on $P$ processors, uses multi-threading and a bulk-synchronous message passing strategy to automatically conduct simulation time advance. In each processor there are $T$ threads and one of them is the master thread in charge of synchronizing with all other $P$-1 master threads to execute the BSP supersteps and exchange messages. In each processor and superstep the remaining $T$-1 threads synchronize with the master thread to start the next superstep, though they may immediately exchange messages during the current superstep as they share the same processor main memory.

3. THE AMEDS MODELING TOOL

In this section we describe our automatic modeling tool based on a parallel simulation algorithm previously presented in [9]. To achieve efficient and scalable performance, the parallel algorithm relaxes strict causality of events in the simulation time and thereby produces approximate simulation metric results. It also supports exact parallel simulation at the cost of reduced performance.

Our proposal aims to facilitate the development of complex systems modeled by Petri Nets. To this end, the developer has to describe his/her model using a simple set of XML tags grouped into CPN-elements and modules. The XML CPN-
elements contains attributes used to characterize the TCPN elements. Namely, the places, transitions and connecting arcs.

Figure 2 shows an example of a simple XML code for the TCPN model presented in Figure 1.(b). Notice that Figure 1.(b) is based on the basic elements of a TCPN as described in Figure 1.(a). The principal module is called \textit{root} which contains all the complex modules defined by the developer and tokens definitions. Inside the \textit{root} component we can define colored tokens by using the \textit{token} tag. In this example, we define two colors named \textit{Message} and \textit{thread}. The Message token has an attribute named \textit{Id \_Msg} initialized with the value 0. The thread token has an attribute named \textit{TID} initialized with the value 1.

The \textit{module} tag is used to define modules with one or more CPN-elements. Each module has an attribute named \textit{starter} used to identify the component of the TCPN which begins the simulation execution. It also has an \textit{input source} tag which defines the input of the module. In this example, module \textit{M0} is the \textit{starter} and it has no input source. Inside the \textit{module} tag we define CPN-elements containing the place, transition, trigger, delay, and select tags. In this example, we define a place \textit{M} inside \textit{E0} with an initial mark \textit{msg}. If the trigger condition is evaluated to true, the tasks defined inside the \textit{select} tag are executed. In this case, we get the \textit{msg} token from place \textit{M} and we add user specified \texttt{C++} code to increase the value of the attribute \textit{Id \_Msg}. Afterwards, we consume the token from place \textit{M}. Inside each TCPN module we can define the output arcs for each transition. Arcs are used to connect CPN-elements and modules. We define \textit{arc0} to connect the output of transition \textit{T0} to the place \textit{A} of module \textit{M1}. We also define \textit{arc1} to connect the output of transition \textit{T0} to the place \textit{M} of the module \textit{M0}.

More complex CPN-elements can be defined as shown in Figure 1.(b) on the right. Conditional arcs can be included into the XML code using the \textit{condition \_group} tag. In this example, the \texttt{cpn->WhoAmI(id)} function receives the token message identifier and determines which CPN-element \textit{Ei} for \textit{i} = 1…\textit{N} is going to process the token. Moreover, a set of CPN-elements performing the same tasks but with different data can be defined using the \textit{array} type. For lack of space, we do not include the rest of the code for the \textit{Ei} CPN-elements. Besides, the XML code for these elements can be obtained using tags already described.

### 3.1. Transformation to Simulation Model

Once we have written the simulation model into a XML file based in our TCPN XML tags, we execute the \texttt{C++} source code generator (XML2C++) tool. This tool allows translating TCPN users models written in XML tags to \texttt{C++} classes in a transparent fashion. The XML2C++ tool is a simple program written in \texttt{Python} that uses two very powerful and flexible \texttt{Python} modules: \texttt{ElementTree} - a simple and efficient API for parsing and creating XML data and \texttt{Cheetah} - a template engine and code generation tool.

The XML2C++ tool functionality consists of the two following stages:

1. \textit{Parsing the XML file by using ElementTree XML API}. This first stage consists on parsing the XML input file, translating and organizing the TCPN XML tags in such a way that \texttt{Cheetah} understands how to process the data. This component operates in a three-level hierarchical fashion:
   - \textit{first level} (ModuleFinder), searching for modules. Modules contain \textit{attributes}, an \textit{input} link, an \textit{output} link and a set of \textit{cpnelements}. CPN-elements have a sophisticated representation and are parsed in the second level.

