Alternate Pi-Demos Simulators

COMP 522

Project Report

Eugene Syriani

Table of Contents

Introduction

PiDemos is yet another formalism for process interaction.
"*[piDemos is] a small process-oriented discrete event simulation language.*" [3]

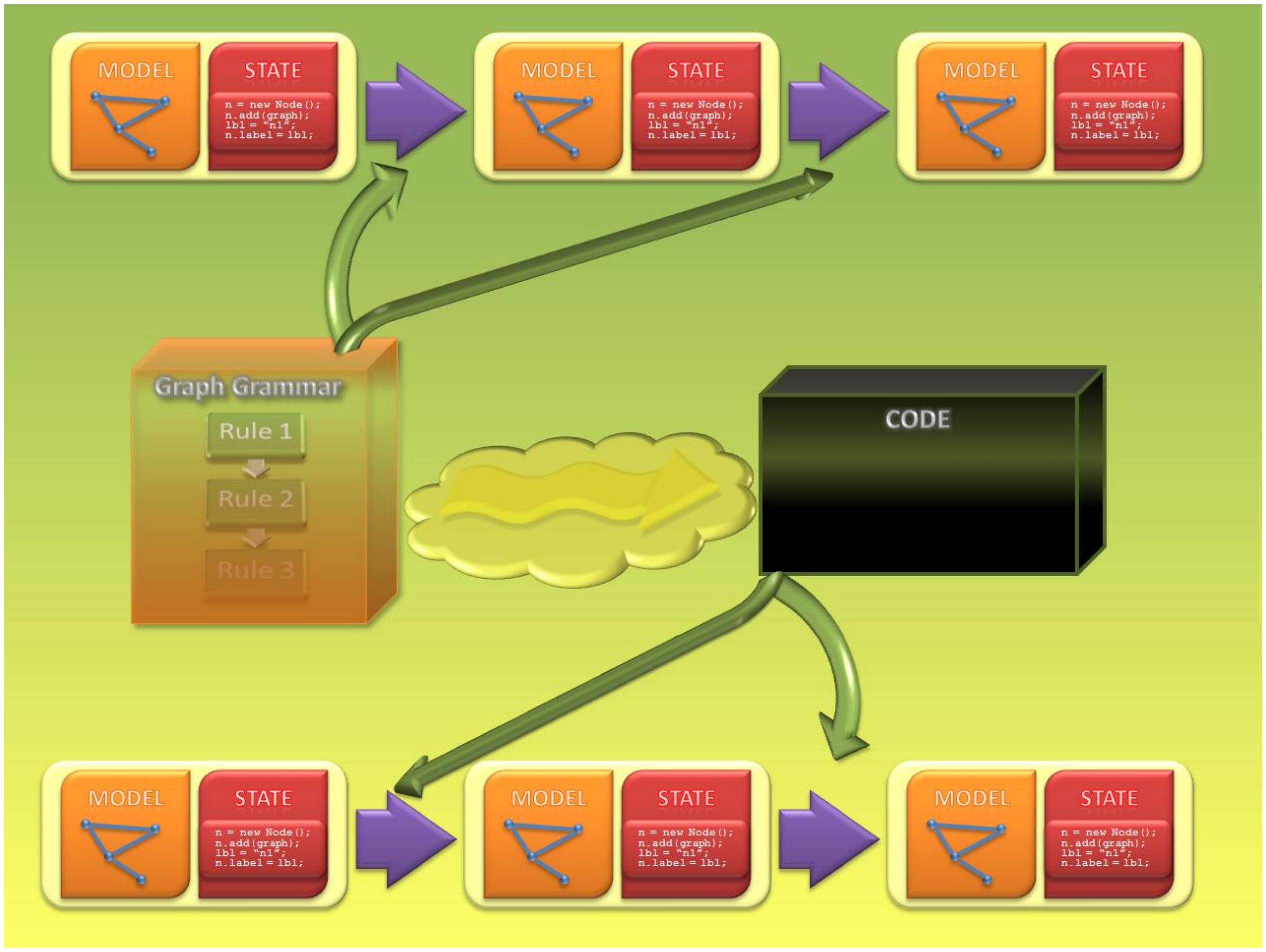
I have previously developed a visual interpretation of this textual language. Also, I have brought it up to life by means of a simulator.
This simulator was based on a graph grammar which, following certain rule patterns, was simulating the behaviour, the semantics of this modelling language.

