Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 603–610

Architectural Redesign of a Distributed Execution Environment

Cosmin M. Poteraș
University of Craiova,
Department of Computers and Information Technology
Faculty of Automation, Computers and Electronics
Craiova, Romania
cpoteras@software.ucv.ro

Mihai Mocanu
University of Craiova,
Department of Computers and Information Technology
Faculty of Automation, Computers and Electronics
Craiova, Romania
mocanu_mihai@software.ucv.ro

Marian Cristian Mihăescu
University of Craiova,
Department of Computers and Information Technology
Faculty of Automation, Computers and Electronics
Craiova, Romania
mihaescu@software.ucv.ro

***Abstract*—This paper describes the architectural redesign of a distributed execution framework called State Machine Based Distributed System which uses a state machine-based representation of processes in order to reduce the applications development time while providing safety and reliability. Initially the system has been built on top of the .Net Framework employing static programming techniques and made use of a custom data storage. The new architecture is intended to take advantage of the fast growing technologies like dynamic languages and graph databases for speeding up even more the applications development and improve the dynamism of the execution model.**

I. Introduction

NOWADAYS, complex systems have become more and more demanding for online visualization and computational steering. Analyzing the outcome of a complex simulation as a post-processing phase is almost unacceptable, while interactivity is a must-have feature. Distributed computing as well as super fast networks, have come to rescue, offering a proper environment for achieving high performance simulation, online visualization and steering. Building complex systems is more a matter of integrating state of the art tools into execution platforms.

Online visualization and computational steering techniques play a very important role in speeding up simulations by allowing on-the-fly analysis and guidance of the ongoing process. Computational steering is nothing else than manual intervention against the ongoing process with the purpose of guiding it towards the space of interest. Computational steering occurs at three different levels: the program level (***program steering,***) which implies changes on the program's state (shared variables), data level (***data steering***), which allows interventions against the data space, and execution level (***dynamic steering***), which implies direct changes to the program's flow by injecting code or invoking routines.

Developing distributed simulation platforms able to provide the means for interactivity has not been easy. Many surveys have documented the task [1], [2], however only few of them get close to fulfilling production needs. It's worth mentioning few of them.

Collaborative Online Visualization and Steering framework, COVS [3] integrates tools for visualization (VTK), communication libraries (VISIT, PV3), steering tools (VISIT, ICENI, gViz).

RealityGrid [4]. [5] is a library which serves as an API which uses check-pointing techniques for steering commands.

CUMULVS (Collaborative User Migration, User Library for Visualization and Steering) [6], [7] developed at Oak Ridge National Laboratory, besides steering, it benefits of powerful recovery techniques and tasks migration.

CSE [8], [9] (Computational Steering Environment) carries out steering through a data manager which works closely with simulation processes (called satellites).

After examining all these platforms it became obvious that choosing the right tools for a distributed simulation and steering system is not an easy task. While ensuring scalability, portability, flexibility, extensibility, could be achieved with the proper tools, ensuring safety seems more like a design task. Writing safe code does not guarantee that the output of the application will be safe. Just imagine the effects of a malfunction of a medical software which assists a surgery. It is essential that at each moment in time, the application is in a consistent and expected state. When designing the application one must make sure that the application reacts accordingly no matter what. This led us to the idea of representing the tasks (processing units) as finite state machines. This way, we force the application developer to consider all states that the application may step into, prior to implementing them properly. Besides an execution model (transitioning states until a final state has been reached) finite state machines serve as packages which can be spread across a distributed environment leading our way towards load balancing and recovery algorithms.

A similar approach has been introduced in [10], and it made use of Statecharts as a conceptual model for a visual tool called Statemate. Statemate is a very powerful simulation environment for statecharts-based models. Statecharts imply a hierarchical and compositional structure increasing the complexity of the model which could lead to an increase of the risk of design errors.

