Modeling Socio-technical Systems with AgentSpring

Alfredas Chmieliauskas[1], Emile J.L. Chappin[2] and Gerard P.J. Dijkema[3]

[1,2,3]Energy and Industry Section
Faculty of Technology, Policy and Management
Delft University of Technology
2628 BX Delft, The Netherlands

a.chmieliauskas@tudelft.nl, e.j.l.chappin@tudelft.nl, g.p.j.dijkema@tudelft.nl

Abstract. *AgentSpring is a new agent-based modeling framework especially suited to model and simulate complex socio-technical systems, such as energy markets or transport infrastructures. Common problems encountered when modeling and analyzing such systems are how to represent the variety of facts that describe the system and how to allow agents to make decisions based on those loosely related pieces of information. AgentSpring proposes a solution to these problems by modeling the systems as graphs consisting of agents, artifacts and relations between them. A graph is one of the most flexible data structures, and is made of vertices and edges that connect them. Agents use sophisticated queries to reason over and traverse the graph data structure and make decisions based on the results they discover. They continuously update the graph to include the effects of the decisions they have made. AgentSpring represents the simulated system as a constantly evolving graph of interconnected facts that can transparently be observed and queried in real time by both agents in the model, and the modeler. In addition, AgentSpring allows for the composition of agent behavior modules, where sophisticated behavior can be produced by combining simpler decision making rules. The modeler can easily create a number of heterogeneous agents by combining these behavioral "lego" bricks. Finally, AgentSpring is open-source and is continuously developed based on the feature requests and contributions of the modeler community.*

Keywords. *Agent-based, modeling, framework, graph, simulation.*

1 Models of Socio-technical Systems

Conceptualizing a phenomenon as a complex socio-technical system is a paradigm increasingly applied to institutional economics and policy research (Chappin, 2011). Socio-technical are systems that span both technical and social components. Such systems include social structures that develop around a particular industrial system, for example, an industrial cluster, energy or transportation infrastructure. They consist of a large

number of diverse technical artifacts, such as machines, factories, pipelines and wires. They also include social components, such as organizations and institutions that shape the technical components and at the same time are shaped by them (Nikolic, 2009).

The common tool used to research such systems is agent-based modeling (Farmer, 2009). Typically, agent-based models (ABM) decompose a system into agents, artifacts and rules that determine their interaction. There are various ABM frameworks and philosophies that propose useful methodologies for decomposing the system or elaborating the behavior of an agent. This makes ABM a discipline on its own. Before finding their way into socio-technical systems research, agent-based models have been applied in natural sciences to simulate such phenomena as spread of viruses or flocking birds.

There are significant differences between modeling bird flocks and social systems. Models of socio-technical systems contain multiple heterogeneous artifacts and actors who have to make decisions based on the versatile information compiled from the state of the system. This particular paradigm inspired the development of a new agent-based modeling framework that would be especially suited for modeling socio-technical systems.

2 Why Another Agent-based Modeling Framework?

Given that ABM is increasingly applied to a wider range of domains, from biology, to architecture, to economics, researchers have become aware of the need for different tools to suit their domain. Different requirements for ABM tools explain the wide variety of ABM packages and frameworks already available. There are more than 60 ABM frameworks/libraries/packages available already[1], some more popular than others. Some of the notable requirements and tools that implement them are: speed (GPU), simplicity (NetLogo), multi-model integration (FLAME), visual tools and IDE integration (RepastSymphony) or 3D visualization (MASON).

Models of socio-technical systems present different set of requirements and challenges. Such models are commonly defined as having many heterogeneous agents, many different (social or institutional) relations and arrangements between these agents, a variety of technical artifacts and complicated decision making behavior. These modeling requirements justified an effort to create a new agent-based modeling framework to address them in a more straightforward fashion. Surely, these requirements could be fulfilled by the existing frameworks, provided some modifications were made. But there was also the opportunity to build a framework that would leverage off the new and powerful open source libraries and changing software development paradigms.

1 http://en.wikipedia.org/wiki/Comparison_of_agent-based_modeling_software

3 AgentSpring Framework

Following sections review the features of AgentSpring that differentiate it from other ABM frameworks and fulfill the requirements presented by modeling socio-technical systems.

