The Implementation of Metagraph Agents Based on Functional Reactive Programming

Valeriy Chernenkiy, Yuriy Gapanyuk, Anatoly Nardid, Nikita Todosiev
Bauman Moscow State Technical University,
Moscow, Russia
chernen@bmstu.ru, gapyu@bmstu.ru, nazgull09@gmail.com, todosievnik@gmail.com

Abstract—The main ideas of the metagraph data model and the metagraph agent model are discussed. The metagraph approach for semantic complex event processing is presented. The metagraph agent implementation based on Functional Reactive Programming is proposed.

I. Introduction

Currently, models based on complex networks are increasingly used in various fields of science, from mathematics and computer science to biology and sociology.

According to [1]: "a complex network is a graph (network) with non-trivial topological features – features that do not occur in simple networks such as lattices or random graphs but often occur in graphs modeling of real systems." The terms "complex network" and "complex graph" are often used synonymously. According to [2]: "the term 'complex network,' or simply 'network,' usually refers to real systems while the term 'graph' is generally considered as the mathematical representation of a network." In this article, we also acknowledge these terms synonymously.

One of the most important types of such models is "complex networks with emergence." The term "emergence" is used in general system theory. The emergent element means a whole that cannot be separated into its component parts. As far as the authors know, currently, there are two "complex networks with emergence" models that exist: hypernetworks and metagraphs.

The hypernetwork model [3] is mature, and it helps to understand many aspects of complex networks with an emergence. It is the hypernetwork model used by Professor Konstantin Anokhin to build a brain model based on a cognitome approach [4].

It is now essential to offer not only a model that is capable of storing and processing Big Data but a model that is capable of handling the complexity of data. The article [5] discusses in detail the advantages of the metagraph model in comparison with the hypergraph and hypernetwork models. From the authors' point of view, the metagraph model is more flexible and convenient than a hypergraph and hypernetwork model for use in information systems.

The metagraph model data processing is based on a multi-agent approach. However, the issue of the effective software implementation of metagraph agents is still open. Currently, there is no doubt that it is the functional approach in programming that makes it possible to make such implementation effectively. Therefore, the work is devoted to the implementation of a multi-agent paradigm using functional reactive programming. The advantage of this implementation is that agents can work in parallel, supporting a functional reactive paradigm.

The article is organized as follows. The section II discusses the main ideas of the metagraph model. The section III discusses the metagraph agent model. The section IV discusses the Semantic Complex Event Processing approach and its' correspondence to the metagraph model. The section V (which is the novel result presented in the article) discusses the metagraph agent implementation based on Functional Reactive Programming.

II. The Brief Description of the Data Metagraph Model

Metagraph is a kind of complex network model, proposed by A. Basu and R. Blanning in their book [6] and then adapted for information systems description in our articles [5, 7]:

$$MG = \langle V, MV, E, ME \rangle, \quad (1)$$

where MG – metagraph; V – set of metagraph vertices; MV – set of metagraph metavertices; E – set of metagraph edges; ME – set of metagraph metaedges.

Metaedge is an optional element of the metagraph model aimed for process description.

Metagraph vertex is described by a set of attributes:

$$v_i = \{atr_k\}, v_i \in V, \quad (2)$$

where v_i – metagraph vertex; atr_k – attribute.

Metagraph edge is described by a set of attributes, the source, and destination vertices and edge direction flag:

$$e_i = \langle v_S, v_E, eo, \{atr_k\} \rangle, e_i \in E, eo = true \mid false, \quad (3)$$

where e_i – metagraph edge; v_S – source vertex (metavertex) of the edge; v_E – destination vertex (metavertex) of the edge; eo – edge direction flag (eo=*true* – directed edge, eo=*false* – undirected edge); atr_k – attribute.

The metagraph fragment:

$$MG_i = \{ev_j\}, ev_j \in (V \cup E \cup MV \cup ME), \quad (4)$$

where MG_i – metagraph fragment; ev_j – an element that belongs to the union of vertices, edges, metavertices, and metaedges.

The metagraph metavertex:

$$mv_i = \langle \{atr_k\}, MG_j \rangle, mv_i \in MV, \quad (5)$$

where mv_i – metagraph metavertex belongs to set of metagraph metavertices MV; atr_k – attribute, MG_j – metagraph fragment.

Thus, metavertex, in addition to the attributes, includes a fragment of the metagraph. The presence of private attributes and connections for metavertex is a distinguishing feature of the metagraph. It makes the definition of metagraph holonic – metavertex may include a number of lower-level elements and, in turn, may be included in a number of higher-level elements.