Bear in mind that this was all done in an AToM^3 environment.

Now, this project will show a way of programming the graph grammar rules (the behaviour of the model) and then extend this to a piDemos kernel, outside of AToM^3 as a plug-in to it. This kernel will take as input a model of the piDemos language (more precisely, the graph from AToM^3) and output a trace of the different transformation steps. In principle they should match the trace produced by the graph grammar.

Testing will ensure the correspondence between these two approaches.

The ultimate goal would be to automatically generate the graph grammar: programmed rewriting system.



The Model

The purpose of having a programmed graph rewriting is to have full access to the control structure. There is a particular formalism that can accomplish that (and even more): the architecture underneath is a *DEVS* model.

User Input

One of the initial decisions was to make the piDemos simulator a plug-in to AToM^3. This is encapsulated by the *UserInput* block. Its behaviour is similar to the graph transformation simulator currently in AToM^3. *UserInput* is an atomic DEVS block that sends graphs and steps and receives terminations.

Graphs are ASG_PiDemos graphs. They represent the abstract syntax graph of a model of the language piDemos in AToM^3.

Steps represent the number of steps the user requests the simulator to perform in a row. If 0 is given, then the simulation ends. If ∞ is given, then the simulation is run in continuous mode.

The reception of a Termination means that either the steps have been performed or that the execution has reached its end. In the latter case, no transformation can be applied to the graph.

Controller

The *Controller* plays the role of the brain of this model. This atomic DEVS block is essential for the logic between the external input and the transformation set. It is the control that receives the graph to transform and the number of steps to be applied. It also acknowledges the user about termination. The *Controller* sends the graph to the transformation kit and waits for a graph in returned: the graph can either be modified or not. This is repeated depending on the steps received.

