Actors on Graph

A new formal model for specification and implementation contact center applications

By Nikolay Anisimov

September 15, 2017

In this post, I will demonstrate how adding graph component to a formal model dramatically increases its expressive power allowing specifying complete complex systems. At the same time, supporting graph models by database vendors makes these models be executable. More specifically, we suggest a combination of graph model with the actor model. This model is a part of a new technology aimed at creation and execution contact center applications in cloud described in this whitepaper.

During the last twenty years of working in contact center industry, I had a dream to invent a means for complete specification of contact center applications comprising customer interaction processing, dialog with customers, interaction routing, self-service, agents' involvement, etc. The enthusiasm was stemmed from old W3C recommendations VoiceXML and CCXML, technologies for specifying IVR-like dialog management and simple call control, respectively. However, both technologies covered only part of contact center applications leaving agent involvement outside of application logic. As a further attempt in this direction, I should mention XContact, an XML-based language to specify complete contact center applications. However, this initiative was not technically successful, as it suffered from a lack of a well-defined formal model. We considered many formal models (e.g., state machines, Petri nets, state charts, Harel's graphs, SCXML, etc.) to find an appropriate formalism but without any success.

In this notes, I introduce a new formal model called Actors on Graph (AoG). AoG is composed of a well-known actor model with a property graph model. The graph component explicitly and naturally describes relationships between contact center entities (i.e., relationships between customers and agents, between agents and their skills, organization structure, and so forth). The actors' part specifies the dynamic of the system. The model is executable in the sense that it could be automatically converted to executable code for the concrete execution environment. This model becomes possible due to substantial progress in graph databases and distributed systems in general.

Formal Model: Actors on Graph

Labeled Property Graphs

In short, a graph is a structure composed of two types of elements: a node and a relationship. Each node represents an entity and each relationship represents how two entities are associated. Graphs have a convenient graphical representation where a node is depicted as a circle and a relationship is represented as an oriented arc between two circles.

A graph is a good model to represent entities and their relationships. But at the same time, it is a low-level model that does not allow the representation of attributes and parameters.

A labeled property graph (LPG) is one of the extensions aimed to address this limitation. In short, a labeled property graph is a graph with nodes and relationships where each node and relationship has a set of properties. Each node may also have a set of labels. Graphically, nodes and relationships are depicted as circles and directed arrows. The simple LPG with two nodes, A and B, and a relationship between them is represented in Figure 3.

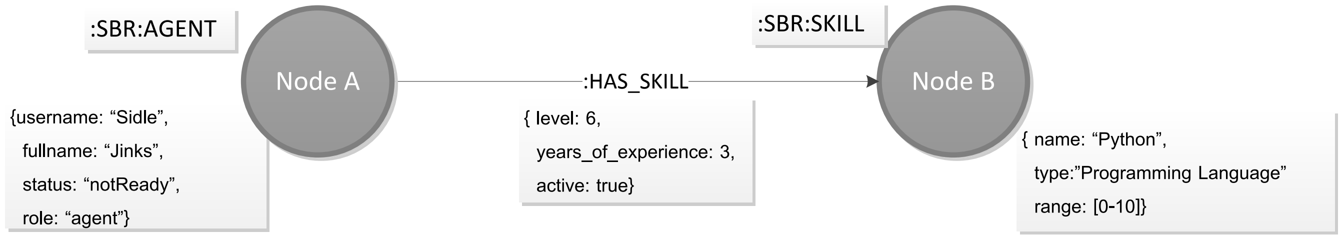


Figure 1: Labeled Property Graph

Node A has a set of four properties and Node B has a set of three properties. The relationship between them also has a set of three properties. Each property is specified as a key-value pair. A set of properties could be serialized using JSON format.

The relationship has the type “HAS_SKILL.”

Nodes A and B also have sets of labels that could be used for grouping nodes. They are defined as a sequence of label names separated by colons. For example, an expression “:SBR:AGENT” means that a corresponding node has two labels “SBR” and “AGENT.” The order of the sequence does not make sense.

This LPG fragment shown in Figure 3 may describe an agent with the name Sidle Jinks has a skill called “Python”. The skill belongs to the type of programming language and its level is measured as an integer within the range between 0 and 10. The agent has a skill level equal to 6 and experience with it equal to three years.