Our previous researches have resulted in the implementation of a state machines-based distributed framework (SMBDS) [11] consisting of an execution engine as well as a class library. The class library reduces the development time considerably by including a set of interfaces and base classes. Developing new applications on top of the frame-

978-1-4673-4471-5/$25.00

work only requires the implementation of these interfaces and classes (parameters and state machines). The execution engine loads them and runs them on the distributed environment. SMBDS has been implemented on top of the .Net Framework using C# language. In order to be able to dynamically load custom parameter/state machines classes, the .Net Reflection package has been used. However, C# still remains a static language and for this reason all nodes in the distributed system needed to have access to all code files which makes the distributed environment harder to maintain (whenever a new type of state machine is needed, all nodes need to be updated with the corresponding code file). To overcome this drawback we started considering the power of dynamic languages. There were arguments in favor and against dynamic languages, which will be discussed later in this paper. However we considered the advantages of using dynamic languages worth assuming the drawbacks. This led us to a redesign process which changed the framework almost entirely. At first, we considered only including the dynamic code as a string and simply run it. But then, as we dig into the load balancing and scheduling algorithms, we realized that the execution model is similar to a graph, so why not changing the execution engine so we can take advantage of the NoSQL graph databases. We will discuss the execution model later in this paper.

The rest of the paper is organized as follows. Section II shortly presents the technologies used for implementation pointing out the main benefits of using them. Section III shortly describes the architecture of the state machines-based distributed system (SMBDS) as it used to be prior to the redesign process and explains the reasons behind redesign. Section III presents a new execution model for the state machines-based distributed framework. Section V concludes the paper and presents our future development intentions.

II. Tools and Technologies

In this section we will argument our decisions regarding the infrastructure.

The first choice that we have had to make after deciding to use a dynamic language, was which language we should use. The most popular dynamic languages that we have considered were Python and Ruby. A 'versus' discussion between Ruby and Python is beyond the scope of this paper. After all they can both reach the scope of our framework. However we decided to go for JRuby, as it stands on top of the Java Virtual Machine which makes it possible to call java code from JRuby offering the possibility of including dynamically invoked java code in our state machines.

The second choice that we have had to make was related to the graph representation of our execution model. We have examined the most popular graph database infrastructures, namely Neo4j, Titan Server with three storage back ends (Apache Cassandra, Apache HBase, Oracle Berkley DB) together with the Tinkerpop stack [12].

Neo4j proved to be the most mature graph database. However it falls fast in distributed environments as it's storage is limited to only one machine.

Titan claims to bring a lot of performance boost. Even though it is a relatively new product and it has some limitations (for example, the most important: indexes need to be declared before first use of the key), Titan Server seems a good choice since it uses very powerful storages behind it.

Considering the CAP theorem (any practical database system can only provide two of the following three: scalability, availability and consistency), Titan can use three storage backends: Apache HBase, which provides consistency and scalability, Oracle Berkley DB which provides consistency and availability and Apache Cassandra which provides scalability and availability.

In our case HBase and Cassandra seem to be the best choices depending on the applications needs (distributed systems require scalability). There is a tradeoff between high availability (Cassandra) and consistency (HBase). Both of them are scalable therefore they are suitable for our framework's needs. Their performance and reliability have been successfully proven in production environments at Facebook or Twitter.

As HBase and Cassandra are open-source, distributed and column-oriented storages, and not really graph databases, it might feel strange to use them as graphs, however the Tinkerpop stack models successfully the graphs on top of these storages and they claim for very high performance results [13], [14], outperforming regular SQL relational databases.

The Tinkerpop stack includes a very powerful graph traversal language, Gremlin, built on top of a widely used java interface for graph databases, Blueprints. This makes Gremlin compatible with multiple existing graph engines and eventually with future engines that will implement the Blueprints interface. Being implemented on top of JVM, Gremlin can run java code, so one can actually traverse the graph with custom java code.

Putting all these pieces together we get a very powerful distributed platform for our framework.

Fig. 1 illustrates the software stack for our framework. JVM is where all tools met. On top of it runs JRuby together with all libraries that implement Blueprints. As a bridge between SMBDS and the graph database storages we are using the Tinkerpop's Gremlin language which complies to the Blueprints interface.