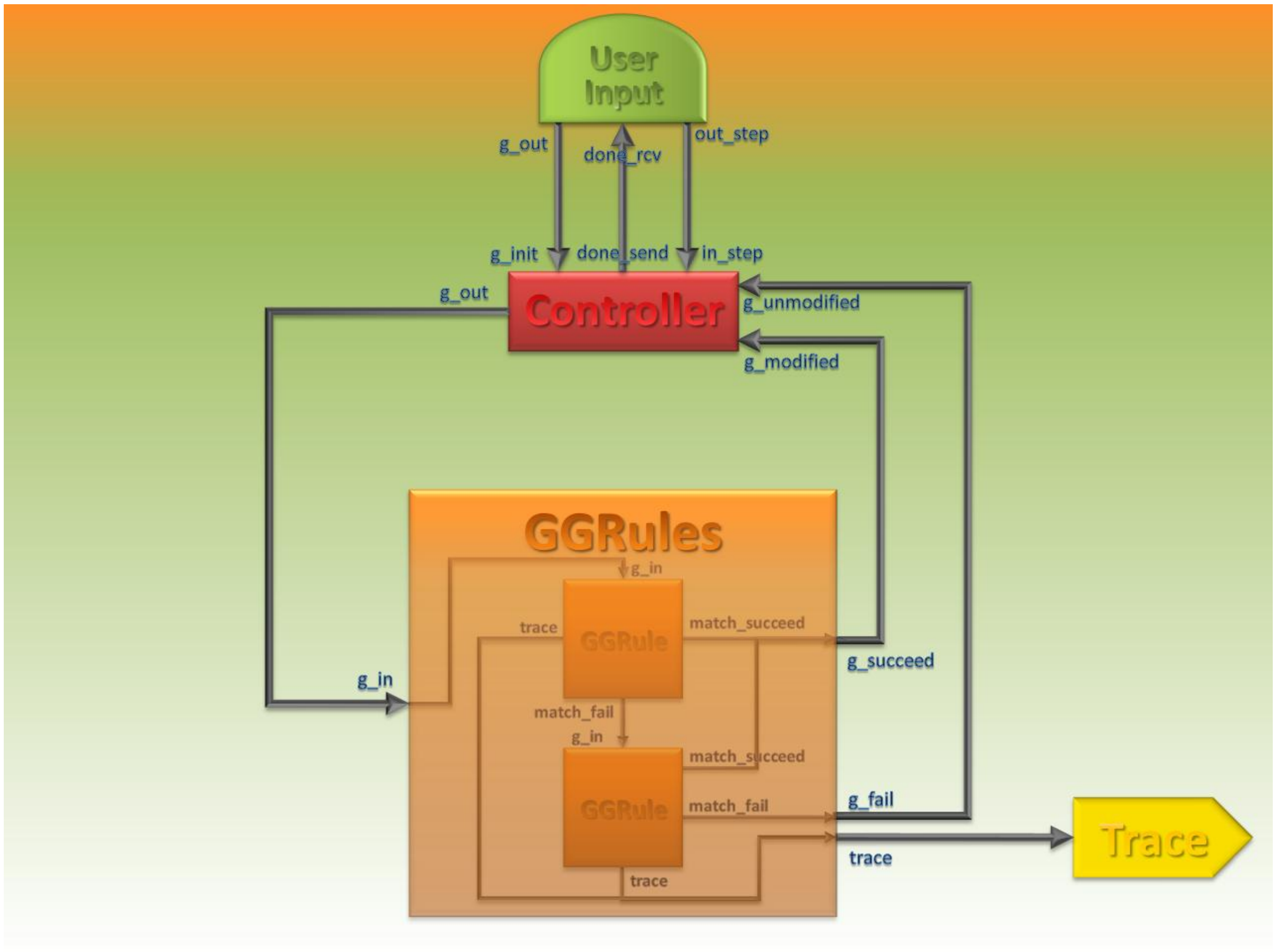
Note that the system could in principle receive several graphs at any time. Also, the user could request for more steps even when there are some steps left.

GGRules

Now we get into the core of the application: the transformation block. *GGRules* is the outside view of what is internally happening. It receives a graph and outputs a graph (a graph that has succeeded a match or failed matching all the rules). It is a coupled DEVS block that encapsulates the graph rewriting procedure. *GGRules* sends a trace of the rule applied during each step.

GGRules is composed of one or more *GGRule* blocks. Each *GGRule* satisfies certain properties. There is at most one rule that is applied per step. If a rule fails, the graph is sent to the next rule until the last rule is reached. If the last rule also fails, then no rules have been applied in this step, hence *GGRules* sends back its input graph. Otherwise it is the newly transformed graph that is sent back.

More details on the execution of a *GGRule* will follow later.

