Software architecture of the challenge propagation model

Viktoras Veitas (vveitas@gmail.com)
The Global Brain Institute, Brussels

Revision 1.1 from December 31, 2012

Revision History

Revision	Date	Author(s)	Description
1.0	November 29, 2012	vveitas	created
1.1	December 31, 2012	vveitas	minor updates and corrections

1 Introduction

This paper is a description of the software architecture of the computer simulation of the challenge propagation model, which I have codenamed *Challprop*. Software architecture is based on the conceptual model outlined in Heylighen [2012] and the mathematical model developed in Heylighen et al. [2012].

2 Special requirements for the computer simulation

I focus on two special requirements for the simulation model:

Ability to observe global and local behaviour ("micro/macro-scope"). This means that we should be able to observe in real time (i.e. during the simulation) or subsequently analyze every chosen component of the network. A component of the network could be a challenge, a situation, an agent, a link, or a combination of these. From the software architecture point of view this implies that we should have the ability to access each component of the model as a separate object.

Ability to observe and analyze the dynamics of the network development ("life-logging"). Running the simulation with certain parameters for the predeter-

mined number of steps and then analyzing the statistics of the resulting network (along with the recorded dynamics of certain variables) may not provide sufficient enough data "granularity" for the analysis. The ideal case would be to have the ability to record all the "elementary" events of the simulation (process of the challenge, change of the benefit, update of link weight, etc.).

Such "micro/macro-scope" and "life-logging" of the simulation would allow, e.g. to "replay" the same simulation with different parameters, to analyze the propagation paths of specific challenges and to observe "life" of chosen agents in the network. Note that due to the probabilistic nature of the propagation and nonlinear effects, re-playing the same simulation from the "life-log" and re-running the simulation with exactly the same parameters may lead to completely different results.

Other requirements such as modularity, the ability to change each and every parameter of the simulation and the ability to collect certain statistical measures are self-evident and need no further explanation.

3 Software architecture

On the abstract level I define the simulation model as a collection of processes affecting a collection of objects. Processes and objects are intrinsically connected and can be viewed as different perspectives of the whole. Processes and their combinations describe how the model behaves dynamically, while objects provide access to a static picture, or "snapshot" of the model at a given point in time. I describe objects first and then connect them to processes because to our "object-oriented" minds this seems to be a more natural way of thinking.

3.1 Objects

From the"static" point of view, the challenge propagation framework is a graph, i.e., a collection of vertexes connected by edges. I use a property graph model to represent the model, which allows the properties of each vertex and an edge in the graph to be specified. The vertexes of the graph represent objects in the challenge propagation model, while edges represent relations between objects. Every object type features a list of certain properties and they are defined values for each instantiated object. Currently, the following **object types** and their properties are defined:

1. *Processor* is a function for processing a challenge. It is a place in the "topological space" of the graph where certain challenges get processed. Processor has these explicit properties:
    - *needVector*[1]

[1] Heylighen et al. [2012], section 4.1.4 Need vectors (page 7)

- *processingMatrix*[2]
- *agentFitness*[3]

2. A *challenge* is a piece of information defined as "a difference that makes a difference" (Heylighen and Joslyn [2001], page 6) with respect to a particular agent in the network. Properties of a challenge are:
   - A fixed-length *vector* of real numbers. The length of the vector is a global variable which represents "dimensionality of the model space"[4];
   - *Status* represents the state of the challenge in the propagation process and can take one of the values $\{non-interpreted, interpreted, processed\}$ (see Figure 1 and descriptions of *interpretation* and *reinterpretation* processes below).
   - Each component of the vector can be "rival" ("material", subject to the conservation law) or "non-rival" ("informational", non-conserved)[5]. The property *rivalComponents* is a set of numbers indicating which dimensions of the challenge are rival. All the others are non-rival.
   - A *source* of the challenge is an identifier of the last agent which processed it. E.g., if a challenge is propagated from agent A to agent B it's source is A, but if it gets propagated to agent C, it's source becomes B. This information is needed for learning process.
3. A *buffer*, which is connected to one and only one *processor*. Each *processor* should have a *buffer* connected to an agent (see Figure 1).
4. *"God" (generator of diversity)*. The generator of diversity creates challenges and delivers them to agents in the network. Therefore, a "work-flow" of challenge propagation starts in *god*, an is therefore the primary source of all challenges. Properties of *god* are:
   - A *distribution* of challenge vector's components, which has its own parameters such as interval size, power law exponent, thresholds, etc. [6];
   - The *delivery mechanism*, which specifies which challenges get delivered to which agents in the network (e.g., all challenges get delivered to all agents or scattered randomly to a fixed number of agents);
   - The *rate* of challenge generation. This parameter will determine how many new challenges are inputted to the network at each time step. This rate can be constant or variable.
5. *Sink*. All challenges which for one reason or another are not processed by the agent