The vertices, edges, and metavertices are used for data description while the metaedges are used for process description. The metagraph metaedge:

$$me_i = \langle v_S, v_E, \{atr_k\}, MG_j \rangle, me_i \in ME, \quad (6)$$

where me_i – metagraph metaedge belongs to set of metagraph metaedges ME; v_S – source vertex (metavertex) of the metaedge; v_E – destination vertex (metavertex) of the metaedge; atr_k – attribute, MG_j – metagraph fragment.

It is assumed that a metagraph fragment contains vertices (or metavertices) as process nodes and connecting them edges. A metagraph fragment can also contain nested metaedges, which makes the description of the metaedge recursive.

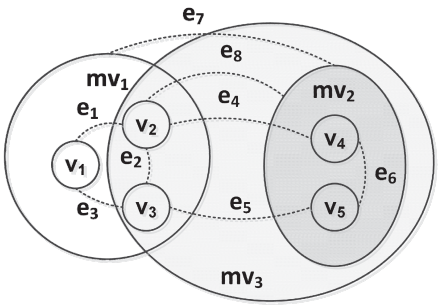

Fig. 1. The example of data metagraph

The example of data metagraph (shown in Fig. 1) contains three metavertices: mv_1, mv_2, and mv_3. Metavertex mv_1 contains vertices v_1, v_2, v_3 and connecting them edges e_1, e_2, e_3. Metavertex mv_2 contains vertices v_4, v_5, and connecting them edge e_6. Edges e_4, e_5 are examples of edges connecting vertices v_2-v_4 and v_3-v_5 are contained in different metavertices mv_1 and mv_2. Edge e_7 is an example of the edge connecting metavertices mv_1 and mv_2. Edge e_8 is an example of the edge connecting vertex v_2 and metavertex mv_2. Metavertex mv_3 contains metavertex mv_2, vertices v_2, v_3, and edge e_2 from metavertex mv_1 and also edges e_4, e_5, e_8 showing holonic nature of the metagraph structure.

The example of a directed metaedge is shown in Fig. 2. The directed metaedge contains metavertices mv_S, … mv_i, … mv_E and connecting them edges. The source metavertex contains a nested metagraph fragment. During the transition to the destination metavertex, the nested metagraph fragment became more complex, new vertices, edges, and inner metavertices are added. Thus, metaedge allows binding the stages of nested metagraph fragment development to the steps of the process described with metaedge.

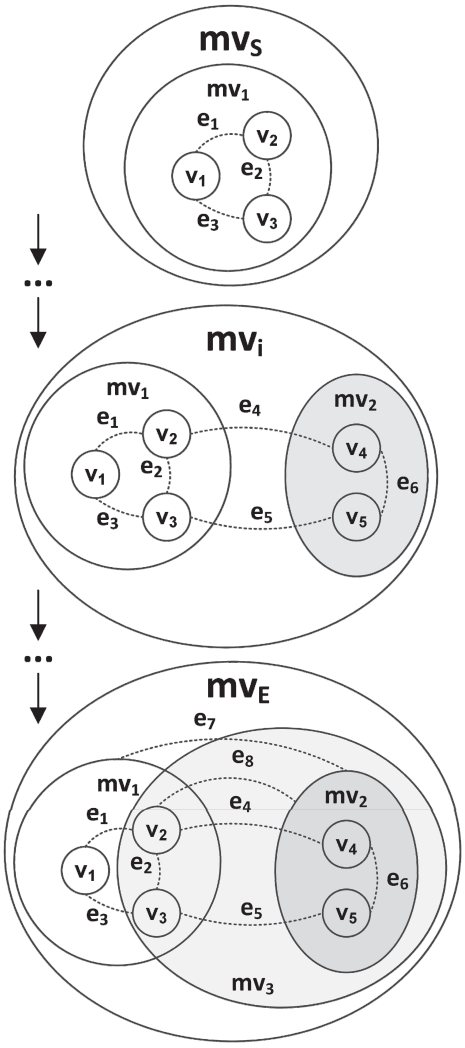

Fig. 2. The example of the directed metaedge

III. The Description of the Metagraph Agent Model

The metagraph model is aimed for complex data descriptions. However, it is not intended for data transformation. To solve this issue, the metagraph agent (ag^{MG}) intended for data transformation is proposed.

The choice of a multi-agent approach is quite evident because this approach is currently widely used in intelligent systems. This approach is proposed to be used for industry 4.0 [8], product life cycle management [9, 10], robotics systems

[11], including social robotics [12]. Therefore, we also use a multi-agent approach for metagraph processing.