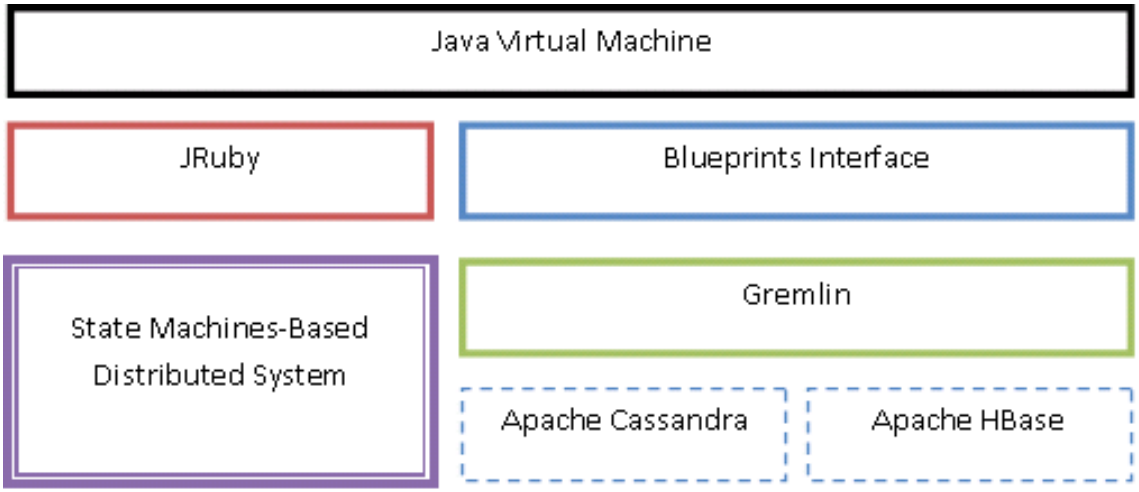

Fig. 1 Software stack

III. Framework's Architecture

This section is intended to reveal the motivation behind the redesign process. Let's examine the static architecture of the system before we proceed with the redesign process.

The challenge behind SMBDS was to offer safety and reliability while taking advantage of a distributed execution environment. SMBDS aims of ensuring safety at design time rather than code safety only, and it does that by representing all computational tasks as finite state machines. This representation forces the application developer to consider all states that the application may step into and react accordingly to each of them. This eliminates the erroneous states while offering a robust and traceable execution model.

The architecture of SMBDS is illustrated in Fig. 2.

SMBDS consists of five modules: Simulation Module (or Processing Module), Visualization Module, Control and Communication Module, Shared Memory and Client Application.

All the magic of SMBDS happens on the simulation level as this is where the simulation tasks, represented as state machines, are being executed.

Each node of the system owns a state machines manager which is the bridge between all modules, mainly because it is responsible for acquiring the state machines from the distributed environment, migrating them to other hosts when necessary, running the state machines and at the same time interacting with the client application for handling steering commands and providing visualization information.