[2] Heylighen et al. [2012], section 4.2.4 Processing (page 10)
[3] Heylighen et al. [2012], section 4.2.10 Agent Fitness (page 12)
[4] Heylighen et al. [2012], section 4.1.1 Vector representation (page 6)
[5] Heylighen et al. [2012], section 4.1.3 Rival and non-rival components (page 7)
[6] Heylighen et al. [2012], section 4.1.7 Generation of vectors (page 8)

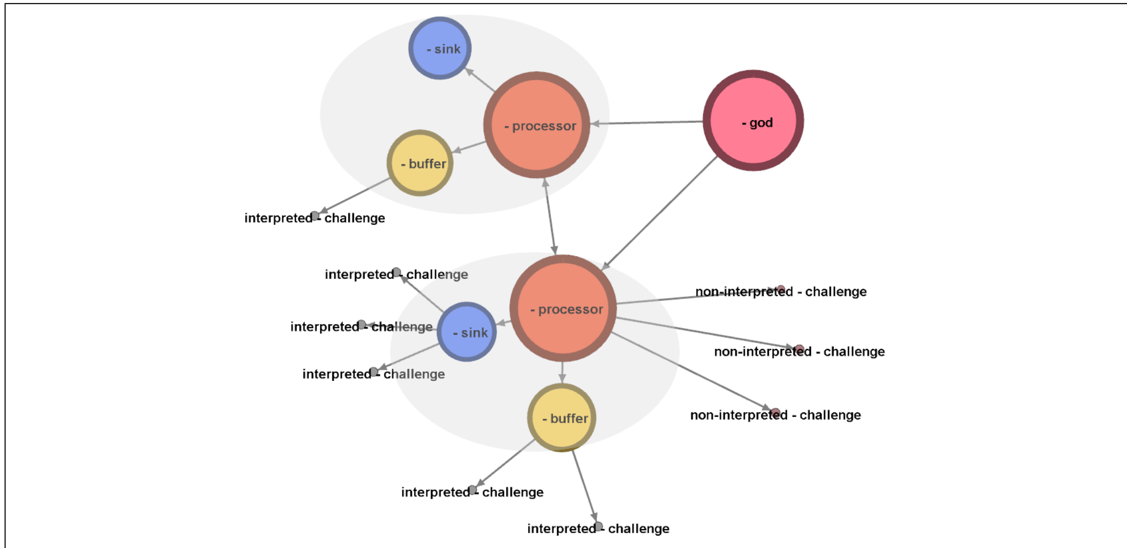

Figure 1: The 'snapshot' of a toy challenge propagation network with two agents and 9 challenges. Agents are marked as grey ellipses containing *processor*, *sink* and *buffer* objects connected between each other. Note that agents are not explicit objects in the underlying graph (grey ellipses are just a visual aid). There are 3 interpreted challenges in the sink of the lower agent, two challenges in the buffer and 3 challenges just received (delivered by *god*) and not interpreted yet. The uninterpreted challenges are also called *situations*[7]

go into its sink. Three objects, *processor*, *buffer* and a *sink*, connected together, constitute an agent (see Figure 1).

Objects in the graph are linked with *edges*. The property graph model allows edges in the graph to have one semantic label (type) and an unlimited number of properties. So we define the following link types in the model (indicated by green italic text in Figure 1) as:

- *isAssociated*. Links of this type are used to connect *god* to *processors* and *processors* among themselves. Links have the *weight* property, which indicates the strength of the link. In the current challenge propagation model, link weights will play a major role in self-organization and learning[8]. The exact mechanism of changing link weights will be encoded in the propagation processes (see below). Note that links between *processors* correspond to links between agents in the challenge propagation model.
- *has*. Links of this type connect *processors* to *buffers* or *sinks*. They can be interpreted as internal relations between the components of a single agent - *processor*, *buffer* and *sink*. Such a relation can be read as, e.g., “processor *has* buffer”. Such a mechanism for constructing the “internal” structure of an agent will allow other components (and probably functionality) to be added to agents in the future.
- *holds* link types relates *challenges* and all other objects. The object to which a *challenge* is linked identifies the state of propagation of the challenge:

[8] Heylighen et al. [2012], section 4.3.9 Reinforcement of links (page 16)

- a *challenge* linked to a *processor* (relation "processor *holds* challenge") means that a challenge has been received from the *god* and not yet pre-processed. In a sense, an agent has not yet "decided" what to do with the challenge: process, put to a buffer or send to a sink;
- a *challenge* linked to *buffer* (relation "buffer *holds* challenge") means that the challenge is remembered for future processing;
- a *challenge* linked to *sink* (relation "sink *holds* challenge") means that this particular challenge was not considered worthy for processing.

3.2 Processes

From the "dynamic" point of view, a challenge propagation network consists of collections of ordered processes. A process in this context is anything that changes the state of the *challenge*. Each *challenge*, from the moment of it is generated till the moment it is relaxed gets "routed" through a number of processes which in turn are related to certain objects in the network. For example, a challenge may be:

(a) delivered to *processor* **A**,
(b) interpreted with respect to the *needVector* of **A**,
(c) selected for sorting in the *buffer* connected to **A**,
(d) selected as having the highest intensity from the buffer,
(e) processed by *processor* **A**,
(f) reinterpreted against the *needVector* of **A** (converted to a *situation*),
(g) propagated to an agent **B**,
(h) interpreted against the *needVector* of **B**,
(i) selected for sending to the *sink* connected to **B**.

The path of this particular challenge in the network could be written as $a \to b \to c \to d \to e \to f \to g \to h$.

The actual path may be much longer and cannot be determined a priori (i.e. before running the simulation). Each challenge will "construct" its own path in the network (starting with the process of generation in *god* and ending in *sink* or anywhere in the network). Multiplicity of these paths will determine the results of a simulation run. Such a multiplicity of paths could also be viewed as a *work-flow* (Heylighen and Joslyn [2001], page 10). Therefore, the overall goal of the simulation modelling, as I see it, is *the search for a work-flow which would show the improvement of the network's information processing capabilities with respect to a certain global criteria* (e.g., efficiency of challenge relaxation, maximum extracted average benefit per time, "global IQ" measure, etc.). Note that while possible internal paths ("within" the agent, or grey areas in Figure 1) can be unambiguously defined in advance, propagation paths between agents in the network cannot. "Construction" of propagation paths between agents during simulation will depend on respective link weights, which, in turn, will depend on learning from past

propagation and processing of other challenges during the same simulation run.

3.2.1 Flowchart of the challenge propagation model

Figure 2, below, graphically depicts the flow of processes, data and control in the challenge propagation model.

Figure 2: Flowchart of the challenge propagation model. Yellow rectangles represent processes of the challenge propagation; dark arrows with solid circles on tails - the flow of challenge(s); grey rectangles - objects and their properties of the challenge propagation network; hollow circles and ovals - decision points related to respective processes; hollow trapezoids - global parameters of the simulation.

Graphical notations used in Figure 2:

- denotes an individual process of the challenge propagation model. connects them into a process flow which defines the overall dynamics of challenge propagation model.
- denotes control data needed for the processes. In terms of the individual simulation run, control data are global parameters of the simulation (e.g., algorithm for the generation of a challenge vector or processing matrix of an agent[9]). Global parameters cannot change during a particular simulation run.
- denotes a different kind of data needed for the processes - objects in the network and their properties (see section 3.1 for the description of objects). It is a major property of the challenge propagation model that processes can both read and write data from and to objects in a challenge propagation network.

[9] Busseniers [2012]

- ⇢ denotes flows of control and simulation data in and out of processes (i.e. input and output).

3.2.2 Descriptions of individual processes

What follows is the list of main processes defined in the challenge propagation model (Figure 2) and their detailed descriptions. Possibilities and limitations of defining challenge propagation as ordered collections of these processes are discussed later (section 5). Note that a mathematical model (Heylighen et al. [2012]) considers even more processes than those included in the current description. Nevertheless, the overall software architecture allows for easy inclusion and exclusion of individual processes and objects into and from the model.