There are two kinds of metagraph agents: the metagraph function agent (ag^F) and the metagraph rule agent (ag^R). Thus $ag^{MG} = ag^F \mid ag^R$.

The metagraph function agent serves as a function with input and output parameter in the form of metagraph:

$$ag^F = \langle MG_{IN}, MG_{OUT}, AST \rangle, \quad (7)$$

where ag^F – metagraph function agent; MG_{IN} – input parameter metagraph; MG_{OUT} – output parameter metagraph; AST – abstract syntax tree of metagraph function agent in the form of metagraph.

The metagraph rule agent is rule-based:

$$ag^R = \langle MG, R, AG^{ST} \rangle, \quad R = \{r_i\}, r_i : MG_j \to OP^{MG}, \quad (8)$$

where ag^R – metagraph rule agent; MG – working metagraph, a metagraph on the basis of which the rules of an agent are performed; R – set of rules r_i; AG^{ST} – start condition (metagraph fragment for start rule check or start rule); MG_j – a metagraph fragment on the basis of which the rule is performed; OP^{MG} – set of actions performed on metagraph. The antecedent of a rule is a condition over the metagraph fragment; the consequent of a rule is a set of actions performed on metagraph.

On the one hand, the principle of operation of metagraph rules is relatively simple and is based on metagraph calculus [13]. On the other hand, effective metagraph rewriting techniques are quite sophisticated and are currently being actively developed. Meanwhile, the classical graph transformation techniques developed for the transformation of flat graphs [24] are not explicitly suitable for the metagraph model. These techniques can only be used with restrictions if the metagraph is represented as a flat graph [17]. However, such a representation was designed exclusively for efficient storage of metagraph data, whereas data processing using such a representation can violate the semantics of the metagraph model.

Metagraph rules can be divided into open and closed. The consequent of an open rule is not permitted to change the metagraph fragment occurring in rule antecedent. In this case, the input and output metagraph fragments may be separated. The open rule is similar to the template that generates the output metagraph based on the input metagraph.

The consequent of a closed rule is permitted to change the metagraph fragment occurring in rule antecedent. The metagraph fragment changing in rule consequent cause to trigger the antecedents of other rules bound to the same metagraph fragment. However, an incorrectly designed closed rules system can cause to an infinite loop of metagraph rule agent.

Thus, the metagraph rule agent can generate the output metagraph based on the input metagraph (using open rules) or can modify the single metagraph (using closed rules).

The example of the metagraph rule agent is shown in Fig. 3. The metagraph rule agent "metagraph rule agent 1" is represented as metagraph metavertex. According to the definition, it is bound to the working metagraph MG_1 – a metagraph on the basis of which the rules of the agent are performed. This binding is shown with edge e_4.

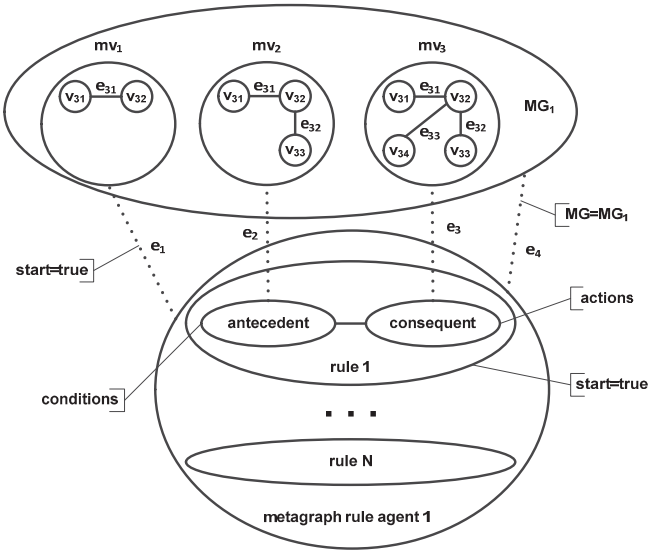

Fig. 3. The example of the metagraph rule agent

The metagraph rule agent description contains inner metavertices that corresponds to agent rules (rule 1 … rule N). Each rule metavertex contains antecedent and consequent inner vertices. In the given an example, mv_2 metavertex bound with the antecedent, which is shown with edge e_2 and mv_3 metavertex bound with consequent, which is shown with edge e_3. Antecedent conditions and consequent actions are defined in the form of attributes bound to antecedent and consequent corresponding vertices.