\footnotetext[3]{http://www.python.org/}  
\footnotetext[4]{http://www.cheetahtemplate.org/}
Figure 2. XML code for a simple TCPN model. Two modules, the first one with one CPN-element. The second module has N+1 CPN-elements and uses conditional arcs to connect transition $T_i$ with the places inside the CPN-elements $E_i$ for $i = 1...N$.

Figure 3. Proposed TCPN scheme simulation based on XML, Python and C++ classes.
• second level (CPNElementFinder), searching for CPN-elements. CPN-elements contain a set of places, a set of arcs and a transition. A transition has a more sophisticated representation and therefore is parsed in the third level.
• third level (TransitionExplorer), parse the transition for the corresponding CPN-element.

Finally, the ConnectionGenerator validates the arcs tags and generates the connections.

Note that we make a processing difference between atomic and nested tags. Atomic tags are simple structures and are parsed by reading its inline description (attributes, input or output links, places, arcs, etc.). Nested tags are complex structures and are parsed in a recursive way by means of a specific component (ModuleFinder, CPNElementFinder, etc.).

2. Generating the C++ Source Code by using Cheetah Template Engine. A very important issue in this stage is the design of well-structured templates. Templates are skeleton classes used to generate the user level layer at the parallel simulator side (see Figure 3). This second stage consists of receiving the data generated by the previous stage and filling up a set of designed templates.

4. USE CASE: MODELLING COMMUNICATION INFRASTRUCTURE

We now illustrate the building of a simulation model for a concrete example. Assume we want to build a model of a network consisting of a number of nodes, each one connected to a component that represents the transport layer of the network. In particular, Figure 4 shows the CPN model for a Fat-tree network [1].

We focus on modeling the costs of communication among processors through a Fat-tree [1] network topology, that interconnects communication switches and cluster processors. This network is commonly used in data centers. Processors are grouped in racks. The communication network is organized in three levels. At the bottom, processors are connected to what is called Edges switches. At the top of the tree are the so-called Core switches and in the middle there are Aggregation switches.

To model and simulate a Fat-tree network, we empirically estimate the cost of sending messages through each switch. We use this cost to set values for transition delays within the CPN-elements and modules. To this end, we run benchmark programs on an actual switch to determine the average communication cost of point to point messages. The Fat-tree rules for building a model are the following. For k port switches, there are k groups called Pods. Each Pod contains k/2 Edges switches and k/2 Aggregation switches. Half of the port of Edge switches are connected to processors and the other half ports are connected to Aggregation switches. Half of the Aggregation switches ports are connected to Edge switches, and the other half are connected to Core switches. Ports of Aggregation switches are connected to ports of Aggregation switches. Fat-tree can host k^2/4 processors and there are k^2/2 Edge and Aggregation switches and k^2/4 Core switches.

Then the communication cost depends mainly on the number of hops that the message has to go through. In general, we have three values c1, c2 and c3 which represent the costs of sending a message to a processor in the same rack, to a processor in an adjacent rack and to a processor in another Pod, respectively, with c1 < c2 < c3.

In Figure 4 each Pod has four processors. The processor with IP 10.0.0.2 is connected to Edge with IP10.0.0.1. The processors with IP 10.0.1.2 and 10.0.1.3 are connected to the Edge with IP 10.0.1.1. Edge switches are connected to Aggregation switches. Aggregation switches are connected to both Edges switches and Core switches.

5. EXPERIMENTS

Results were obtained on a cluster with 16 CPUs of 64 bits with Intel Q9550 Quad Core 2.83GHz processors and RAM memory of 4GB DDR3 1333Mz. All experiments were run with all data in main memory. We used the C++, MPI, BSPon-MPI and OpenMP libraries to implement our programs.

5.1. AMEDS-Tool Accuracy

To validate our proposed TCPN-based simulation tool, we run a set of benchmark programs. These benchmarks are also used to properly tune the simulator cost parameters, which is critical to the comparative evaluation. We obtained benchmarks for unicast, broadcast and all-to-all communication operations from the NetPIPE® protocol. Time was measured in microseconds ($\mu$s) using the gettimeofday() function. Unicast implies sending a message to a single processor. This benchmark works with an even number of processors. The idea is to measure the communication between two processors without interference from others. Process pairs are selected through the processes identifier (pid). Then, a processor with odd pid value communicates with the processor pid – 1, and a processor with even pid value communicates with the processor pid + 1. Pid allocation is performed internally by MPI. Communication is implemented through the MPI primitives MPI_Send() and MPI_Recv. Both are blocking.