1. *Generation.* Challenge generation is a process carried out by the *generator of diversity* object, which takes as input an algorithm for challenge generation and the distribution of challenge vector values[10]. Specifying the distribution of challenge vector values puts certain constraints on the overall structure of challenges. A structure of the initially generated challenges can be seen as the "environment" of the challenge propagation network. An interesting question, put forward in Zenil et al. [2012], and to be tested during the simulation modelling, is how much structure is needed to cause self-organization in the model.

2. Initial *propagation* (delivery). In challenge propagation network, the *generator of diversity* will be connected to all agents. *Propagation* process will ensure that all generated challenges are delivered to agents for processing and further propagation through the network. There are several options for initial propagation of challenges: individual challenge can be delivered to one agent, copied to several agents or, alternatively, to all agents in the network. These alternatives will be tested in the process of simulation modelling. Note that the same *propagation* process will be employed for propagation of challenges from one agent to another (this aspect is described below).

3. *Interpretation* and *Reinterpretation.* A challenge created by the *generator of diversity* is also called a *situation*[11]. From the point of implementation, challenge and situation are the same object. Rather, we distinguish the different status of the challenge object (see description of the *status* property above). We use the term "situation" for saying "a challenge with status "non-interpreted"". *Interpretation* process takes as input the interpretation function (which is currently defined as subtraction of *needVector* from situation's vector[12]). *Reinterpretation* is the inverse function of *Interpretation.*

4. *IntensityCalculation. Intensity* is a property of the link between a particular agent and a *challenge*, and is a measure of potential benefit which can be extracted from

[10] Heylighen et al. [2012], section 4.1.7 Generation of vectors (page 8)
[11] Heylighen et al. [2012], section 3.2 Flow of Activity (page 4)
[12] Heylighen et al. [2012], section 4.1.4 Need vectors (page 7)

that challenge (see description of the *Processing* process). More generally, intensity is a measure of valence of the particular challenge to a particular agent (Heylighen [2012], page 7). Currently, the function for intensity calculation is defined as the absolute sum of a challenge vector's components [13].

5. *Selection*. After interpretation of a challenge, it can either be selected for further processing and benefit extraction or rejected. *Selection* process implements the attention mechanism of an agent, which is responsible for the "selection for relevance" (Weinbaum [2012], page 5). The selection function can be defined as a threshold for challenge intensity (e.g., rejects challenges with negative intensity value).

6. *Rejecting*. If a *challenge* is rejected by the *Selection* process, it is pushed out of the propagation ant sent to the *Sink* of the respective agent (see Figure 1).

7. *Ordering*. If a *challenge* is selected for further processing, it is then added to the list of challenges in the *Buffer* object of the respective agent and the list is ordered by the intensity of each challenge. If the *Buffer* was full before adding the new challenge, then the challenge with the least intensity in the ordered list is rejected (see *SelectChallengeWithLowerIntensity* process). Note that this process implements a more advanced version of "selection for relevance" as described in Weinbaum [2012].

8. *UpdatingTheBuffer* process updates the buffer contents with the outcome of *Ordering* process.

9. *SelectChallengeWithLowestIntensity* and *SelectChallengeWithHighestIntensity*. The *Buffer* of an agent is an ordered stack with a challenge of the highest intensity at the top and a challenge with the lowest intensity at the bottom. The model may require either lowest or highest intensity challenge to be accessed during the process of propagation. Therefore, *SelectChallengeWithLowestIntensity* takes the challenge from the bottom of the *Buffer* and *SelectChallengeWithHighestIntensity* takes the challenge from the top.

10. *DecisionToProcess*. The flow of processes related to processing of challenges by the agent is "parallel" to most of the processes described above (see Figure 2). At each point in time, each agent in the network separately "decides" whether to process a challenge from the *Buffer* or do nothing (*skipTimeStep*). Each decision will depend on the globally defined *process synchronization mechanism*, which is currently simply "process one challenge every time step".

11. *Decay*. An *agentFitness* property is measure of the fitness of an agent in the network (Heylighen et al. [2012], section 4.2.10 Agent Fitness (page 12)). In order to implement the principle that the agent gains fitness by taking action and loses fitness by not taking action or both[14], we introduce the *Decay* process, which reduces the fitness of each agent in the network by equal an amount at each time

[13] Heylighen et al. [2012], section 4.1.5 Intensity (page 7)
[14] Heylighen [2012], section 3.2 Definition of challenge (page 7)

step of the simulation.