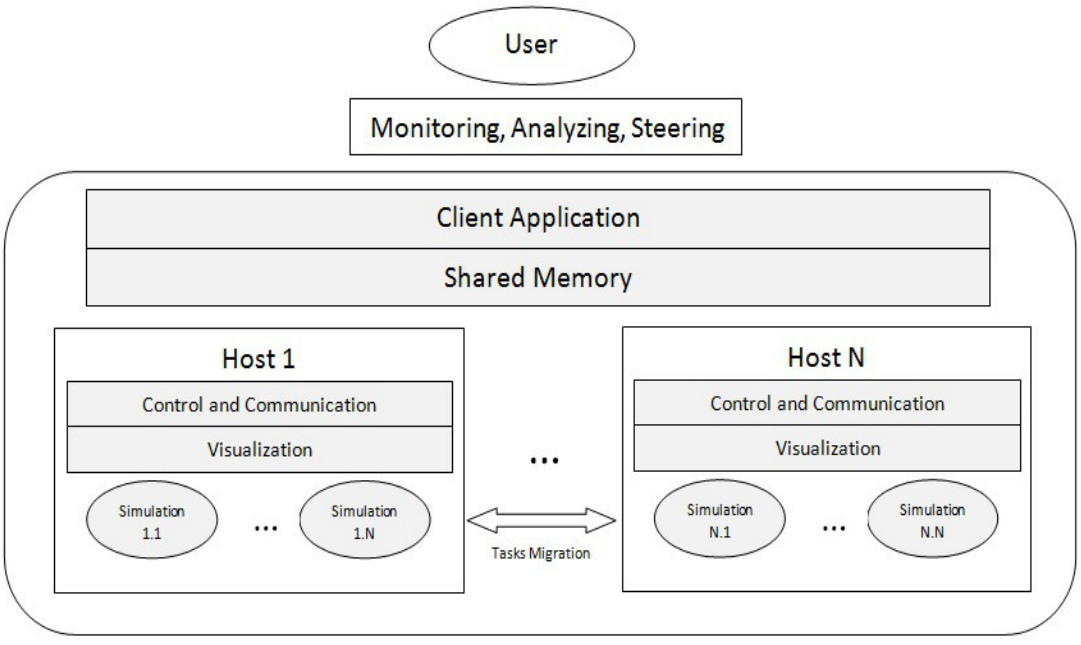

Fig. 2 State Machines-Based Distributed System's architecture.

The control and communication module is able to acquire data, forward output data towards the visualization pipeline while monitoring the available resources and executing steering commands.

As we are dealing with a decentralized architecture, the shared memory took the form of a distributed storage space and it holds the system's parameters (steerable variables).

The user analysis the output data filtered by the visualization pipeline, monitors the state of the computational resources as well as the execution distribution across nodes, and interacts with the system by launching new tasks (state machines), changing the execution parameters (stored in the shared memory), guiding the simulation towards the space of interest.

Concerning the applications development, SMBDS exposes a class library which facilitates the implementation of custom finite state machines which once created, will be passed to the state machines managers and executed.

Implementing new custom state machines requires the implementation of an interface (IParameter), representing the state machine's parameters, if not already implemented, and the extension of a state machine class (StateMachine). Based on these two, the framework's engine is able to manage the execution of state machines. Figure 3 illustrates the class diagram of the framework's execution engine.

The engine's main class is the StateMachineManager. Each host will launch a state machines manager which will run continuously until final states are reached. The main role of this class is to manage the execution of state machines but also migrate the state machines to (pack and send) and from (receive and unpack) other hosts (state machine managers). Packing a state machine consists of pausing the execution and extracting data from the state machine (extracting StateMachineData object), while unpacking restores the machine's execution by creating an instance of the corresponding state machine class, initializing the state machine with the StateMachineData object and lastly resuming the execution from the state where it has previously been paused. Any computations performed for the interrupted state, carried out before packing, will be discarded.

It becomes obvious that migrating a state machine consists of simply sending the StateMachineData object towards the destination host. The StateMachineData class holds execution data of a state machine: a list of parameters, a transition table, the current state, a unique identifier (needed for tracking the machine especially when dealing with migration), the type of the state machine (used to pick the correct state machine class when packing/unpacking the machine using .Net Reflection) and a list of final states.