LPG Ontology

When we model a real-world system, we describe many entities of the same type. For example, the system may contain many customer interactions with the same structure.

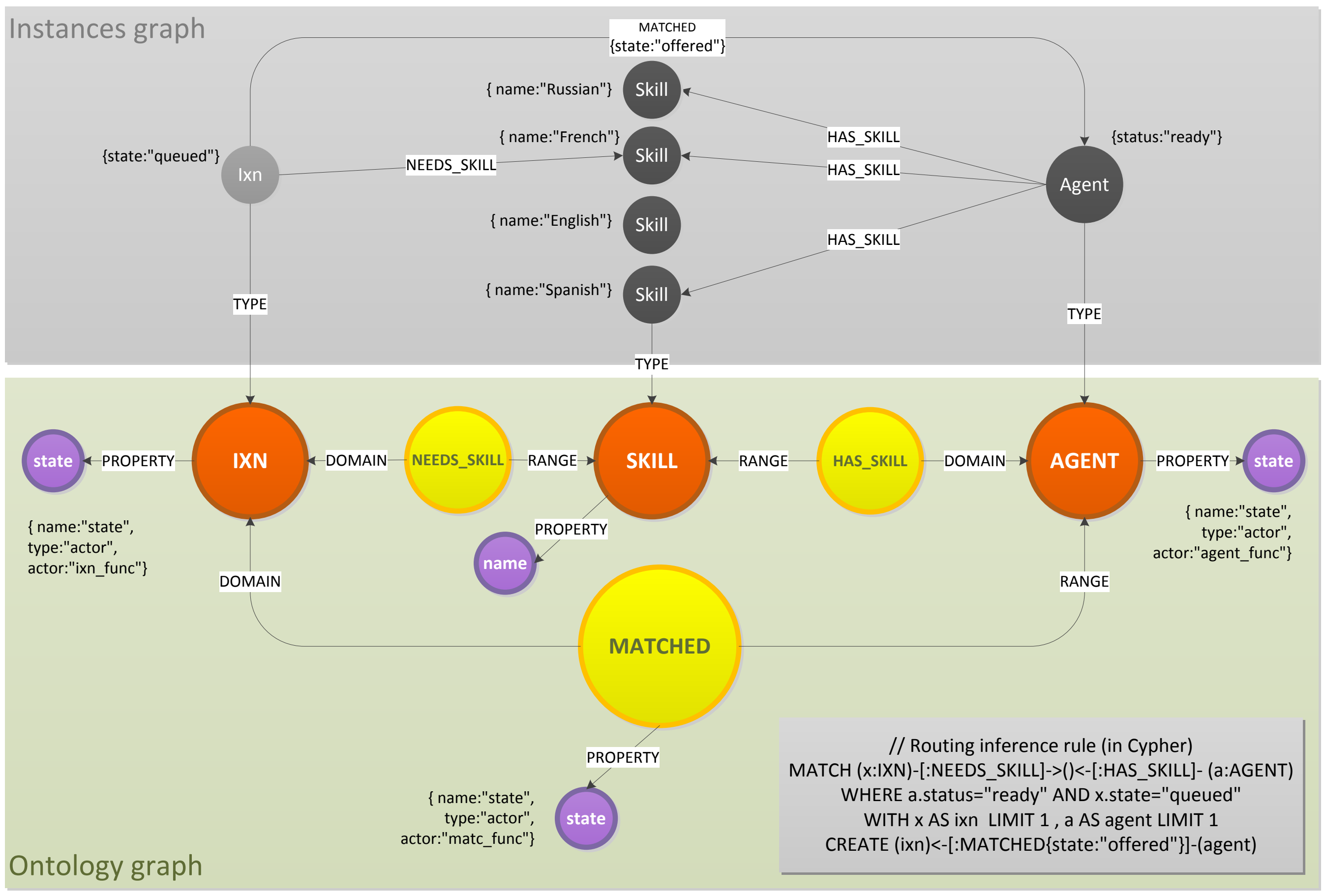
To represent a structure of the model, we will use LPG. We introduce three types of nodes: Object Class, Relationship Class, and Property. They allow describing node and relationship instances as well as their properties, respectively.

We will call the description of graph structure an LPG ontology, which reflects the fact that it contains knowledge about the structure of the system. Figure 2 illustrates the usage of ontology.