The start condition is given in the form of attribute "start=true." If the start condition is defined as a start metagraph fragment, then the edge bound start metagraph fragment to agent metavertex (edge e_1 in the given example) is annotated with the attribute "start=true." If the start condition is defined as a start rule, then the rule metavertex is annotated with attribute "start=true" (rule 1 in the given example). Fig. 3 shows both cases corresponding to the start metagraph fragment and the start rule.

The distinguishing feature of the metagraph agent is its homoiconicity, which means that it can be a data structure for itself. This is due to the fact that according to the definition metagraph agent may be represented as a set of metagraph fragments, and this set can be combined in a single metagraph. Thus, the metagraph agent can change the structure of other metagraph agents.

In order to combine the data metagraph model and metagraph agent model, we propose the concept of "active metagraph":

$$MG^{ACTIVE} = \left\langle MG^{D}, AG^{MG} \right\rangle, AG^{MG} = \left\{ ag_i \right\}, \quad (9)$$

where MG^{ACTIVE} – an active metagraph; MG^{D} – data metagraph; AG^{MG} – set of metagraph agents ag_i, attached to the data metagraph.

Thus, active metagraph allows combining data and processing tools for the metagraph approach. Similar structures are often used in computer science. As an example, we can consider a class of object-oriented programming language, which contains data and methods of their processing. Another example is a relational DBMS table with an associated set of triggers for processing table entries.

The main difference between an active metagraph and a single metagraph agent is that an active metagraph contains a set of metagraph agents that can use both closed and open rules. For example, one agent may change the structure of active metagraph using closed rules while the other may send metagraph data another active metagraph using open rules. Agents work independently and can be started and stopped without affecting each other.

IV. The Metagraph Approach for Event Processing

The idea of event processing based on the metagraph approach is discussed in [13]. According to [14], the event is defined as "a significant change in state," and event processing is defined as "a method of tracking and analyzing (processing) streams of information (data) about things that happen (events), and deriving a conclusion from them. Complex event processing (CEP) is event processing that combines data from multiple sources to infer events or patterns that suggest more complicated circumstances." According to [15]: "The goal of complex event processing is to identify meaningful events (such as opportunities or threats) and respond to them as quickly as possible."

Currently, much attention is paid to the concept of Semantic Complex Event Processing (SCEP), which is closely related to specific Semantic Web (SW) technologies. The book [16] offers an interesting idea that the results of a complex event processing in the creation of meaningful complex situations (such as opportunities or threats) and respond to them. The difference between descriptions [15] and [16] is that description [16] is more detailed and suggests the creation of meaningful complex situations as a result of complex event processing. It is a complex situation that is used for further analysis.

Thus, nowadays, the concept of an event is treated not as a separate data point, but as a complex situation with graph semantics. The RDF approach as a part of Semantic Web technologies is used to describe such semantics.

However, unfortunately, the RDF approach has several limitations for complex situation description [17]. There are reification limitation and N-ary relationship limitation. The root of limitations is the absence of the emergence principle in the flat graph RDF model. It is shown in [17] that the metagraph model addresses RDF limitations in a natural way without emergence loss. Thus, the metagraph model can be used instead of the RDF model for the Semantic Complex Event Processing approach.

The surprising fact is that to describe events in the metagraph model, it is not necessary to introduce new definitions. A complex semantic event is itself a metagraph.

There is also no need to change the metagraph agent definition. The agent's metagraph fragment for start rule check may be considered either as a fragment of static metagraph storage or a dynamic metagraph complex semantic event description.

The metagraph complex semantic event may contain either data metagraph fragment or metagraph agent description. Thus, the triad "data metagraph" – "metagraph agent" – "metagraph event" may be considered as a homoiconic structure based on the metagraph model.

In the following section, we will consider the software implementation of metagraph agents that use events based on functional reactive programming.

V. The Metagraph Agents Implementation

The construction of multi-agent systems based on functional reactive programming is a natural continuation of the concept of a reactive agent as an element based on behavior and reaction to external events.

Currently, many programming languages have implementations of reactive programming based on the actor approach. One of the most famous is the Akka library for the Scala language. There is also an implementation Akka.NET for languages on the .NET platform, such as C#, F#.

Reactive programming involves building an interdependent network of various elements that are either subscribed to various events, or are producers of these events, or implement both options. While reacting to the events, the network elements change their state, which applies to all network elements dependent on them. Thus, with each state change, a certain wave of changes passes through the reactive network of the system, which is the main idea of this approach.

Functional reactive programming (FRP) is based on two main ideas: behavior and event. Behaviors are mutable states that could be changed by some handling functions, which are called by events. It may look like an obvious anti-pattern for functional programming – mutable global states that are changed by some external events, but the power of types and separating mutable values their behaviors helps create fast, handful, and consistent systems based on this behavior.