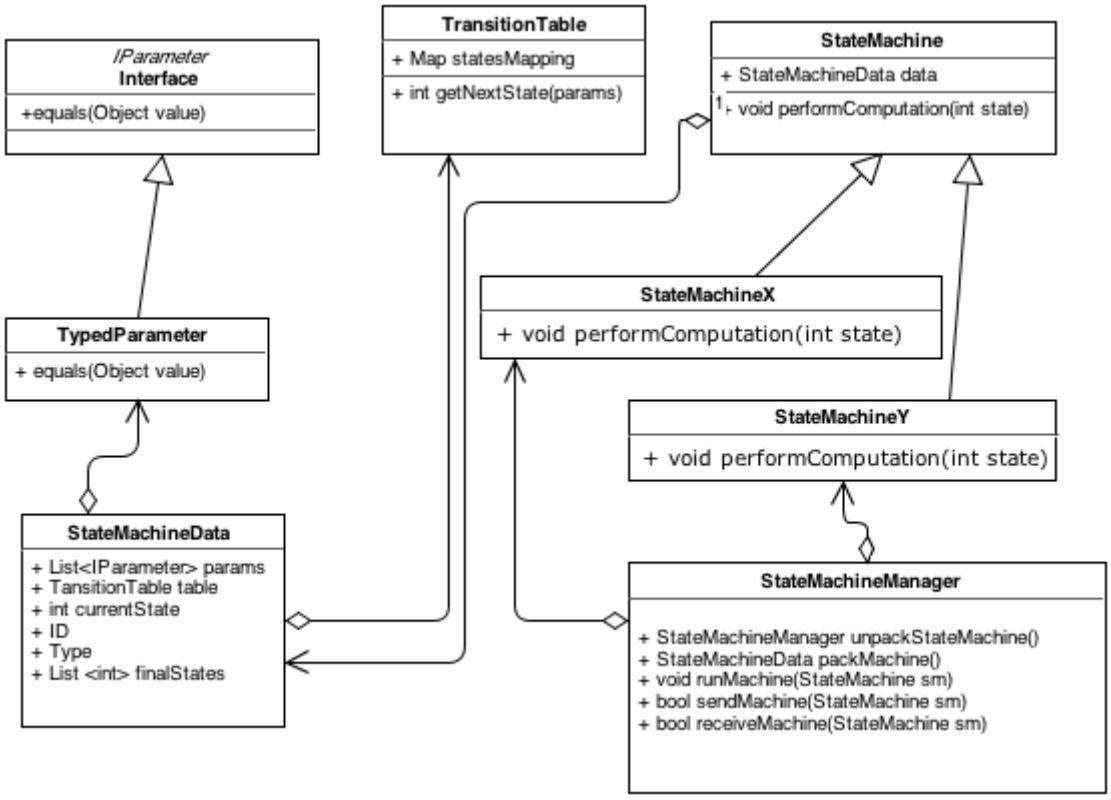

Fig. 3 SMBDS class diagram.

The StateMachine class exposed by the framework's library is an abstract class exposing an abstract method called performComputation which executes the code associated with the current state. This method is being invoked by the state machines manager after each transition. The computations are carried out taking as input the values of the machine's parameters. The parameters are usually altered by the computations of the current state. After the execution of the current state the machines manager invokes the getNextState() method of the machine's transition table which based on the current values of the parameters and the current state determines which is the next state to be transitioned. The manager loops again and invokes the performComputation method for the newly transitioned state unless the state is not final.

To make it more clear, we will include a sample template of how the performComputation method could look like:

```
void performComputation(List<IParameter>
  params,int currentState){
      switch(currentState){
       case: 1
           {//code for state 1}
           break;
       case: 2
           {//code for state 1}
           break;
       …..
       case: N
           {//code for state N}
           break;
       default:
           throw new Excetion
             ("Invalid state. Design time error")
      }
  }
```

It becomes obvious that the business logic of an application will reside in the performComputation method, in the parameter classes as well as in the transition table. This is why the application developer will be needed to derive from the StateMachine class and implement the performComputation method for every type of state machine.

The TranzitionTable class is nothing more than a mapping between <parameter values, current state> tuples and future states. The business logic in case of the TransitionTable class resides in the values it contains rather than in its implementation.

The parameter classes will implement the IParameter interface which exposes the *equals* method which is required to match the values of two parameters and return *true* if they are considered to be equal or *false* otherwise. The *equals* method's implementation might range from very simple equalities to very complex checkings against custom objects, depending on the business logic requirements.

That being said, we can resume the applications implementation requirements to three steps:

1. Implement the parameters classes
2. Implement a state machine class for each type of state machines needed by the application, by extending the StateMachine class
3. Define transition tables.

Here is where the main drawback of the system appears. Since the framework has been developed using a static language (C#) the newly developed state machine class needs to be available on all nodes in order for that node to be able to run state machines of that type. So we can easily identify three important drawbacks at this stage:

- Updating a running application may not be an easy task due to security restrictions.
- All state machines need to comply to one of the classes, which avoids building and executing custom state machines on-the-fly
- The development time increases by the fact that any change in code needs to be spread across the distributed environment in order to be well tested. Also if one needs to run a certain state machine only few times, the development time might exceed the usage time for that state machine.