The ontology graph is placed in a bottom part of the picture. It contains objects and relationships defining a structure of the instance graph. The ontology defines three types of node instances: interaction (IXN), agent (AGENT), and skill (SKILL). It also has three types of relationships: NEED_SKILL, HAS_SKILL, and MATCHED. The last relationship represents the results of routing as the execution of an inference rule (see the bottom right corner of Figure 2). Property nodes define a data structure of corresponding instances.

The upper part of Figure 2 contains the instance graph that is, in turn, partitioned into the configuration (black nodes) and operational parts (gray nodes).

LPG is a good means for representing a structure of the system. At the same time, the system evolves in time by creating and deleting new instances, and changing properties of instances. The dynamic behavior of the system needs to be specified, and to address that need, we will introduce the notion of actors.



```
// Routing inference rule (in Cypher)
MATCH (x:IXN)-[:NEEDS_SKILL]->()<-[:HAS_SKILL]- (a:AGENT)
WHERE a.status="ready" AND x.state="queued"
WITH x AS ixn  LIMIT 1 , a AS agent LIMIT 1
CREATE (ixn)<-[:MATCHED{state:"offered"}]-(agent)
```

Figure 2: Example of Ontology Graph and Instances for Skill-Based Routing

Actors

The actor model is a popular model of concurrent computations introduced by Professor Carl Hewitt in 1973. A good informal introduction to the actor model can be found in this blog. The actor model has made an impact on several programming languages. On the other hand, it could be implemented by almost any procedural programming language.

The main notion of the actor model is a notion of *actor* that is a fundamental unit of computation. An actor is an entity that can do the following:

- Receive messages
- Perform some computations
- Have local memory for keeping a state
- Send messages
- Create other actors

Actors have no common memory and communicate only via message exchange. To send a message, an actor should know an address of the corresponding actor. When an actor sends a message to another actor with its address, the platform delivers the message no matter where the actor is located.

Actors on Graph

Our formal model called Actors on Graph (AoG) is a composition of LPG and the actor model. In short, some node and relationship instances are equipped with actors that can communicate with each other only via arcs of the graph.

The AoG model is defined as follows:

- The model contains LPG with LPG ontology.
- Each node or a relationship may be associated with an actor.
- An actor is defined for object or relationship class and each corresponding instance will have this actor.
- Each actor has only one variable to specify its state. This variable is defined as a property of a corresponding node or relationship with the type "actor."
- An actor can receive messages from adjacent actors.
- An actor can receive messages from outside of the system to initiate model computation.
- Upon receiving some message from adjacent actor, an actor may return a respond message to this actor.
- Each actor may broadcast a message to all other adjacent actors. An adjacent actor is an actor that is associated with an adjacent node or relationship.
- An actor can create and delete nodes and relationships.
- Upon receiving some message, an actor can do nothing to stop the disturbance wave in this direction.

Figure 3 illustrates actor computation on graph.

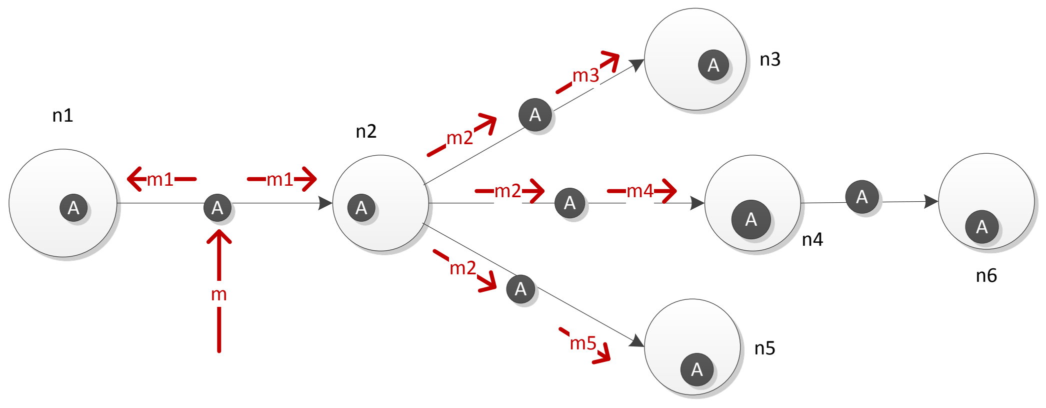


Figure 3: Example of Actors on Graph Model

The LPG contains six nodes and five relationships, each of which is associated with an actor. The model is triggered by an external message m sent to the actor of relationship (n1,n2) between nodes n1 and n2. The actor starts computation processing of the received message and may change its state. It also may send a message $m1$ to the adjacent actor of nodes n1 and n2. The actor of n2, in turn, does some computation and broadcasts a message m2 to the actors of relationships (n2,n3), (n2,n4), (n2,n5). And so on. Notice the actor of node n4 on receiving the message m4 stops the wave and does not send any message towards n6. Messages propagate to all other actors and the whole computation terminates.

