C++-PaT-Net

JVP is controlled by PaT-Nets (Parallel Transition Networks). They are simultaneously executing finite state automata. Every clock tick, they call for action and conditionally make state transitions.

For the development of JVP, we implemented PaT-Nets in C++. Each class of PaT-Nets is defined as a derived class of class `LWNet`, which stands for Light Weight PaT-Net. Its nodes are defined with their associated actions and transition rules in its constructor, while actions and conditions are defined as member functions in each PaT-Net. Other class-specific initialization and termination can be performed in constructors and destructors of each class, respectively[1]. Local variables can be defined as member variables in class definitions of PaT-Nets.

Other features of the C++-PaT-Nets include:

(1) The definition of PaT-Nets is extended so that it can have multiple states at the same time. It enables us to represent simple parallel execution of actions in a single PaT-Net.

(2) A PaT-Net can send messages to other PaT-Nets. (message passing), and can also wait for their reply.

Instances of PaT-Nets are stored on a list, and they are scanned every tick.

[1]Common initialization and termination processings as a PaT-Net are performed in the constructor and destructor of class `LWNet`, respectively.

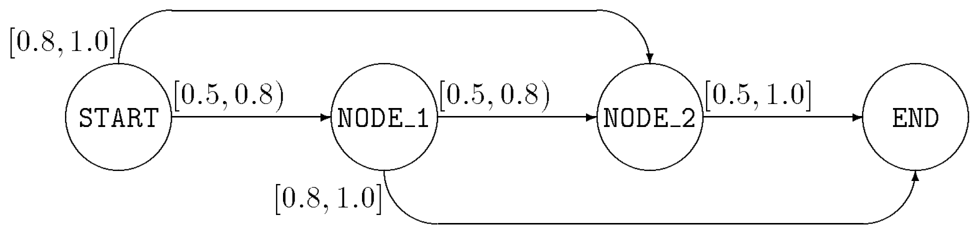

Figure 1: A sample PaT-Net.

1 Defining PaT-Nets

Let us define a sample PaT-Net where a random value in $[0, 1]$ is generated every tick and then conditionally makes transitions depending on the value. Its structure is shown in Figure 1.

Figure 2 is a sample header file defining the sample PaT-Net as `SmplNet`. `SmplNet` is defined as a derived class of class `LWNet`, and thus `lwnet.h` needs to be included in the header file.

Nodes in a single C++-PaT-Net are identified by numbers from 0 to $n-1$ if n is the number of the nodes. For readability, however, node identifiers should usually be defined by an enumeration as shown from lines 5 to 12. Local variables, action functions, conditions functions, constructors, and destructors are also declared in the class definition. Arguments and return values for action/condition functions are defined in `lwnet.h` as:

```
typedef void (LWNet::*ACTFUNC)(void);   // action function
typedef Bool (LWNet::*CONDFUNC)(void);  // condition function
```

A program file for `SmplNet` is shown in Figure 3 In the figure, action/condition functions and constructor/destructor are defined.

At the top of the constructor, arguments for `LWNet` must be provided. In the declaration in `lwnet.h`, the constructor of `LWNet` has two arguments:

```
LWNet(int sz, NodeId start = 0);
```

The first argument is the number of nodes, and the second is the start node id. The second argument can be omitted, and the default start node id is 0.

In the constructor, the structure of the net is defined. For example,

```
#include "lwnet.h"

class SmplNet : public LWNet {
private:
// node identifier (in literal)
    enum {
        NODE_START,
        NODE_1,
        NODE_2,
        NODE_END,
        NUM_NODES   // the number of nodes
    };
// local variables
    double rgv; // randomly generated value in [0,1]
// action functions
    void actfunc_s(void);
    void actfunc_1(void);
    void preactfunc_1(void);
    void postactfunc_1(void);
    void actfunc_2(void);
// condition functions
    Bool condfunc_5(void);
    Bool condfunc_8(void);
public:
    SmplNet();  // constructor
    ~SmplNet(); // destructor
};
```