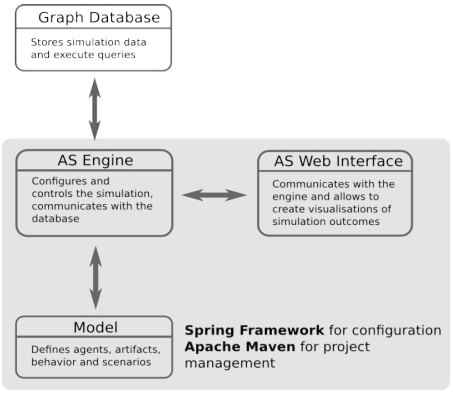

Fig.1. AgentSpring Framework structure.

3.1 Software Stack and Design Paradigm

AgentSpring is developed as an open-source tool. This implies that anyone can use and contribute to the platform. AgentSpring is available on-line. AgentSpring is based on Java technologies and runs on all popular operating systems: Linux, Windows, and Mac.

AgentSpring gets its name from and makes use of Spring Framework – a popular enterprise software development framework, that promotes the use of object oriented software patterns (Johnson et al, 2009). One such pattern calls for separation of data, logic and user interface (Krasner and Pope, 1988). Most modeling frameworks mix the three, which it is a reasonable choice when creating smaller models. However, the separation of concerns (Hursch and Lopes,1995) and other patterns are especially helpful guidelines for creating models and applications that are sophisticated, extend-able but transparent.

3.2 Graph Data Structure and Native Graph Database

Some of the existing ABM frameworks allow the model data to be stored in a relational database (RDMBS). RDMBS relies on tables to store information and usually require a table to be created for each type of object (Java class) to be stored. When modeling socio-technical systems with many different types of objects, RDBMS are not a very practical alternative. The RDBMS are optimized to deal with large numbers of things of the same type, not with smaller number of things of different types. In other words, the RDBMS's are designed to scale to size (many things), not complexity (many types).

To enable storing a complex network of heterogeneous actors and artifacts in a simulation environment AgentSpring makes use of graph data structures. Mathematics defines a graph a set of objects - called nodes - connected by links, called edges (Bondy and Murty, 1976). There numerous types of graphs, or graph topologies, and their properties that are the target of study of graph theory and discrete mathematics. Computer science refers to a graph as an abstract data structure (ADS) (Skiena, 1998). Most computer languages implement the common ADS, such as stacks, lists, maps, sets and trees, but not graphs. These data structures have distinct properties and provide methods enabling the developer to organize and work with data. It is only recently that the changing needs of the Internet technologies have contributed to the development of alternative data structures and databases, including native graph databases (Angles and Gutierrez, 2008).

A graph database is a database that uses graph data structures with nodes, edges, and properties to represent and store information (Eifrem, 2009). Java objects and relations between them already resemble a graph, making storing them in a graph database relatively straightforward. Every object can be mapped to a node, its primitive members (such as int, double and boolean) are mapped to node properties. Object's complex members are mapped to other vertices and an edge between them is created. Graph databases allow the graph to scale to hundreds of agents, millions of things and relations between them, as represented in appendix D. Such graph databases already power the social networking and other Internet services. The application of graph databases in ABM is novel and promising as it allows for more straightforward representation and storage of the system modeled. Currently, AgentSpring uses Neo4J graph database[2], which not only provides excellent performance, but also features Spring Framework integration and supports multiple query languages.

3.3 Graph Queries

The ability to query the graph and inspect the state of the simulation is a key benefit of using a graph database in a simulation environment. These queries can be used both by the modeler and the agents in the model. The agents query their environment and use the information to drive their decision making logic. The modeler can use the queries to extract relevant information about the state and the outcomes of the simulation. AgentSpring makes use of four different methods to query the graph: Gremlin query language[3], Cypher query language[4] and SPARQL query language[5].