The FRP was suggested in 1997 [18]. The main idea of FRP is mutable state elements called behaviors and occurrences of some messages, which are called events at some time. The behaviors are time-dependent values of some type. Reactivity is modeled by combinators, which produces new behaviors based on initial ones and events. It could be imagined as a function of

time with the state that changes to a new one when some event occurs. At that moment, the behavior value is replaced by the value that carries the event. Event is a discrete bundle of changes in the system; the event could carry some value or be empty, in the latter case, events will be just triggers for a system reaction.

With the first generations of FRP frameworks, events only had one occurrence, so that single-fire events are removed from the event streams after their occurrence. Most FRP implementations may occur multiple times and could be used in recursive descriptions of the network. In this case, we describe behavior as a state depended on its own events.

Some ways could give us comprehension problems: for example, an article [19] presents a number of issues with the explicitness of FRP implementation, such as ELM's complexity with initial value descriptions or Reflex's recursive monadic computations when the state is initiated by its own messages. However, all those difficulties are a small price for implementing reactive systems with the full power of self-rebuild.

Another approach to implement FRP architecture is to use the Arrow type class [20] to replace events and behaviors with "signals," which represents both of them as a stream of Maybe type, with occurrences implemented as Just value. This way is used in several Haskell libraries such as Yampa or Haskell-inspired language Elm, totally designed for FRP development. This article will be operating "classical" FRP architecture with behaviors and events.

As it is described in [21], behaviors and events could be identified as data types with corresponding functions:

```
at    :: Behavior a -> Ba
occs  :: Event a -> Ea
```

In FRP, Behavior and Event types carry a collection of several combinators to represent at some categorical elements. Behaviors could lift functions into themselves for it to be applied:

```
liftn :: (a1 -> ... an -> b)
-> (Behavior a1 -> ...
-> Behavior an -> Behavior b)
```

Moreover, as it is described in classical FRP articles [18, 21], the semantic instance for Behavior and Events corresponds to Functor, Applicative, Monad, and Monoid instances. That provides an opportunity to perform complex operations on these entities.

After a brief description of the basic principles of FPR, we shall describe how to add agent-based architecture into FRP. Despite the fact that the principles of FRP are implemented by a large number of libraries in different languages, our task was to propose the implementation of agents in FRP using the Haskell language.

As it was discussed in previous sections, the agent is a mutable structure that reacts to external messages. At first glance, it seems reasoned to map Agents into Behaviors and Messages into Events. However, this approach is not suitable because the Agent represents a more complex concept than Behavior. A simple representation of metagraph agents in the form of some objects enclosed in the Behavior would not allow ultimately to perform internal operations on them reactively and send and receive messages directly to the internal elements of the agent but would be a simple use of the FRP structure as just a complex container.

Consider the definition of the metagraph function agent (7). In that way, Agent consists of Behaviors agent and inner Events, which makes it possible to reconfigure the internal structure and its general condition. The main idea of metagraph agents is its homoiconic structure, which allows it to reconstruct its properties, actions, and goals when running the program. In order to implement this approach, it is possible to imagine a metagraph agent as a structure of Behaviors that listen to each other and emit and receive Events, thus representing properties or sets of properties. That dynamically changing structure of metagraph agents implements the main property of homoiconicity.

For Haskell FRP implementation of the metagraph agent, we will strictly consider its' definition, formulas (1), and (4). We may describe a metagraph agent as a complex type of Behaviors that contains vertices and edges of a metagraph. In this implementation, on its upper layer, the metagraph is a union (in this case, IntMap is used) of vertices, edges, metavertices, and metaedges:

```
MAgent edge node = MAgent {
      mAgentId    :: MAgentId,
      mAgentEdges :: IntMap (Behavior
(MAgentEdge edge vertex)),
      mAgentVertex      :: IntMap
(Behavior (MAgentVertex edge vertex))
}
```

The implementation of the metagraph vertex fully complies with formulas (2) and (5) as a set of attributes and an internal metagraph. Since the features of the Haskell language allow us to use optional types, we can combine the implementation of the vertex and metavertex into a single whole using type Maybe. In cases where the described object is a meta-vertex, the vertexMAgent field will exist, if the object is a simple vertex, then the value will be Nothing. Moreover, due to the fact that we enclose a potentially existing metagraph in Behavior, this element can go from the state of the vertex to the meta-vertex and vice versa during execution:

```
MAgentVertex edge vertex = MAgentVertex {
      vertexId          :: VertexId,
      vertexAttrs       :: vertex,
      vertexMAgent      :: Behavior (Maybe
            (MAgent edge vertex))
      }
```