The broadcast benchmark program measures the transmission of a message that will be received by every device on the network. This benchmark works with any number of processors. The MPI_Bcast primitive is used to perform the MPI broadcast of a message of given size. All processors involved

3http://www.scl.ameslab.gov/Projects/NetPIPE/
in the operation, must execute this command. Only one processor (called root) sends a message to everyone else. The operation is completed when all processor have received the message. The time of this operation is determined as the maximum time of all processors to complete this operation.

Finally, the all-to-all benchmark program works with any number of processors. We implemented one benchmark program using the MPI primitive \texttt{MPI_Alltoall()}. All processors simultaneously send a message to all other processors. The idea is similar to the previous benchmark for measuring time. Namely, the communication time is given by the maximum time required to complete this operation on all processors.

As a validation experiment, Figure 5 shows the latency reported by the benchmark programs and our TCPN-based tool for all three communication patterns. These experiments were performed by using messages of different size as shown in the $x-$axis. The parallel simulation was executed with $P = 16$ processors. Results show that our tool is capable of almost overlapping each point reported by the benchmark programs in the curves of Figure 5.(a), Figure 5.(b) and Figure 5.(c). In other words, results show that the executed simulations are capable of estimating the latency achieved by different communication patterns over a Fat-tree network.

Finally we calculated the root mean square error of the deviation which is a measure of the differences between values obtained by the real benchmark programs and the values reported by the simulations. It is defined as $\epsilon_m = \sqrt{\sum (x_i - \bar{x})^2 / (n \cdot (n - 1))}$ and we calculated the relative error ($\epsilon_r$) as $\epsilon_m / \bar{x}$. In Table 1 we show the relative errors achieved by the TCPN-based simulator tool. In general, the error values reported in the table for all the communication benchmarks are very small.

Table 1. Relative error for benchmarks.

<table>
<thead>
<tr>
<th>Communication pattern</th>
<th>$\epsilon_m$</th>
<th>$\epsilon_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast</td>
<td>0.0005%</td>
<td>0.47%</td>
</tr>
<tr>
<td>All-to-all</td>
<td>0.3%</td>
<td>1.02%</td>
</tr>
<tr>
<td>Unicast</td>
<td>0.002%</td>
<td>0.82%</td>
</tr>
</tbody>
</table>

5.2. Performance Evaluation

In the following experiments we evaluate the performance achieved by our AMEDS-Tool. In particular, we evaluate the running time, the speed-up (\textit{Sequential Time/Parallel Time}) and the memory usage in bytes when running the AMEDS-Tool with different number of processors $P = \{2, 4, 8, 16\}$.

Figure 6 shows running time normalized to 1 in order to better illustrate the comparative performance. To this end, we divide all quantities by the observed maximum in each case. The $x$-axis show the message size. We tested the unicast, all-to-all and broadcast operations to evaluate the Fat-tree network model with 64 processors and 8 Pods. In this experiment we measure the simulation time required to send 60,000 messages. For the all-to-all and unicast communication patterns the AMEDS-Tool with $P = 16$ reduces running times by 25% in average. For the broadcast operation the simulation time is dramatically reduced up to 90% as we increase the number of processors. With this particular communication pattern the message size has a higher impact on the running time achieved by the parallel simulation of the Fat-tree network.

Figure 7.(a) shows the speed-up achieved by our tool when testing different communication patterns. As in previous figure we simulate a Fat-tree network with 64 processors and 8 Pods. We report a speed-up close to the optimal.
results show that our tool efficiently scale.

On the other hand, Figure 7.(b) shows the amount of bytes used by a sequential simulator, our AMEDS-Tool and a Time-Warp (TW) simulation [4]. In this experiment we test the Broadcast operation over the Fat-tree network. Results show that our tool can dramatically reduce the amount of memory required by traditional parallel simulation strategies. Of course, the amount of memory used by the sequential algorithm cannot be improved, but with a minimum increase of about 20%, our proposed tool can improve the performance of sequential algorithm.

6. CONCLUSION

We presented the AMEDS-Tool based on Timed Colored Petri Nets for modelling large-scale and complex systems. In particular, we presented the Fat-tree network as a case of
study. All Petri Nets elements can be combined to produce fairly complex components. The logic must be specified by the model developer. These specifications include conditions by which transitions must be triggered and the selection of alternative paths for tokens. The result is a flexible framework that enables developers to define very large models providing interfaces to define and evaluate different components in a test environment.

ACKNOWLEDGMENT
This work has been funded by Project CH/12/04 MINCYT-CONICyT and CONICET, Argentina.

REFERENCES