IV. Dynamic Execution Model

To overcome the drawbacks identified in the previous section we have considered the use of dynamic languages. Instead of implementing classes every time we needed a new type of state machines, we can now include the code as string on the state machine object itself and invoke it dynamically when needed. This raises an important controversy specific to dynamic languages, namely code safety. As the code is not compiled prior to running it, it might contain erroneous code (typos, references to undefined variables or methods, etc) which can damage the execution Besides trying to overcome unsafe code at design time by considering all states that a machine can step into prior to writing the code, we could make use of an adequate development strategy like test driven development and we can increase code safety. However it is well known that dynamic programming requires a lot more attention when writing code than static programming. Assuming that we need to pay attention when writing code, and tediously test it before running it in production, this tradeoff gives us a lot of flexibility.

We can now run as many custom state machines as we want without having to implement classes and move code files across the running system, while reducing the development effort.

Extending the StateMachine class for each type of state machines is no longer needed. We simply make use of the StateMachine class and add a new attribute called *codeMapping* which will map states to their corresponding code represented as strings.

For example, in Ruby Language, the *codeMapping* attribute could be of type Hash and have as keys the state numbers, and as values string containing the code that needs to be run for the associated state:

```
codeMapping = {
   1 => "puts 'code for state 1'",
   2 => "puts 'code for state 2'",
   # …..
   N => "puts 'code for state N'",
}
```

In this context, the state machines manager, would simply invoke the code defined for the current state.
So, instead of calling:

```
performComputation(params, state)
```

it will simply invoke the code dynamically:

```
eval(machine.codeMapping[state])
```

At this stage, our state machines have become abstract collections of states which are nothing but objects that combine code and data, and define some kind of ordering between them resulted from the transition table.

Each state is very similar to a function which takes input parameters and input data, executes some custom code against them and outputs transition parameters and data needed by other states, and so on. If we were to represent the structure of states machines based on their states, data and transition table, it would look like a graph. The nodes of the graph would be the states or data objects, and the arrows would be the dependencies between states constrained by the value of the transition parameters (according to the transition table). Taking it further, we can represent all state machines running at a certain moment and we can conclude that we're dealing with a graph of states and data objects with dependencies between them and we no longer care to which state machine they belong to. Such a graph is illustrated in Fig. 4.

The figure includes the states of two finite state machines (S_{1i} for the former and S_{2i} for the latter) and their input/output data objects (d_i). The transitions between states are being conditioned by the machine's parameters (P) values. For example, machine 1 will move from state S_{10} to state S_{12} only if the value of P_1 matches the value on the arrow (relationship) between S_{10} and S_{12}, which is V_{12}. Each state can be launched only if all data dependencies are satisfied, and the state is considered to be executed completely only after all output data has been delivered. In case a state needs as input data, another state's output, the former state can't be launched unless the latter has been completed. The execution relies on the *execution pointers*. These are nodes in the graph and they point out the last executed state of each state machine. As the machine traverses throughout it's states the corresponding pointer will move to the new state.