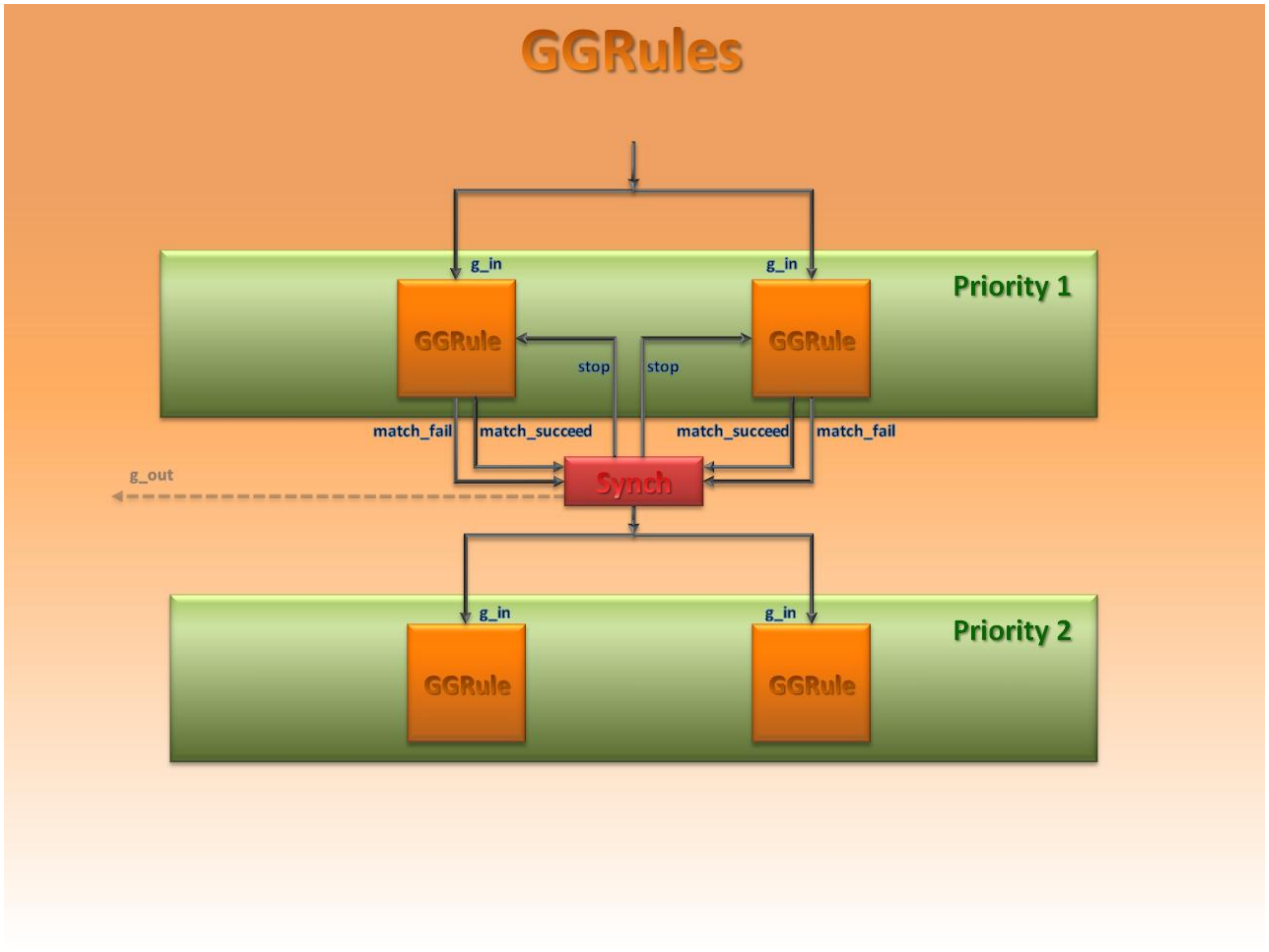


Assumptions

During the modelling part of the system, some interesting interrogations have arisen.

Let's look at the rules as a whole (this is the intention of *GGRules*). We might, for obvious reasons, assign priorities to each of the rules. This determines an ordered sequence of the rules. But what if two rules have the same priority? This means that any one of them that have a matching will be executed. How to model this situation in our DEVS model?

I found out that having a *Synchroniser* was the trivial answer. Each rule with the same priority (call it a pool) will receive its input in parallel from the previous pool. A failed matching of a rule is notified to the *Synchroniser*. If it has received failure notices from all rules in the pool, then it passes the input to the next pool. On the other hand, as soon as one rule has successfully executed, it notifies the *Synchroniser* which, in turn, stops the rest of the rules in the pool from executing. It then sends the output out of *GGRules*.