12. *Processing* starts the collection of the most important processes in the model. This process applies a *processingFunction* (globally defined parameter of the simulation) to the challenge vector. Currently, challenge processing is defined as a matrix transformation[15]. Note that each agent in the network (more precisely, *Processor* object) also has its individual *processingMatrix* generated by a globally defined matrix generation algorithm (Busseniers [2012]). The average result of this process will be a challenge with reduced intensity. This result splits to three final processes, described below.

13. *Propagation*. Processed challenge is reinterpreted (see above) and sent to the other agents in the network. This is actually the same propagation process described above as initial propagation, but the source of the challenge in this case is not the *generator of diversity* but another agent. Again, we have several options to define the propagation mechanism (which is a global variable of the simulation): challenges can be copied to all connected agents, can be passed only to the agent with the strongest link, or distributed probabilistically according to link weights.

14. *BenefitCalculation*. The benefit of an agent from processing a challenge is defined as the difference between intensity of incoming challenge and intensity of processed challenge[16]). *Processing*, *Propagation* and *BenefitCalculation* processes, combined together in an agent take a challenge with a certain intensity and 'splits' the intensity into two parts, one of which increases the fitness of an agent (i.e. *agentFitness*), and another is left for other agents to extract (i.e. left in the challenge vector).

15. *Learning*. Each *challenge* will keep the information about its propagation path (*source* property). Currently, this path consists of the single last travelled link. The *Learning* process reinforces the source link depending on the results of the *Processing* process: the more benefit is extracted from the respective challenge, the stronger its source link becomes[17]. Note that by regulating the length of the propagation path, "remembered" by the challenge, it is possible to simulate a "bucket brigade" algorithm for link strength revision (Holland et al. [1989], page 72). It is expected that the learning mechanism will be one of the main sources of self-organization dynamics resulting in stable network structures. Challenges and requirements for the analysis of the simulations' data are assessed in section 5 below.

[15] Heylighen et al. [2012], section 4.2.4 Processing (page 10)
[16] Heylighen et al. [2012], section 4.2.6 Calculation of Benefit (page 9)
[17] Heylighen et al. [2012], section 3.3 Global Dynamics (page 5) and section 4.3.9 Reinforcement of links (page 16)

4 Implementation with the Tinkerpop stack

The software for the simulation of the challenge propagation model will be implemented on the Tinkerpop graph stack, which is an ecosystem of the state-of-the-art Groovy and Java based open source tools for querying and manipulating Graph databases, databases optimized for working with Big data.

The simulation model was defined as a collection of processes working on a collection of objects (see sections 3.1 and 3.2). The process part of the model will be implemented using Pipes framework, which is the dataflow framework inspired by Kahn process networks (Rodriguez [2012]). There are indications that in certain cases a Kahn process network can be modelled using the Petri net formalism (Wikipedia [2012a]). In terms of Kahn process network, propagation paths of the challenges ("chains" or pipelines of processes) in the challenge propagation network would constitute a process graph.

For example, the *Interpretation* process, described above, can be implemented with this code:

```
class Interpretation implements PipeFunction<FramedAgent, FramedAgent> {

 private final String function;

 public Interpretation(String function) {
  this.function = function;
 }

 public FramedAgent compute(FramedAgent agent) {
  if (function == "subtraction") {
        String need = agent.getNeedVector();
        Iterable<FramedChallenge> challenges = agent.getNonInterpretedChallenges();
        int agentID = agent.getID()
        challenges.each {
         FramedChallenge challenge = it;
         RealVector realNeed = Utils.deserializeVector(need);
         RealVector realChallengeVector = Utils.deserializeVector(challenge.getVector());
         RealVector interpretation = Utils.subtractChallenges(realChallengeVector, realNeed);
         challenge.setVector(Utils.serializeVector(realChallengeVector));
         challenge.setStatus("interpreted")
        }
  }
  return agent;
 }
}
```

The process then can be put into a context of *SideEffectPipe*:

```
Pipe interpretationPipe = new SideEffectFunctionPipe(new Interpretation("subtraction"))
```

If *interpretationPipe* is inputted, a list of references to agents in a challenge propagation network, it reads data about all non-interpreted challenges attached to each agent, interprets them, writes data back to the network and outputs the same list.