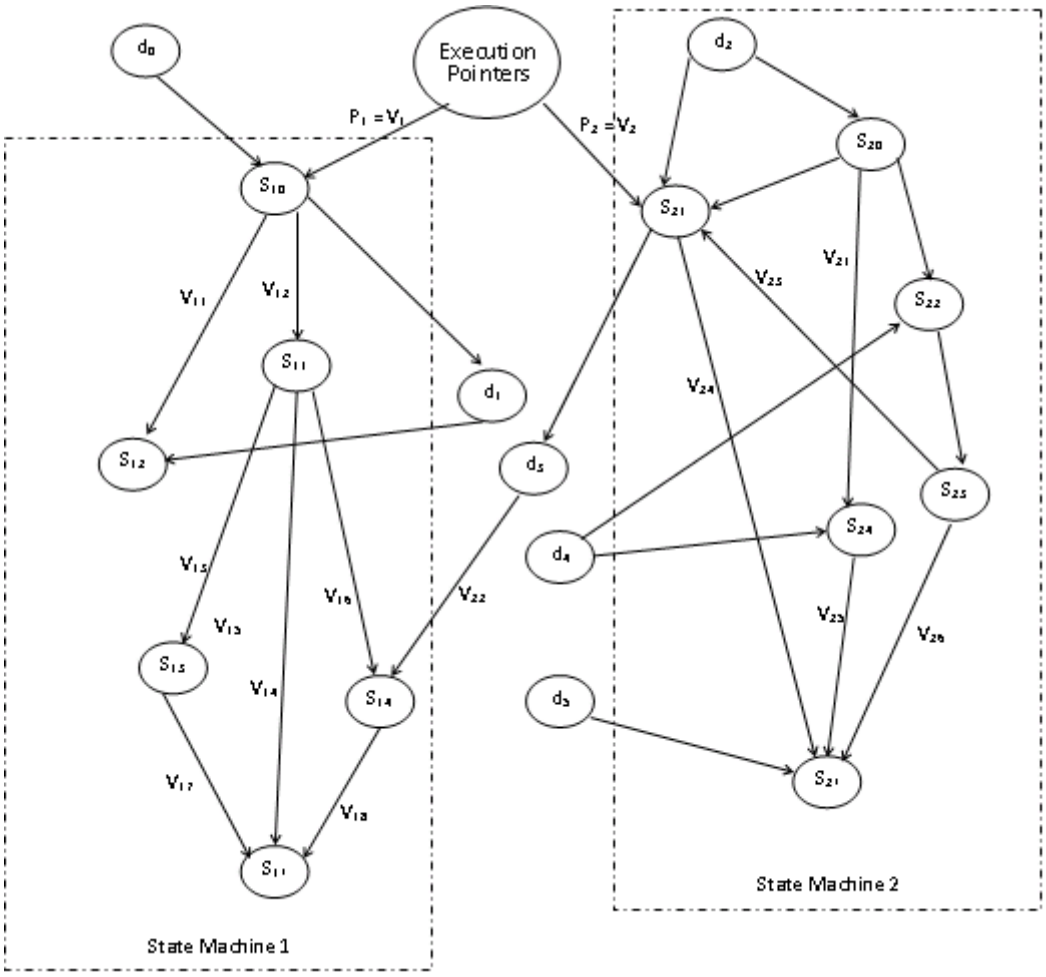

Fig. 4 Execution graph.

It becomes obvious that the execution graph has to be shared by all nodes (execution managers). Normally this would be an important loss for our architecture since the old architecture was not centralized. Keeping the graph in the shared memory area would not be very efficient since it has a distributed architecture and it would require important amount of synchronization communication which would slow down the system considerably.

Fortunately, graph databases and their spectacular evolution have allowed us to benefit from a very powerful infrastructure for distributed environments which has been discussed in section II.

Considering the new execution model we can easily observe that the simulation module of the old architecture no longer handles state machines entirely but states of machines.

For this reason the execution manager available on each host will be responsible for acquiring states that can be immediately executed, run them, save the output and then handle another state.

The algorithm can be resumed by presenting the approximate structure of the class that handles the database (GraphDB), and the main loop that keeps the engine running until the execution is finished.

```
class GraphDB

def exist_final_states
 execution_pointers.out.each do |state|
  return false if not state.is_final?
 end
 return true
end

def pick
 execution_pointers.out.out.filter
       {P = V && in(data).available}
end


def save_output(state)
 database.save(state.output_data)
 exec_pointer= state.in.in(exec_pointer)
 exec_pointer.save_params
    (state.output_params)
 exec_pointer.out = state
end

end
```

The exist_final_states method checks if all final states have been reached, in which moment the execution should stop, by traversing the graph starting from the execution pointer nodes, following the outgoing relationships and returning true if all states pointed by the execution pointers are final states, otherwise it returns false.

The pick method starts from all the execution_pointers, traverses down the tree through the last completed states, down to the next state by checking the current parameters' values and returns the first state found. The method can be easily adjusted to return more than one state depending on the load balancing policy.

The save_output method, saves the output data objects resulted after processing the current state into the graph, identifies the corresponding execution pointer and moves its position to the currently completed state.

The main loop run on every process (simulation process), is similar to the following piece of code:

```
while graphDB.exist_final_states
  current_state = graph_db.pick()
  //dynamically run the Ruby code
  eval(current_state.code)
  graph_db.save_output(current_state)
end
```

V. Conclusions and Future Work

The proposed architecture brings more flexibility to our distributed framework by allowing the developer to write dynamic code and at the same time reduces the development effort by not requiring the implementation of static code for each type of custom state machine. As a consequence the update process of a running application no longer requires sending code files across the distributed environment, but simply running finite state machines with embedded custom dynamic code.