But this implies further assumptions to be made. One of the major drawbacks of the *Synchroniser* is the lack of modularity. The *Synchroniser* must know about how many rules are in the corresponding pool to send and receive information. Of course this goes in contradiction with the intention of a DEVS model which is fundamental property is to be modular. Hence, I opted for a much simpler solution reflects the same behaviour from the outside. That is to link the rules one by in sequence in the order of their priority. If *GGRule* A and *GGRule* B turn out to have the equal priority, then arbitrarily choose If *GGRule* A (or *GGRule* B) first, followed by *GGRule* B (or *GGRule* A). I have allowed myself such a simplification because of the sequential behaviour of today's computers. But the model would perfectly be valid if I were to choose the former solution.

Code behind GG rules

I have provided an operational semantics for the piDemos modelling language by means of code.

Template-style coding

Since the ultimate goal is to be able to automatically transform the different rules into code, I have written them in an as much generative style as possible. While doing so, I have started to come up with certain template forms of writing. This is a snippet of the code that matches a graph to the *EXIT* rule:

```
match = 0
try:
    for end in graph.listNodes['End']:
        for contains in end.out_connections_:
            if contains.__class__.__name__ == 'Contains':
                for transaction in contains.out_connections_:
                    if transaction.__class__.__name__ == 'Transaction':
                        if transaction.isMoving.getValue() == ('True', 1):
                            match = 1
                            break
                    if match: break
            if match: break
        if match: break
except:
    return None

if not match:
    return None
```

Note that `graph` is a `Graph` object which extends the `ASG_PiDemos`. Here is a snippet of the code for the transformation of in the *MOVE* rule:

```
node in contains.in_connections_:
    node.out_connections_.remove(contains)
for node in contains.out_connections_:
    node.in_connections_.remove(contains)
nodeList = graph.listNodes[contains.__class__.__name__]
nodeList.remove(contains)
graph.listNodes[contains.__class__.__name__] = nodeList

contains = Contains()
graph.listNodes[contains.__class__.__name__].append(contains)
contains.in_connections_.append(block2)
block2.out_connections_.append(contains)
transaction.in_connections_.append(contains)
contains.out_connections_.append(transaction)

block1.blocked.setValue(('True', 0))
block2.blocked.setValue(('True', 1))
transaction.resumed.setValue(('True', 0))
```

Matching Algorithm (LHS)

A rule has to first check if it can apply the transformation. This is done by iterating through the graph starting by a node and visiting its neighbours.

The initial node would be determined by means of a dependency graph on the LHS of the rule. The graph exploration starts from each one of the roots of the dependency graph in turn. For each visited node check its type and the values of its attribute.

Interesting cases happen when we look for a particular structure in the graph. I have encountered two of them here: lists and queues. List operations can be found in the `Hold` class. Queue operations can be found in the `Get` and `Put` class.

Transformation Algorithm (RHS)

The transformation part follows a very narrow template. It is divided in three steps: removal, insertion and modification of nodes.

When removing a node, links going in and out of that node should also be removed.

When creating nodes, links to other new nodes already existing nodes should also be established.

Setting the values inside nodes may need to refer to other nodes.

With more work a much more rigid template can be elaborated to ease the generation of code.

Simulation and Testing

Simulation of the DEVS model leads to an output of the trace of the different rules used for the transformation of the input graph.

Several input graphs have been created and fed as parameters to the simulator.

Tests have been driven by graphs that lead to no transformation or full transformation. Even a graph that leads to deadlocking has passed with correct behaviour of the GG system.

Source files

All code files have been programmed in Python 2.5

The folder `Coded_PiDemos_Simualtor` contains all needed files to run simulations.

`GraphTransformExperiment.py` is the file that executes the simulation(s).

`GraphTransformModel.py` is the file that contains the complete DEVS model.