Figure 2: A header file for `SmplNet`.

```
#include <stream.h>
#include <stdlib.h>
#include <sys/time.h>
#include "smplnet.h"

void SmplNet::actfunc_s(void)
{
    rgv = drand48();
    cout << "Get " << rgv << " in actfunc_s" << endl;
}

void SmplNet::actfunc_1(void)
{
    rgv = drand48();
    cout << "Get " << rgv << " in actfunc_1" << endl;
}

void SmplNet::preactfunc_1(void)
{
    cout << "Executing preactfunc_1" << endl;
}

void SmplNet::postactfunc_1(void)
{
    cout << "Executing postactfunc_1" << endl;
}

void SmplNet::actfunc_2(void)
{
    rgv = drand48();
    cout << "Get " << rgv << " in actfunc_2" << endl;
}

Bool SmplNet::condfunc_5(void)
{
    return (rgv >= 0.5) ? TRUE : FALSE;
}

Bool SmplNet::condfunc_8(void)
{
    return (rgv >= 0.8) ? TRUE : FALSE;
}
```

Figure 3: A program file for `SmplNet`.

```
SmplNet::SmplNet()
: LWNet(NUM_NODES)
{
    cout << "Executing SmplNet constructor" << endl;

// defining PaT-Net structure
    defnormalnode(NODE_START, (ACTFUNC)&SmplNet::actfunc_s);
    deftrans(NODE_START, (CONDFUNC)&SmplNet::condfunc_8, NODE_2);
    deftrans(NODE_START, (CONDFUNC)&SmplNet::condfunc_5, NODE_1);

    defnormalnode(NODE_1, (ACTFUNC)&SmplNet::actfunc_1,
     (ACTFUNC)&SmplNet::preactfunc_1, (ACTFUNC)&SmplNet::postactfunc_1);
    deftrans(NODE_1, (CONDFUNC)&SmplNet::condfunc_8, NODE_END);
    deftrans(NODE_1, (CONDFUNC)&SmplNet::condfunc_5, NODE_2);

    defnormalnode(NODE_2, (ACTFUNC)&SmplNet::actfunc_2);
    deftrans(NODE_2, (CONDFUNC)&SmplNet::condfunc_5, NODE_END);

    defexitnode(NODE_END);
// initializing drand48
    struct timeval tv;
    struct timezone tz;
#ifdef sgi
    BSDgettimeofday(&tv, &tz);
#else
    gettimeofday(&tv, &tz);
#endif
    srand48(tv.tv_sec);
}

SmplNet::~SmplNet()
{
    cout << "Executing SmplNet destructor\n";
}
```

Figure 3: A program file for `SmplNet` (contd.).

```
defnormalnode(NODE_START, (ACTFUNC)&SmplNet::actfunc_s);
```

defines that the node `NODE_START`$(= 0)$ is a normal node where the action `actfunc_s()` is executed every tick. If action functions are provided as the third and forth arguments in `defnormalnode()`, they are the pre-action and post-action of the node, respectively. The pre-action is executed once when the state changes to the node, while the post-action is called once when the state changes to another node.

For defining transition rules, `deftrans()`s are used. For example,

```
deftrans(NODE_START, (CONDFUNC)&SmplNet::condfunc_8, NODE_2);
```

defines that, in `SmplNet`, the state changes to `NODE_2` if `condfunc_8()` returns `TRUE`; otherwise, the state is kept to be `NODE_START`

The transition rules represented by `deftrans()` are evaluated in the order of `deftrans()`s. If the two lines:

```
deftrans(NODE_START, (CONDFUNC)&SmplNet::condfunc_8, NODE_2);
deftrans(NODE_START, (CONDFUNC)&SmplNet::condfunc_5, NODE_1);
```

are exchanged, then any value in $[0.5, 1]$ makes the state change to `NODE_1`.

`defexitnode()` defines an exit node, where the PaT-Net is terminated. For example,

```
defexitnode(NODE_END);
```

defines the node `NODE_END` to be an exit node.

2 Running PaT-Nets

Figure 4 is a simplest program running `SmplNet`, and an output from the program is shown in Figure 5. `LWNet::advance()` makes a single-tick execution.

To create and delete PaT-Nets arbitrarily in programs, `new` and `delete` operators in C++ are used. A sample program is shown in Figure 6.