One can think of AoG as the conventional actor model where communications between actors are limited by graph topology. Moreover, only local communications between adjacent actors are allowed.

Model Implementation

There are many ways to implement the AoG model using different technologies and programming languages. In this project, we use the following approach based on Neo4j, Node.js, and AWS Lambda. As we will see, the model implementation is straightforward and could be done for any other technology stack and cloud environment.

LPG and LPG ontology are implemented with the aid of the Neo4j database. Actors and other internal procedures are implemented as Node functions based on the Neo4j library. Communication between adjacent actors is implemented as invoking one actor function from another one. All these functions are implemented and deployed as several AWS Lambda functions.

Thus, all contact center infrastructure is implemented as a node.js library. Contact center applications are built as an AoG model, LPG graph equipped with actor functions.

Model Serialization

Currently, the AoG model is serialized by a Cypher query that creates LPG in Neo4j, plus a set of NodeJS functions for node and relationship actors. The example of this notation representing specification object class for interaction is shown below:

```
CREATE (ixn:ObjectClass:O:SBR{name:'IXN',type:'ObjectClass',new_type:'ObjectInstance',role:'customer',...})
CREATE (ixn)-[:PROPERTY]->(:Property:O:SBR{type:'property',name:'media_type',owner:'system',cardinality:1,...})
CREATE (ixn)-[:PROPERTY]->(:Property:O:SBR{type:'property',name:'ixn_id',owner:'system',cardinality:1,...})
CREATE (ixn)-[:PROPERTY]->(:Property:O:SBR{type:'property',name:'customer',owner:'system',cardinality:1,...})
CREATE (ixn)-[:PROPERTY]->(:Property:O:SBR{type:'property',name:'state',owner:'system',cardinality:1,...})
```

This format works but looks quite peculiar. This forces us to think about the creation of some domain specific language (DSL). At the same time, a Neo4j graph has a convenient JSON version of LPG representation. This format could be employed for our DSL and final version of AoG DSL will be in JSON format. This is convenient when working with AoG is planned with the aid of some framework.

One may ask - what about XML as a means of serializations? Indeed, VoiceXML and CCXM used XML format. I do not exclude XML but we should understand that XML is nothing else but a serialization format of formal model.

Examples

Example: Skill-based interaction routing

Let us return to the Figure 2 that depicts a graph of a contact center application known as skill-based routing. In short, this type of routing distributed inbound interaction to agents with corresponding skills. Object classes IXN and AGENT depicted by orange nodes define customer interactions and agents respectively. Agent skills are described by the object class SKILL. The relationship class HAS_SKILL defines relationships between agents and their skills. The relationship class NEEDS_SKILL specifies relationships between customer interactions and agent skills.

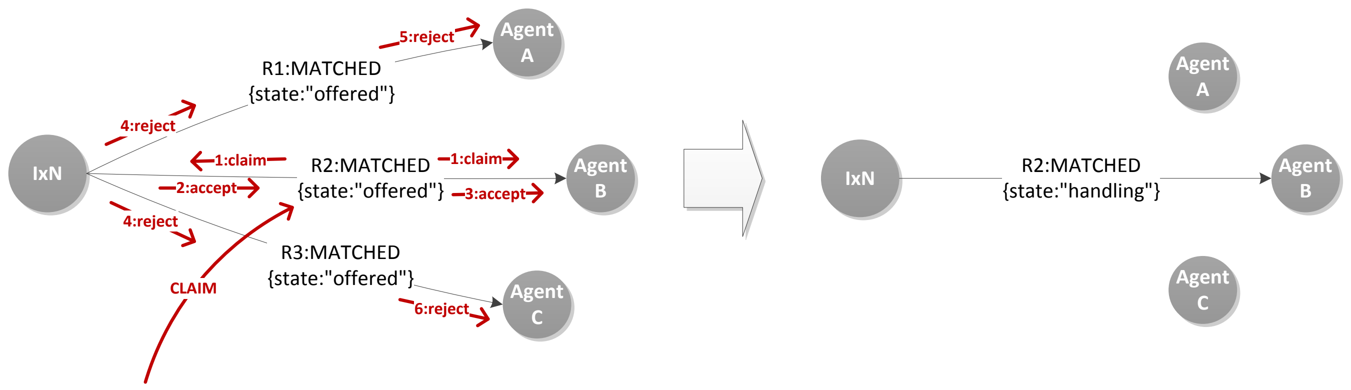
If you take a look at instances you could see that the system contains one interaction and one agent. The agent is configurable instance and is created at configuration time as well as her skills. The interaction is operational instance is created in a real time when a customer contacts the system (e.g. by calling by phone).