Without overwhelming the reader with peculiarities and differences in these languages it is important to note that they can be used interchangeably. The choice is there to provide

2 http://neo4j.org/
3 https://github.com/tinkerpop/gremlin/wiki
4 http://docs.neo4j.org/chunked/1.4/cypher-query-lang.html
5 http://www.w3.org/TR/rdf-sparql-query/

the modeler with flexibility to use the method best suited for the purpose. Examples of such queries will be discussed with more detail in the case study to follow.

3.4 Agent Behavior

AgentSpring decouples agents, their behaviors and their environments making the pieces reusable, composable and easy to manage. Experience has shown that modular and reusable models are the only kind of models that can accommodate changing scope and new research questions. This decomposition is inspired by the artificial intelligence classic "Scripts, Plans, Goals, and Understanding: An Inquiry Into Human Knowledge Structures" by Roger C. Schank and Robert P. Abelson. The book suggests that human behavior and understanding of the world is compartmentalized as scripts that are used to execute bigger plans and higher goals (Schank and Abelson, 1977). When executing a plan to go to a restaurant, a person would invoke a script to make reservation in advance, call a cab, perhaps dress up and so on. AgentSpring makes use of the scripts concept to encode agent behavior in a modular way. Agents behave the simulation by invoking various scripts. Simulations are made by combining agents and scripts that rule their behavior in a predefined or random sequence. Modularity and composition of behavior logic are essential for making behavior-rich models.

Agents use graph queries defined by the modeler to extract the relevant pieces of information from the simulation. That information facilitates the decision making algorithms of the agent. The brevity and conciseness of the query languages allow to write powerful queries that collect information without iterating though lists or maps. The main advantages of using query languages for agent intelligence are two. Firstly, they provide the modeler with high level of expression - to encode complex questions in a straightforward way. Secondly, they provide a unmatched performance advantage for the level of complexity they embody. Performing a similar query iterating through lists, maps and other ADS or even using SQL queries on a relational database would penalize models performance significantly.

3.5 User Interface

Finally, AgentSpring offers a web-based user interface. See figure 3 for a snapshot of a model result charts displayed via the web interface. This feature makes AgentSpring accessible both to the developer and the user (for example, a policy maker) via a familiar interface of any web browser.

The interface allows to start, pause and stop the model, to change and create graphs by writing queries and to observe a textual log. Additionally, the interface can be used to select various predefined scenarios and to change key parameters in the model. The web interface also exposes the running simulation to a variety of statistical and visualization

tools. Currently, AgentSpring includes a R[6] library allowing seamless integration with the statistical package. A model in AgentSpring can also be controlled from command line without running the AgentSpring user interface. Please see figure in the appendix C for the snapshot of the user interface.

4 Case study: Interactions Between Power, Natural Gas and CO2 Markets in the EU

The following section serves to demonstrate the use of AgentSpring framework in building a model. The model explores the interactions between power, fuel (natural gas) and CO2 markets in the European Union. The purpose of the model is explain the dynamics between the different energy policies and the actions of the market participants. The following passages discuss the actors and their behaviors, but not the outcomes of the simulation.

4.1 Model Structure

The main actors in the model are power producers who sell the generated electricity in the power market, they buy fossil fuels needed by their power plants in the fuel markets, and they purchase the CO2 emission allowances in the centralized EU ETS auction. In order to sell or buy in a power, fuel or CO2 the agents submit their bids or offers to the market. The market is cleared given the supply and demand and a uniform price and volume is determined.

Besides the bidding behavior the power producers invest in new power plants that change the supply dynamics in the power market. The new power plants have different fuel requirements and CO2 emissions profile that affects the demand characteristics in the fuel markets and the CO2 allowances auction.