`GraphTransformRules.py` is the file that contains all the different rules, one per class.

`Graph.py` is the file that defines the input graph. Some examples for testing have been added to it.

`pydevs` is the folder containing the DEVS definition for python.

`ATOM3` is the folder containing a subset of what is needed from AToM^3.

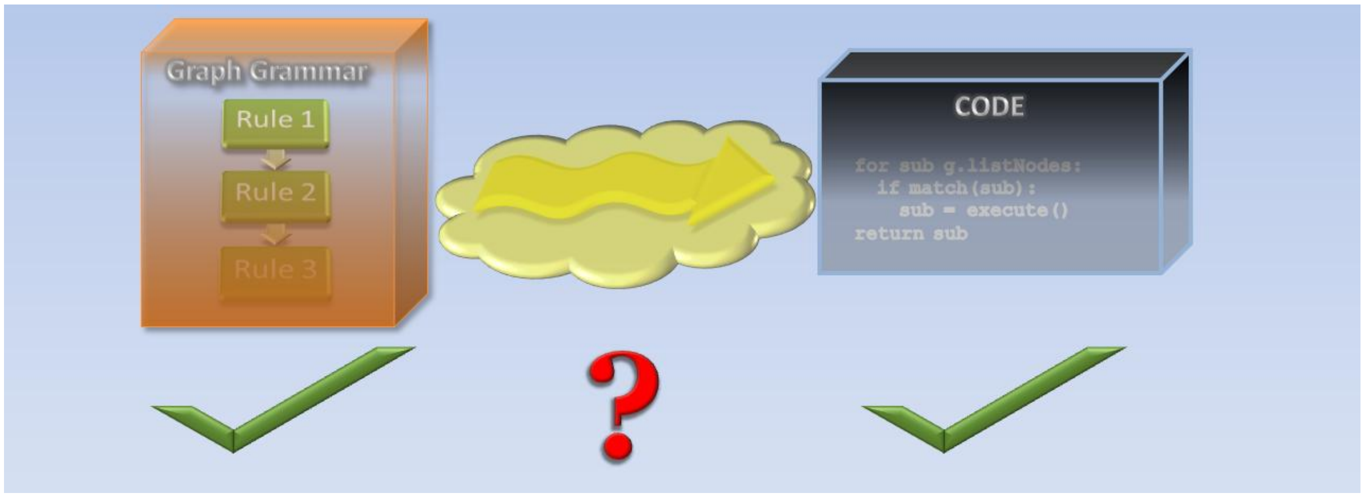
Conclusion

One of the powers of DEVS models is the notion of time. In this project I did not make use of it at all. One may think about extending the Graph Grammar transformations to timed GGT.

A slight extension to the project can be to integrate it with the complete AToM^3 source code. Here, I have taken a simplified version of the classes needed to run the simulator.

Another improvement that can be brought to the model is at the level of the actual code inside each rule. I have based myself on the code generated by AToM^3 for graph transformations. This is a rather ad-hoc approach and not very efficient. Instead, one could use Himesis graphs and make use of the very efficient matching algorithm to significantly increase the running-time.

Also, after the results of this project, I am quite certain that it will be possible to automatically generate graph transformation and more generally model transformation.



```
for sub g.listNodes:
  if match(sub):
    sub = execute()
return sub
```

References

[1] Eugene Syriani, *Modelling syntax and semantics of piDEMOS in AToM^3*, MSDL Summer Presentations, 2006

[2] Hans Vangheluwe, *The Discrete EVent System specifcation (DEVS) formalism*, Modelling and Simulation Lecture Notes, 2002

[3] G. Birtwistle and C. Tofts, *Operational Semantics Of Process-Oriented Simulation Languages --Part 1: piDemos*, 1993

[4] Marc Provost, *Himesis: A Hierarchical Subgraph Matching Kernel for Model Driven Development*, 2005