The new graph execution model breaks the finite state machines into independent states and handles them all together. As new state machines arrive, their states get appended to the graph and dynamically allocated to computational resources (through each host's execution manager).

The new approach ensure fair load balancing by having the nodes acquire the processing tasks (machines' states) as their resources become available as opposed to static allocation (prior to launching the application) or centralized dynamic allocation algorithms.

Reducing the amount of work required by a task at the minimum of only one state also improves the load balancing.

Our near future research thoughts refer, to evaluating the performance of the new architecture and identify the performance gain as well as the weaknesses of the new platform architecture.

Dynamic load balancing algorithms have always been an important lead for us and they will be given special concern in the near future.

Data distribution plays a very important role when we deal with load balancing algorithms in distributed environments, as it is more efficient to move the processing towards data than the other way around, therefore special interest will be shown on this issue.

Creating tools for better monitoring and steering of the distributed environment, of both resources and execution is also on our to-do list.

REFERENCES

[1] W. Gu, J. Vetter and K. Schwann. An annotated Bibliography of Interactive Program Steering, SIGPLAN Notices 29 (1994), pp. 140-148 and Technical Report GIT-CC-94-15 (Georgia Institute of Technology)

[2] R.J. Allan and M. Ashworth. A Survey of Distributed Computing, Computational Grid, Meta-computing and Network Information Tools, available from http://www.ukhec.ac.uk/publications/reports/survey.pdf

[3] Morris Riedel, Wolfgang Frings, Sonja Habbinga, Thomas Eickermann, Daniel Mallmann, Achim Streit, Felix Wolf, Thomas Lippert, Andreas Ernst, Rainer Spurzem: Extending the collaborative online visualization and steering framework for computational Grids with attribute-based authorization. GRID 2008: 104-111

[4] S. Jha, S. Pickles, and A. Porter. A Computational Steering API for Scientific Grid Applications: Design, Implementation and Lessons. In Workshop on Grid Application Programming Interfaces, Brussels, Belgium, Sept. 2004.

[5] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning and A. R. Porter, Computational Steering in RealityGrid, Proceedings of the UK e-Science All Hands Meeting, September 2-4, 2003

[6] J. A. Kohl and P. M. Papadopoulos. Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS. In 2nd SIGMETRICS Symposium on Parallel and Distributed Tools, Welches, OR, Aug. 1998.

[7] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. Intl. Journal of High Performance Computing Applications, 11(3):224-236, Aug. 1997.

[8] J.J. van Wijk and R. van Liere. An environment for computational steering. In G.M. Nielson, H. M¨uller, and H. Hagen, editors, Scientific Visualization: Overviews, Methodologies, and Techniques, pages 89–110. Computer Society Press, 1997.

[9] R. van Liere, J.D. Mulder, and J.J. van Wijk. Computational steering. Future Generation Computer Systems, 12(5):441–450, April 1997.

[10] David Harel, Michal Politi - Modeling Reactive Systems with Statecharts: The Statemate Approach, McGraw-Hill, Inc. New York, 1998, ISBN:0070262055

[11] Cosmin M. Poteras, Mihai L. Mocanu - A State Machine-Based Parallel Paradigm Applied in the Design of a Visualization and Steering Framework, Recent Researches in Applied Informatics, Proceedings of the 2nd International conference on Applied Informatics and Computing Theory (AICT '11), ISBN : 978-1-61804-034-3, pp232-236, WSEAS, Prague, Czech Republic, September 26-28, 2011

[12] www.tinkerpop.com

[13] Rodrigues, M.A., Broecheler, M., "Titan: The Rise of Big Graph Data", Public Lecture at Jive Software, Palo Alto, 2012

[14] Broecheler, M., LaRocque, D., Rodrigues, M.A., "Titan: A Highly Scalabe, Distributed Graph Database", GraphLab Workshop 2012, San Francisco, 2012