WORLD-WIDE TOKEN FLOW USING
OBJECT PETRI NETS BASED ON TCL

JÜRGEN NÜTZEL; WOLFGANG FENGLER
nuetzel;wfengler@theoinf.tu-ilmenau.de
http://www.theoinf.tu-ilmenau.de/ra1/
Faculty for Informatics and Automation
Technical University of Ilmenau
D-98684 Ilmenau, Germany

Abstract
In this paper we present our approach for distributed computing and simulation. We will discuss a special high-level Petri net simulator which is coupled to a script interpreter. The paper will explain the net elements and the textual syntax to define and run high-level Petri nets. Our Petri net interpreter is called OPNTcl and comes as an extension package for Tcl8.0. The paper describes additionally how to split an OPNTcl net across several machines connected to the Internet. Finally an example will be shown how tokens can deliver data as well as Tcl code which can be started on the target machine. Such code fragments can also include command which may extend the Petri net structure dynamically.

Keywords: Simulation, Petri nets, Script Language

1 INTRODUCTION

Petri nets are widely accepted for modelling and simulation of distributed systems. Petri nets with uniform tokens are not sufficient in many applications. Often tokens with data values are needed. If the Petri net includes guard functions and software actions which are using token's data, a programming language will be helpful. In our approach we coupled a high-level Petri net simulation engine with the Tcl (Tool command language) interpreter from Ousterhout [1], [2]. Our engine comes as an extension package for Tcl version 8.0 [3]. Tcl is available for all Unix, Windows and Macintosh platforms. Tcl in combination with Tk (Tool kit) allows to write graphical applications (using the Tk part) which are portable across all platforms from above.

Our Object Petri net simulation package named OPNTcl extends the Tcl language by a set of specific script commands like Transition, PlaceOrQueuePlace etc. These commands are a subset of those which may be used to define Petri net structures. Further commands are provided for tokens, data class definition and net execution. A special hierarchy concepts is supported that allows to use sub nets like sub-routines.

If a post-arc definitions addresses a machine name the tokens may flow across the Internet to the addressed place located on the remote machine. There is not problem to couple machines of different CPU type because the data are always strings. Having a real interpreter language - Java is not an interpreter language - allows us to generate and send program code with the tokens without any any demand for a compiler.
2 THE OPNTCL PACKAGE

Designing a Petri Net is like writing a Tcl script. Building up the net structure means to evaluate the script in the Tcl interpreter. Evaluating the special Sim command starts the net simulation. Every script has to start with the following line

```
package require opntcl
```

This command line tells Tcl interpreter to load the OPNTcl package if needed. After that line the net definition follows. Some simple examples will explain the net elements and its potentials. Starting with an example with uses no high-level elements.

```
Place p1 10; # the place p1 with 10 uniform tokens
Place p2 0 3; # the place p2 with the capacity 3
Transition t1 {
    # any Tcl code which will be executed while the transition fires
}
Pre p1 t1 1; # pre arc from place p1 to transition t1
# Test and Inhibit define testing pre arcs
Post t1 p2 1; # post arc from transition t1 to place p2
```

Writing Sim 10 at the end of the script will start the small simulation. The Tcl code which is assigned to transition t1 runs only 3 times. After the third step the simulation command returns with the number seven which tells the script that seven steps are left. If the transition code includes any syntax errors the simulation command will abort with an error code. You can use the Tcl catch command to prevent that abort. A Tk button can be used to trigger single simulation steps:

```
button .b -text Sim -command {
    if {[catch {Sim 1} result]} {
        puts "Error: $result"
    } else {
        if ($result != 0) { puts "Net Dead" }
    }
}
pack .b ; # shows the button
```

To build greater models with repeating structures special commands are provided by the package. Every Petri net defining command which is placed between BeginNet and EndNet belongs to a subnet with its own namespace. In combination with the proc command which defines a subroutine and special im- and export commands from the package parametric net macros are possible.

```
proc ChainNet ( name in out (length 1) ) {
    BeginNet $name
    ImportPlace p0 $in ; # $in under p0 in subnet visible
    for {set i 1} {($i <= $length)} {incr i} {
        Transition t$i
        Place p$i
        Pre p[expr $i - 1] t$i 1
        Post t$i p$i 1
    }
    EndNet $name
}
```
ExportPlace $out p$i ;# p$i under $out in higher level
EndNet
}

Place Start 10
ChainNet n1 Start End1 100
ChainNet n2 End1 End2 100

The example above defines a net structure with 200 places and transitions. All net elements form one chain which starts with place Start and ends with place End2.