To manage and execute PaT-Nets in a unified fashion, class `LWNetList` is prepared. `LWNetList::addnet()` is used to register a PaT-Netin `LWNetList`, while `LWNetList::advance()` is to make a single-tick execution for all the PaT-Nets on the `LWNetList`. Figure 7 illustrates the usage of `LWNetList`.

```
#include "smplnet.h"

void main(void)
{
    SmplNet snet;

    while(snet.advance())
        ;
}
```

Figure 4: A program for running SmplNet.

```
LWNet constructor
Executing SmplNet constructor
Get 0.428522 in actfunc_s
Get 0.0187866 in actfunc_s
Get 0.5905 in actfunc_s
Executing preactfunc_1
Get 0.330349 in actfunc_1
Get 0.853521 in actfunc_1
Executing postactfunc_1
Executing SmplNet destructor
LWNet destructor
```

Figure 5: An output from the sample program

```
#include "smplnet.h"

void main(void)
{
    SmplNet* snet = new SmplNet;

    while(snet->advance())
        ;
    delete snet;
}
```

Figure 6: A program for running SmplNet.

```
#include "smplnet.h"

void main(void)
{
    LWNetList::addnet(new SmplNet);

    while(LWNetList::advance())
        ;
}
```

Figure 7: A program for running `SmplNet`.

A major difference between the Figures 4 and 6 and Figure 7 lies in the execution of the destructor `SmpleNet::~SmplNet()`. In `LWNetList`, a PaT-Net is treated not as an instance of a derived class of `LWNet` but as one of the class `LWNet`. Thus `SmpleNet::~SmplNet()` is not executed in Figure 7. To make a terminal processing in a PaT-Net stored in `LWNetList`, the terminal processing should be performed in a dummy node, and then the dummy node is placed just before an exit node.

3 Running PaT-Nets in Jack

A simplest way to run PaT-Nets in *Jack* is to manage PaT-Nets in `LWNetList` and then make `LWNetList` execute every tick with `BindSimulationFunction()`. A sample modification to `jack_main.c++` is shown in Figure 8.

4 Node Types in PaT-Nets

The class `LWNet` has eight types of nodes: normal-node, call-node, par-node, join-node, indy-node, kldp-node, halt-node, and exit-node. This section explains the usages of these types of nodes.

Normal-node. As described in Section 1, normal-nodes are defined by `defnormalnode()`:

```
    void defnormalnode(NodeId dnid, ACTFUNC dact,
     ACTFUNC dpreact = 0, ACTFUNC dpostact = 0);
```

```
int
main (int argc, char *argv[])
{
    ...

    BindSimulationFunction(LWNetList::advance, 0);  // inserted

    ...

    /*
     * Parse command line files
     */
    parsefiles(argc,argv);

    DoCmds();

    return(0);
}
```

Figure 8: A sample modification to `jack_main.c++` in *Jack*.

`dnid`, `dact`, `dpreact`, and `dpostact` are a node id, an associated action function, a pre-action function, and a post-action function, respectively. The associated action function is executed every clock tick if the state is in the node. The pre-action function is executed once when the state is changed to the node, and the post-action function is executed once when the state is changed from the node to another node.

Call-node. Call nodes are defined by `defcallnode()` in the form of:

```
    void defcallnode(NodeId dnid, LWNEW dfnew);
```

The node `dnid` calls another PaT-Net invoked by `dfnew` and then wait until the child net terminates. The type `LWNEW` is declared as:

```
    typedef LWNet* (*LWNEW)(void);
```

and it is a "new" function for the class of the child PaT-Net. A sample "new" function for `SmplNet` is shown in Figure 9.

Par-node. Our C++-PaT-Net is extended so that it can have multiple states at the same time. Par nodes spawn several states, each of which corresponds to a node, and are defined by `defparnode()` in the form of:

```
LWNet* new_SmplNet()
{
    return new SmplNet();
}
```

Figure 9: A "new" function for `SmplNet`.

```
    void defparnode(NodeId dnid, NodeId br0, NodeId br1,
     NodeId br2 = -1, NodeId br3 = -1);
```

The node `dnid` spawns several states corresponding to `br0`, `br1`, `br2`, and `br3`. In the current implementation, two to four states are spawned at the same time. Depending on the number of the spawned states, `br2` and `br3` can be omitted. These spawned states are usually joined together by a Join node described below. The first spawned state corresponding to `br0` has precedence over the other spawned states. This precedence is related to kldp nodes.

Join-node. Join nodes join together the spawned states, and are defined by `defjoinnode()` in the form of:

```
    void defjoinnode(NodeId dnid, NodeId dpar, NodeId dnext);
```

The node `dnid` waits until all the states spawned by the par node `dpar`, and then the state moves to the node `dnext`. If some of the states spawned by `dpar` are terminated by indy/kldp/halt nodes as described below, then the join node `dnid` waits eternally. If a par node p_1 spawned by another par node p_0 further spawns the states, then all the states/nodes spawned by p_1 should first be joined by a join node corresponding to p_1, and then the states/nodes spawned by p_0 should be joined by another join node.