Routing logic is as inference rule expressed in terms of Cypher query. When this inference rule is executed a new relationship MATCHED is established between the interaction and the agent as they have common skill “French”.

The only question left is how we could identify what language skill needs the interaction? Suppose the interaction is a telephone call that initially has no such information. Obviously, we could collect this information from a customer using a conversation dialog (e.g. by IVR). But how to accomplish this dialog this is another story.

Example: Agent reservation

In the previous example we have seen how interactions are matched to agents. This was implemented as creation corresponding relationships MATCHED with state property “offered”. In practice, matching should be accompanied by actual assignment the interaction to one agent. To accomplish this, a procedure usually called agent reservation is employed. Normally a new interaction is offered to all matched agents. When some agent decides to take the interaction he/she claims it via his/her desktop. This intention propagates to the interaction and it accepts the claim and rejects other offerings. A relationship MATCHED for the accepted agent transitions to “handling” state.



```
% Actor for interaction IxN node
receive
    claim -> respond accept
             broadcast reject
end
```

```
% Actor for relationship MATCHED
receive
    CLAIM ->  broadcast claim;
    accept -> state=”matched”
              broadcast accept;
    reject -> broadcast reject
              delete self
end
```

```
% Actor for agent node
receive
    claim -> inform
    accept -> inform
    reject -> inform
end
```

Figure 4: Agent reservation procedure

Figure 4 illustrates an implementation of agent reservation procedure using AoG. The LPG contains four object instances representing interaction (IxN) and three agents A, B, and C. The interaction is matched with and offered to all three agents. The specification includes three actors expressed in Erlang-like pseudo code.

Suppose the agent B decides to take the interaction. She sends a claim request to the relationship R2 indicating the agent and the selected interaction. This will result in the following sequence:

1. The actor of R2 broadcast the claim message to actors of interaction IxN and agent B. The actor of agent B may inform its agent about starting the reservation procedure.
2. The actor of interaction IxN responds to the actor of R2 with acceptance message,

3. The state of the relationship R2 is changed to “handling”. The acceptance message propagated to the actor of agent B and may inform its agent about completing the reservation procedure.
4. At the same time the agent of interaction IxN broadcasts a reject message to other relationships R1 and R3.
5. The actor of R1 retransmits the reject message to its agent actor and deletes the relationship R1.
6. The same is done by the actor of relationship R3.

As a result, we have a graph where the node of interaction IxN and the node of agent B are connected by the relationship MATCHED in state “handling”. This interaction is not offered to any another agent any mode.

Conclusion

In these short notes, I informally introduced a new formal model called actors on a graph to be used for complete specification of contact center applications. The model is a composition of well-known actor model with a model of labeled property graph. In this notes, we considered the only adequacy of the model to a wide class of contact center applications and related areas. Definitely, the model needs future research in different directions of formal correctness to avoid undesirable loops and other logical errors, ways of serialization and implementation in concrete cloud environments, performance aspects.

More about this project could be found this whitepaper.

About the Author

Nikolay Anisimov, Ph.D., is a computer scientist with a strong academic background. He is an industry veteran with twenty years’ experience in contact center technologies, is the author of numerous patents, technical and research papers, articles in industry journals and whitepapers. Nikolay has worked for Genesys, Alcatel-Lucent, FrontRange Solutions, Five9, Aspect Software, and Bright Pattern, Inc. He is a co-founder of Contact Technology Labs, Inc. You can contact Nikolay at Email: anisimov@computer.org