Several processes then can be joined into a *pipeline* thus creating a complete work-flow:

```
Pipeline pipeline =  new Pipeline(interpretationPipe,
                                  intensityCalculationPipe,
                                  selectionPipe,
                                  orderingPipe)
pipeline.setStarts(challengedAgents)
```

The collection of objects will be implemented as a Blueprints-enabled graph. Blueprints is a Java library for interfacing with the property graph model. As noted in Section 3.1, the collection of challenge propagation model's objects will be implemented as a property graph. Making this graph Blueprints-enabled will allow to work with it using Gremlin graph traversal language, which enables "visiting" vertexes and edges of a graph algorithmically (Rodriguez and Neubauer [2010], page 7).

The challenge propagation model requires certain processing of data (e.g. matrix transformations) which cannot be conveniently implemented using Gremlin. For this kind of processing we shall use custom classes written in Java and Groovy languages (or anything else that can be efficiently integrated into the whole infrastructure). The required interface between processes and objects in the graph will be constructed using Frames, another member of the Tinkerpop stack, which enables any component of a property graph to be exposed as a Java object (including agent, challenge, buffer and any other object defined in section 3.1).

The propagation of challenges, implemented as nodes in the graph, will be achieved by graph rewriting. That is, the process of propagating a challenge from one agent to another will simply amount to deleting the link between challenge and the first agent and creating a link to a new one.

The infrastructure outlined above will allow special requirements for computer simulation to be achieved ("micro/macro-scope" and "life-logging", see section 2), but will not automatically provide them. Basic functionality of the "micro/macro-scope" can be achieved by querying the graph directly using Gremlin console in a text-based mode. Other considerations regarding special requirements are discussed in the next section.

5 Further questions

5.1 "Life-logging"

Using the infrastructure outlined in section 4 challenge propagation model's simulation will become a "meta-process" of building chains of lower level processes. It seems, that the complete history of this meta-process would be a sort of process graph, but for an unequivocal description a more formal analysis is needed. Informally, I refer to the complete history of the simulation as a *life-log*. There are at least two alternatives for achieving "life-logging" within the current infrastructure of the challenge propagation model:

1. *Time-aware graph database.* Blueprints allows for the use of different graph databases

("backends") for storing a property graph. One of the options is FluxGraph, a temporal graph database on top of the time aware database Datomic. A time aware graph database, in addition to storing a graph, also stores information about all changes of the graph, together with the timestamp of a change. For an example of time-aware graph application see Boldi et al. [2008]. The usage of a time aware database would enable a "snapshot" of stored graph to be extracted at any time within the simulation or would enable the history of graph development to be reconstructed (perform what is called "a time travel"). Furthermore, as FluxGraph is a graph database that should allow for "temporal graph traversals" (i.e. traversing the propagation path of a single challenge). The only drawback I can see is that Datomic is not a fully open source product. Nevertheless, FluxGraph seems like an ideal option for the "life-logging", but the product is fairly recent (about one month at the time of the writing); therefore, some testing and trying is needed;

2. *Event logging.* A somewhat less elegant, but perfectly viable option is the standard event logging. For example, all events on a graph could be logged to an XML file using OpenXES, an implementation of XES standard for event logs. An advantage of this approach is that the "life-logging" system would be completely separated from the graph database, making overall software architecture more modular. On the other hand, in order to "traverse" these logs as a graph, certain processing (probably non-trivial) will be needed. There are interesting tools and approaches for converting process logs into Petri nets and/or perform process mining (e.g. ProM6, a process mining workbench).

5.2 Petri nets

The association between Kahn process networks and Petri nets brings up the possibility of analyzing the challenge propagation model's simulation using Petri net formalism and a large base of tools and formal methods related to Petri nets. Kahn process networks are associated with a dataflow framework, which is used to implement processes in the simulation (section 4). Petri net formalism was invented in 1939 and is probably the oldest and most developed formalism for description of processes, which in addition to a graphical notation, offers "an exact mathematical definition of the execution semantics and a well-developed mathematical theory for process analysis" (Wikipedia [2012b]). The extent to which Petri net formalism is able to describe the challenge propagation model is not obvious, but finding links between the two may be very useful. The relation between the challenge propagation model and Petri nets would be even more interesting when taking into account the plans to integrate the chemical organization theory into the model[18] and the fact, that chemical reaction networks are formally equivalent to Petri nets (Dittrich [2009], page 328).