Indy-node. Indy nodes are defined by `defindynode()` in the form of:

```
    void defindynode(NodeId dnid, NodeId dpar, NodeId dnext);
```

If the control comes to the indy node `dnid`, then the nodes spawned by the par node `dpar` and their descendents (further spawned nodes) are all terminated, and the control moves to the node `dnext`.

Kldp-node. As described above, the first spawned state/node in a par-node has precedence over the other spawned states. We call its the other nodes are *dependents* of the first node. If a state/node s_1 is a dependent of another state/node s_0, then all the nodes spawned by s_1 are also dependents of s_0. If a state/node s_1 is a dependent of another state/node s_0, then s_1 is also the dependent of the *first* node spawned by s_0 The above rules are applied recursively. Kldp nodes are defined by `defkldpnode()` in the form of:

```
void defkldpnode(NodeId dnid, NodeId dnext);
```

After the kldp node `dnid` kills all its dependents, the control moves to the node `dnext`.

Halt-node. Halt nodes are defined by `defhaltnode()` in the form of:

```
void defhaltnode(NodeId dnid);
```

The halt node `dnid` terminate the node itself without invoking its following nodes. If the number of current active states/nodes is zero, then the PaT-Net itself is terminated.

Exit-node. Exit nodes are defined by `defexitnode()` in the form of:

```
void defexitnode(NodeId dnid);
```

The exit node `dnid` terminates the PaT-Net itself.

5 Other Features on Conditions and Transitions

Condition functions. Two condition functions `defaultcond()` and `finishcond()` are prepared in C++-PaT-Net. The function `defaultcond()` which always returns `TRUE` enables us to make transitions form a node immediately. The function `finishcond()` returns `TRUE` in a node immediately after the function `markfinished()` is called in the node.

Node transition via node branch functions In addition to node transitions via pairs of a condition function and a next node id, the C++-PaT-Net supports node transitions via pairs of a condition function and a node branch function which returns the next node id. The function `deftrans()` is overloaded as follows:

```
    void deftrans(NodeId dnid, CONDFUNC dcond, NodeId dnext);
    void deftrans(NodeId dnid, CONDFUNC dcond, NODEBRNCFUNC dnbf);
```

The type `NODEBRNCFUNC` is declared as:

```
    typedef NodeId (LWNet::*NODEBRNCFUNC)(void);
```

The order of evaluation of condition functions is independent of the third argument of `deftrans()`. Condition functions for a node are evaluated in the order of associated `deftrans()` in the program.

6 Communication via message passing

Communication between LWNet is realized by sending a request from one LWNet to another LWNet, and later returning its reply from the receiver of the request to the sender. Request format and reply condition/format can be defined as required. Waiting should be explicitly defined in some action function in each LWNet.

To send a message to a PaT-Net, `requestjob()` is used. It is in the form:

```
    void requestjob(int type, void* data, JobReport* ret);
```

and sent to a PaT-Net`a` via `a.requestjob(...)`. Typically request types and associated data (attributes and parameters) are sent via `type` and `data`, respectively. The concrete format of `data` should be pre-determined among senders and receivers.

The type `JobReport` is used to send a reply from the receiver. Its general definition is given as:

```
class JobReport {
public:
    virtual void reset() = 0;       // reset JobReport as UNREPORTED
    virtual int isreported() = 0;   // TRUE if reported; FALSE otherwise
};
```

`JobReport` should be "unreported" until the reply is issued. `reset()` is used to initialize/reset `JobReport`, and `isreported()` returns whether the reply is issued (`TRUE`) or not (`FALSE`). Concrete classes for reply message should be defined as a derived class of `JobReport`. The most typical derived class of `JobReport` is given as `GPJobReport` in `gpjobrep.h`.

Once the `requestjob()` is sent from the PaT-Net n_p to the PaT-Net n_c, the net n_c processes the requested job with some functions like `procrequest()`. n_c can judge whether it is requested by `isrequested()`. If n_c replies, for example, by assigning `COMPLETED` to `result` in `GPJobReport`, n_p knows it by `isreported()`. In other words, n_p typically calls `isreported()` every clock tick. In case of `GPJobReport`, `isreported()` returns non-zero if and only if `Result` is not `UNCOMPLETED`, and the above works well.