The implementation of the edge, in turn, corresponds to the definition of the edge and metaedge from formulas (3) and (6). In the same way, as in the vertex implementation, we use type Maybe to combine the edges and the metaedges in the same type. The direction and attributes fields describe the corresponding mathematical representations in the metagraph. Thus, we also get a variable structure that can change its type

from edge to metaedge and vice versa depending on external or internal events:

```
MAgentEdge edge vertex = MetaEdge {
      edgeId       :: EdgeId,
      edgeDirected:: Directed,
      edgeFrom     :: Behavior
            (MAgentVertex edge vertex),
      edgeTo       :: Behavior
            (MAgentVertex edge vertex),
      edgeAttrs    :: edge,
      edgeMAgent   :: Behavior
            (Maybe (MAgent edge vertex))
}
```

Using that structure, we may describe functor instances for agent's vertices, edges and the agent itself:

```
instance Functor (MAgent edge vertex) where
  fmap f m = {
    mAgentEdges = fmap f <$> mAgentEdges m,
    mAgentVertex = fmap f <$> mAgentVertex m
}
instance Functor (MAgentVertex edge vertex)
where
  fmap f v = v {
    vertexAtrrs = f $ vertexAttrs v,
    vertexMAgent = fmap f <$> vertexMAgent v,
}
instance Functor (MAgentEdge edge vertex) where
  fmap f e = {
    edgeMAgent = fmap f <$> edgeMAgent e,
    edgeFrom = f <$> edgeFrom e,
    edgeTo = f <$> edgeTo e
}
```

To implement the definition of rule-based agent, we must strictly satisfy formula (8). To store the list of rules, there is an external table, which is implemented based on the definition of an agent as a set of metagraph and a list of rules, for which the correspondence between the agent and the rules to which it obeys is satisfied. As a result, the final type will look as follows:

```
RMAgent = {
      rmAgentId    :: AgentId,
      mAgent       :: Behavior MAgent,
      mrules       :: Behavior [Rules]
}
```

Such a metagraph agent implementation gives an opportunity to rebuild its structure during the execution. All elements of agents are Behaviors, so they are mutable functions of time and external Events that produce both internal elements of the agent and other agents or the system itself. Thus, Events in that approach could be used directly as agent messages; they can carry agents and agent's subgraphs or any other type if required. Implementation of agents seems to provide a kind of homoiconicity, as the agent could change their actions and structure using FRP mechanisms such as Behaviors and Events.

The agent implementation example (shown in Fig. 4) contains six metavertices: mAgentVertex1, mAgentVertex2, mAgentVertex3, mAgentVertex4, mAgentVertex5, mAgentVertex6 and two metaedges: mAgentEdge1, mAgentEdge2. Some of vertices are regular vertices (mAgentVertex1, mAgentVertex2, mAgentVertex3, mAgentVertex5, mAgentVertex6) and don't have internal metagraphs, one of them (mAgentVertex4) is metavertex and contains vertices mAgentVertex5, mAgentVertex6. One of edges is regular edge (mAgentEdge2) and another one is metaedge (mAgentEdge1), it contains mAgentVertex3. Vertices mAgentVertex4 and mAgentVertex6 producing messages as Events received by Behaviors: mAgentEdge1, mAgentEdge2, mAgentVertex3. All elements are contained within Behaviors, which allows them to transform reactively.

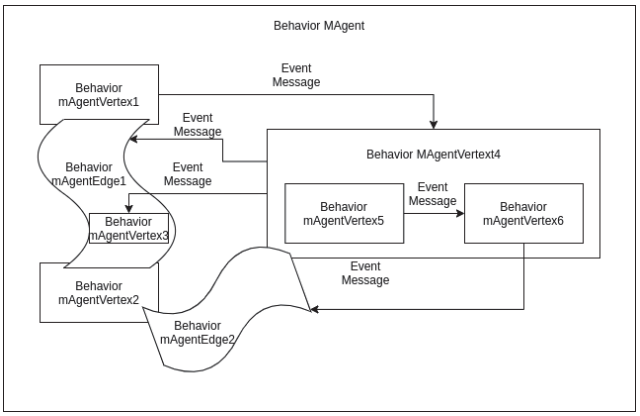

Fig. 4. The example of metagraph agent reactive implementation

For internal logic, the agent uses a rule-based engine, in which rules and functions are described in the internal metagraph in some kind of DSL. This DSL allows implementing the internal logic of agents, perform functions to change their own or external states, receive or send messages, superimposing on the logic of the FRP, and working at a meta-level, controlling the internal behavior of agents.