Figure 3 on page 13 is the illustration of an alternative representation of the challenge propagation model as a Petri net.

[18] Heylighen et al. [2012], section 4.2.5. Correspondence with chemical organization theory (page 10)

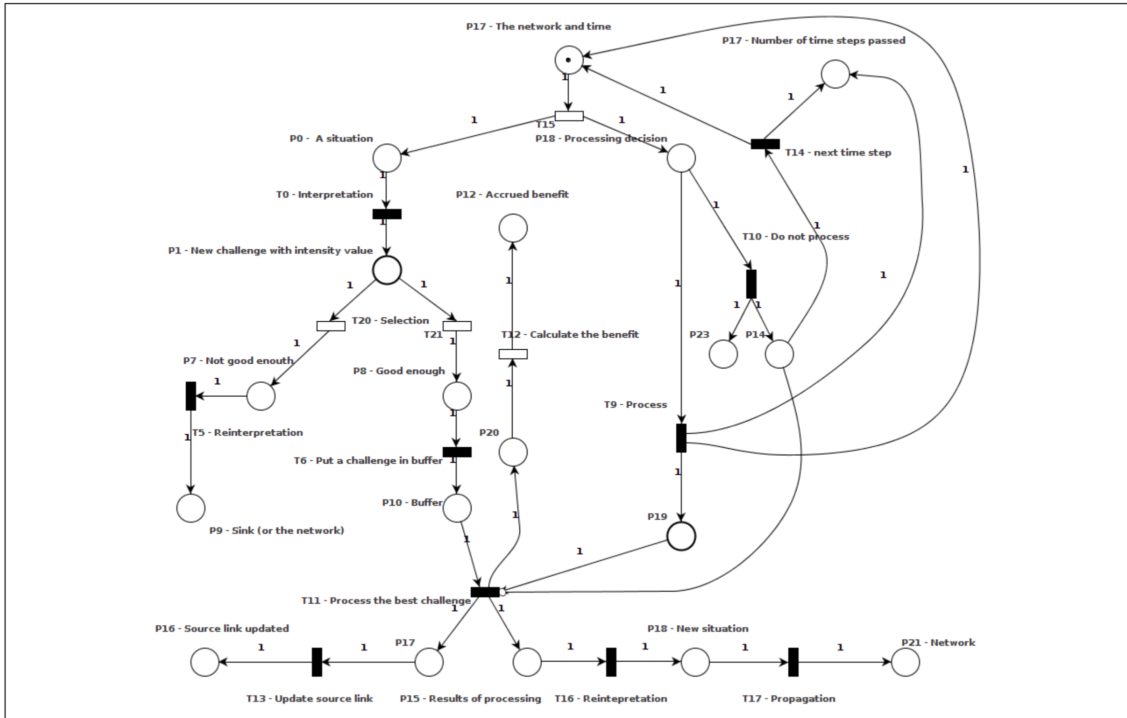

Figure 3: A Petri net describing internal processes of an agent. Circles denote objects (*processor*, *buffer*, *sink*), states ("buffer is not full") or have no meaning at all. Bars mostly denote transitions in Petri net terminology and processes in the terminology of the challenge propagation model. Tokens can be interpreted as challenges or time steps, depending on their place in the net.

5.3 Parallelism

Parallelism of processes is a desirable feature that would allow more realistic dynamics to be expected from the simulation model. Parallelism may possibly be achieved using these tools (or combining them):

- *Gpars* or Groovy Parallel Systems is a Groovy concurrency library which is an "open-source concurrency and parallelism library for Java and Groovy that gives you a number of high-level abstractions for writing concurrent and parallel code"[19]. One of the possible advantages of GPars is that it is written in the same language as the Tinkerpop stack - Java / Groovy.
- *Actor model* of concurrency using Akka Java API as advocated in Wampler [2011], page 41. This model and library may integrate well with Pipes framework, but it is only my general impression from Wampler [2011], ant it should be checked.
- *OpenCL* and *CUDA* are languages for writing programs which execute on graphics processing units (GPUs). GPU computing is an increasingly popular way of powering simulations of massively multi-agent systems. In principle, a code written in these languages can be integrated into the whole simulation infrastructure with

[19]The GPars Project - Reference Documentation, Introduction

the help of libraries like JCuda.