Besides these low level Petri nets high-level nets with data values attached to the tokens are possible. For definition of the data structures the package provides the DataClass command. Although Tcl stores every data values in their string representations the package defines five predefined base data classes: string, int, char, double and boolean. The data class command allows either to define new base data classes or new complex data classes. Tokens of specific data classes accept only strings which represent a value which is conform to token’s data class.

```tcl
# some sample base data classes:
DataClass colours string (enum (red green blue))
DataClass tclcode string
DataClass byte int (rangeint {0 to 255})
DataClass negativeDouble double (rangedouble {to 0})
DataClass even int (basename {
    if (!($value % 2)) (return 1)
    return 0
})
# some sample complex data classes:
DataClass remoteplace ((string place) (string machine) (int port))
DataClass message ((result code) (tclcode code) remoteplace)
```

For a token definition the package command Token is used. A token can be defined outside a place or directly inside the place definition. The OPNTcl provides three high-level place commands for three different scheduling strategies. QueuePlace stands for FIFO strategy, StackPlace for LIFO ordering and RamPlace for random access. A place may hold tokens of different data classes. A high-level arc is only defined for one data class. A variable at the arc is used to access tokens passing through.

```tcl
set tol [Token message {{sender s1} {code {puts Hallo}}}]  
queuePlace buffer "$tol [Token int 50]"  
QueuePlace out;# empty place  
Transition t1 { puts "Token value: [Get a]"}  
# t2 executes the code:  
Transition t2 { eval [Get b code] }  
Pre buffer t1 1 int a ;# passes int tokens to t1  
Pre buffer t2 1 tclcode b ;# passes tclcode to t2  
# double the value of the token from place buffer:  
Post t1 out 1 int c { Set c [expr [Get a] * 2] }  
```

At pre arcs additional boolean filter function may assigned to select specific tokens from the pre place. A RamPlace is the only place type which allows the arc to search in the whole place. In QueuePlaces and StackPlaces only the end of the queue is accessible. To combine values from several pre places the filter function can also be assigned to the transition. In this paper we left out to explain the time stamps which the tokens gets if they enter the place. The filter function may use this time stamps to model specific delay functions.
3 DISTRIBUTED PETRI NETS

A special extension to the Post command allows to send tokens to a place which is located in an OPNTcl net running on a remote machine. The Tcl interpreter supports easy socket communication. This feature allows to distribute a net simulation across the whole Internet. Only places from the top level are accessible from remote machines.

```tcl
post tl ( pl ossi.theoinf.tu-ilmenau.de 100 ) 1 int
```

In this case the post-arc does not test the capacity of `pl` on the remote machine. To define a net which is additionally able to receive tokens from remote further commands have to be executed.

```tcl
# Init the socket communication across a specific port
StartMessageServer 100; # defines the port number 100
SimForEver; # Start the simulation loop
```

The command `StopMessageServer` will close the communication connection.

4 EXAMPLE

To manage the work flow within an distributed company Petri net models are suitable. A OPNTcl model combines data flow and the flow of code which may view and modify the data.

```tcl
replace incoming
transition work ( catch {
  set work result [eval [get work code]]; # execute code
  set m [get work machine]; set po [get work port]; set pl [get work place]
  # create net structure which sends tokens back to send:
  transition backto$m
  pre results backto$m 1 message work [concat {[get work machine] == }] $m
  post backto$m "$pl $m $po" 1 message work; # the remote place
  } catchresult )
pre incoming work 1 message work
replace results
post work results 1 message
```

REFERENCES