Among other things, this approach allows separating the data processing that can be done in parallel. Most of FRP implementations (for example, Haskell Yampa) has parallel functions, but do not allow to describe parallel computations "out of the box"; instead, they should be implemented by the developer. Article [22] describes the ways to parallel FRP computations, and they find their way on developing Yampa library [23], powerful and fast FRP realization. The cornerstone of paralleling Behaviors is the idea of splitting events into several ones to each Behavior which is subscribed to splitting Event.

One could spawn each new Behavior in a different thread so that it will be computed separately. In this case, there will be no struggle for resources, because when Behavior tries to change a state, it produces Event. Thus, the system does not have direct value mutations – all changes are performed reactively. As described in [23], signal functions (Yampa's way to implement Behavior FRP ideas) could be executed in parallel by using the switchers implemented in that library. The idea of representing Behavior as Signal was given in [20], where parallel switchers were used to separating computations in independent values.

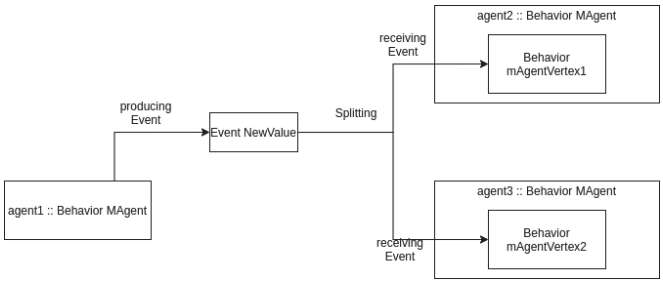

Fig. 5. The example of splitting events

The example of splitting events is shown in Fig. 5. The agent1 producing Event to which the vertices (mAgentVertex1, mAgentVertex2) of two independent agents (agent2, agent3) are subscribed. The event is splitted into two parallel threads that will be computed separately.

Thus, implementing metagraph-based Multi-Agent Systems via FRP has a lot of advantages. The use of behavior-event mechanics allows us to organize a new level of abstraction, as it provides the property of homoiconicity to metagraph agents. With that approach, agents could rebuild other agents and themselves reactively by emitting messages with the new parts of an agent's internal structure. Thanks to this developing way, the system obtains a significant advantage over non-reactively-based multi-agent systems. In addition to the general advantages of the functional language, the FRP provides the multi-agent system with the possibility of simple parallelism at the level of computing inside agents, which allows us to scale the system to extensive models.

VI. Conclusion

The metagraph model is a kind of complex network model. The emergence in the metagraph model is established using metavertices and metaedges.

For the metagraph model processing, the metagraph function agents and the metagraph rule agents are used. The distinguishing feature of the metagraph agent is its homoiconicity, which means that it can be a data structure for itself. Thus, the metagraph agent can change the structure of other metagraph agents.

To describe events in the metagraph model, it is not necessary to introduce new definitions. A complex semantic event is itself a metagraph. There is also no need to change the metagraph agent definition. The agent's metagraph fragment for start rule check may be considered either as a fragment of static metagraph storage or a dynamic metagraph complex semantic event description.

The metagraph agent may be implemented based on a functional reactive programming paradigm using the Haskell programming language. Such an implementation provides a kind of homoiconicity, as the agent could change its actions and structure using FRP mechanisms such as Behaviors and Events. The use of the FRP approach simplifies parallel data processing.

The proposed approach can be considered as a framework for complex model transformation. In particular, the article [5] discusses examples of neural network structure construction and the polypeptide chain synthesis.

References

[1] B.S. Manoj, Abhishek Chakraborty, and Rahul Singh, *Complex Networks: A Networking and Signal Processing Perspective*. New York: Pearson, 2018.

[2] V. Chapela, R. Criado, S. Moral, and M. Romance, *Intentional Risk Management through Complex Networks Analysis*. SpringerBriefs in Optimization, Springer, 2015.

[3] J. Johnson, *Hypernetworks in the Science of Complex Systems*. London: Imperial College Press, 2013.

[4] K.V. Anokhin, *Kognitom: gipersetevaya model mozga* [The cognitome: a hypernetwork brain model]. *Trudi XVII vserossiyskoy konferencii "Neuroinformatics-2015"* [Proc. XVII all-russian conference "Neuroinformatics-2015"], Moscow, 2015, pp. 14-15.