Parallelism is not seen as an immediate priority of the simulation model, but it is useful to foresee the future possibility of extending the architecture into this direction.

5.4 Other issues

Some of the other issues to be considered are:

(a) *Computational complexity* of the model and related performance of proposed software architecture. The preliminary scale of a challenge propagation model is:

- number of agents: 10^4;
- number of links per agent 10^2, so number of links in the network: 10^6;
- very sparse challenge vectors: 10^2 dimensions with 3-5 non zero dimensions;
- assuming that at each time step 10^2 new challenges will be generated by *generator of diversity* (amounting to 10% of agents in the network), after 10^5 time steps there will be 10^7 objects in the network (so, nodes in the underlying graph).

The complete list of all free parameters of the model and their ranges is given in Heylighen et al. [2012], Table 1 (page 20).

The above scale is no cause for concern given capabilities of the Titan graph database, for example[20]. Yet I have to admit that I have not at all investigated this question and do not really know how to approach it in a decent way, considering the required processing and “life-logging” functionality of the model.

(b) *“Hypothesis testing machine”*. The model will have quite a big parameter space, including not only numerical parameters, such as the number of dimensions of a challenge vector, but also functions and algorithms (see Figure 2). Simulation modelling is a way of searching this parameter space by testing hypotheses about optimal values and combinations of the parameters. I am sure that such hypothesis testing will have to be performed semi-automatically, probably employing automatic testing and automatic debugging techniques. Such a framework will require another layer of software to be built on top of the architecture described in this paper.

References

Boldi, P., Santini, M., and Vigna, S. (2008). A large time aware web graph. *SIGIR Forum*, 42(2):33–38. http://vigna.dsi.unimi.it/ftp/papers/TimeAwareGraph.pdf.

Busseniers, E. (2012). Construction of processmatrix. *GBI internal memo 2012-10*.

[20] Aurelius LLC. (2012). Titan: a highly scalable, distributed graph database

Dittrich, P. (2009). Artificial chemistry. http://www.springerlink.com/index/10.1007/s11538-006-9130-8.

Heylighen, F. (2012). Challenge propagation: a new paradigm for modeling distributed intelligence. *GBI working paper 2012-01*. http://pespmc1.vub.ac.be/Papers/ChallengePropagation.pdf.

Heylighen, F., Busseniers, E., Veitas, V., Vidal, C., and Weinbaum, D. (2012). Towards a mathematical model of the global brain: architecture, components, and specifications. *GBI Working Paper 2012-05*. http://pespmc1.vub.ac.be/Papers/TowardsGB-model.pdf.

Heylighen, F. and Joslyn, C. (2001). Cybernetics and second-order cybernetics. In *Encyclopedia of Physical Science & Technology (3 rd*. Academic Press.

Holland, J. H., Holyoak, K. J., Nisbett, R. E., and Thagard, P. (1989). *Induction: Processes of Inference, Learning, and Discovery*. A Bradford Book, reprint edition.

Rodriguez, M. A. (2012). On the nature ofpipes. *Marko A. Rodriguez Web Page*. http://markorodriguez.com/2011/08/03/on-the-nature-of-pipes/.

Rodriguez, M. A. and Neubauer, P. (2010). The graph traversal pattern. *arXiv:1004.1001*. http://arxiv.org/abs/1004.1001.

Wampler, D. (2011). *Functional Programming for Java Developers: Tools for Better Concurrency, Abstraction, and Agility*. O'Reilly Media.

Weinbaum, D. (2012). A framework for scalable cognition: Propagation of challenges, towards the implementation of global brain models. *GBI working paper 2012-02*. http://pespmc1.vub.ac.be/ECCO/ECCO-Papers/Weaver-Attention.pdf.

Wikipedia (2012a). Kahn process networks. *Wikipedia, the free encyclopedia*. http://en.wikipedia.org/w/index.php?title=Kahn_process_networks&oldid=508781798.

Wikipedia (2012b). Petri net. *Wikipedia, the free encyclopedia*. http://en.wikipedia.org/w/index.php?title=Petri_net&oldid=521950824.

Zenil, H., Gershenson, C., Marshall, J., and Rosenblueth, D. (2012). Life as thermodynamic evidence of algorithmic structure in natural environments. *Entropy*, 14(11):2173–2191. http://www.mdpi.com/1099-4300/14/11/2173.