[5] V.M. Chernenkiy, Yu.E. Gapanyuk, G.I. Revunkov, Yu.T. Kaganov, Yu.S. Fedorenko, and S.V. Minakova, "Using metagraph approach for complex domains description", *In: Selected Papers of the XIX International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2017)*, Moscow, Russia, October 9-13, 2017, pp. 342-349. Web: http://ceur-ws.org/Vol-2022/paper52.pdf

[6] A. Basu, and R.W. Blanning, *Metagraphs and Their Applications*. Springer, 2007.

[7] V.M. Chernenkiy, Yu.E. Gapanyuk, A.N. Nardid, A.V. Gushcha, and Yu.S. Fedorenko, "The Hybrid Multidimensional-Ontological Data Model Based on Metagraph Approach", *Lecture notes in computer science*, vol. 10742, pp. 72–87. Springer, 2018.

[8] V.B. Tarassov, "Enterprise total agentification as a way to industry 4.0: Forming artificial societies via goal-resource networks", *In: Intelligent Information Technologies for Industry 2018*, AISC, vol. 874, pp. 26-40. Springer, 2018.

[9] V. Taratukhin, and Y. Yadgarova, "Towards a socio-inspired multiagent approach for the new generation of product life cycle management", *Procedia Computer Science*, vol. 123, pp. 479-487, August 2017.

[10] V.O. Karasev, and V.A. Sukhanov, "Product Lifecycle Management Using Multi-agent Systems Models", *Procedia Computer Science*, vol. 103, pp. 142-147, October 2016.

[11] A.V. Nazarova, and M. Zhai, "Distributed Solution of Problems in Multi Agent Robotic Systems", *Studies in Systems, Decision and Control*, vol. 174, pp. 107-124, 2019.

[12] V.E. Karpov, and V.B. Tarassov, "Synergetic artificial intelligence and social robotics", *In Intelligent Information Technologies for Industry 2017*, AISC, vol. 679, pp. 3-15. Springer, 2017.

[13] Yu. Gapanyuk, "The Semantic Complex Event Processing Based on Metagraph Approach", *In Biologically Inspired Cognitive Architectures 2019*, AISC, vol. 948, pp. 99-104, Springer, 2019.

[14] D. Luckham, *Event Processing for Business: Organizing the Real-Time Enterprise*. John Wiley & Sons, 2012.

[15] Wil M. P. van der Aalst, *Process Mining: Data Science in Action*. Springer, 2016.

[16] K. Teymourian, *Knowledge-Based Complex Event Processing: Concepts and Implementation*. Südwestdeutscher Verlag für Hochschulschriften, 2016.

[17] V.M. Chernenkiy, Yu.E. Gapanyuk, Yu.T. Kaganov, I.V. Dunin, M.A. Lyaskovsky, V.A. Larionov, "Storing Metagraph Model in Relational, Document-Oriented, and Graph Databases", *In: Proceedings of the XX International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2018),* Moscow, Russia, October 9-12, 2018, pp. 82–89. Web: http://ceur-ws.org/Vol-2277/paper17.pdf

[18] C. Elliott, and P. Hudak, "Functional Reactive Animation", *In the proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. Web: http://conal.net/papers/icfp97/icfp97.pdf

[19] S. Krouse, "Explicitly Comprehensible Functional Reactive Programming", *REBLS'18*, November 2018. Web: https://futureofcoding.org/papers/comprehensible-frp/comprehensible-frp.pdf

[20] H. Nilsson, A. Courtney, and J. Peterson, "Functional Reactive Programming, Continued". *In: Proceedings of the 2002 ACM*

SIGPLAN Haskell Workshop (Haskell'02). Pittsburgh: ACM Press, Oct. 2002, pp. 51–64.

[21] C. Elliott. "Push-pull functional reactive programming", *In Haskell Symposium*. Web: http://conal.net/papers/push-pull-frp/push-pull-frp.pdf

[22] J. Peterson, V. Trifonov, and A. Serjantov, "Parallel Functional Reactive Programming", *In: Practical Aspects of Declarative Languages. Second International Workshop, PADL 2000.* Boston, 2000, pp. 16-32. Web: https://ia800206.us.archive.org/6/items/springer_10.1007-3-540-46584-7/10.1007-3-540-46584-7.pdf

[23] A. Courtney, H. Nilsson, and J. Peterson, "The Yampa arcade." *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell.* August 2003, pp. 7–18. Web: https://www.antonycourtney.com/ pubs/hw03.pdf

[24] H. Ehrig, C. Ermel, U. Golas, and F. Hermann, *Graph and Model Transformation. General Framework and Applications*. Springer, 2015.