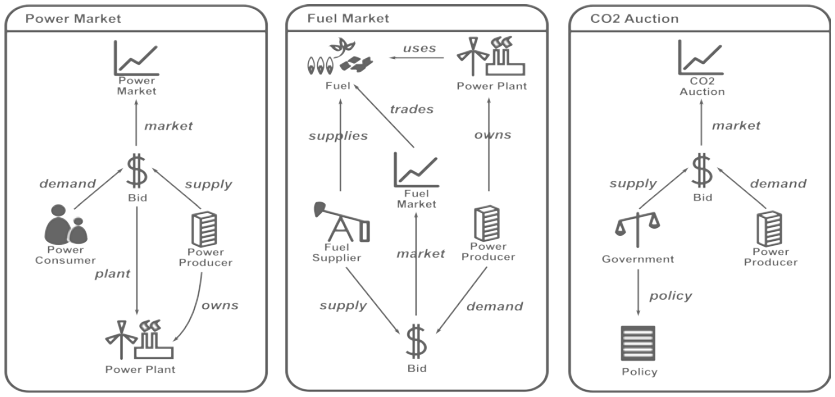

Fig.2. Model structure as a graph. Each icon represents a node, an arrow – an edge between the nodes. The graph consists of both agents (such as Power Producer) and artifacts (Power Plant, Bid).

6 http://www.r-project.org/

While the agents and the markets constitute the social components of the system, the technical artifacts in the model are the power plants, the power generating technologies and fuels.

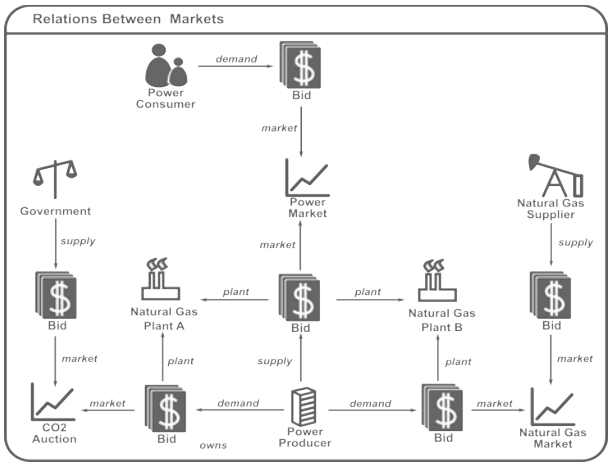

Fig.3. Relations between markets and agents are formed via bids and offers. Both bids and offers in the model are labeled as "Bid" and their nature is determined by the type of relationship (demand or supply).

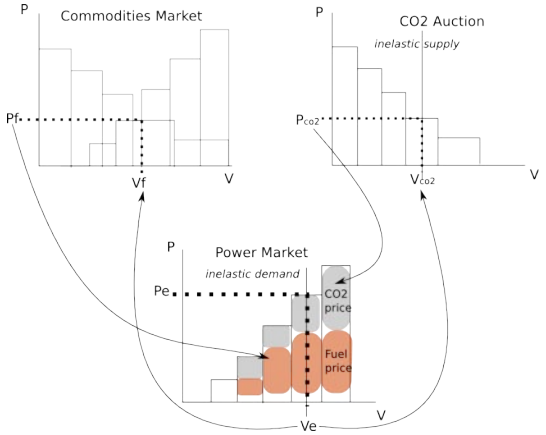

Fig.4. Interactions between markets. The outcomes of the power market determine the inputs to the fuel and CO2 markets and vice-versa.

The diagram above is an abstraction of the interactions between markets. Power producers supply offers into the power market, one offer per power plant. The offer price is the marginal cost of producing one megawatt-hour. The principal components of marginal cost are fuel costs and the costs of CO2 emissions allowances. The prices of the fuels and CO2 allowances are determined in the fuel markets and the CO2 auctions. The

commodities traders submit their offers to sell fuel and the government sets the volume of the CO2 allowance auction. The market clearing prices are incorporated in the offers to the power market, but the power market volume and the power plant portfolio determine the bid volumes submitted to the fuel markets and the CO2 auction. With these simple relations complex patterns emerge within a simulation. The different bidding and investment strategies of the power producers add to the unpredictability of the simulation outcomes.

The following passages will discuss a subsection of the model in more detail to exemplify the workings of AgentSpring.

4.2 Defining Agents

In order to define an agent in AgentSpring one has to declare the `Agent` interface provided by the framework. It is also necessary to annotate the class with `@NodeEntity` annotation that will ensure that object is stored in the graph database.

```
@NodeEntity
public class PowerProducer extends AbstractAgent implements Agent {
....
}
```

4.3 Defining Artifacts

In order to define an artifact (a thing that is not an agent) in AgentSpring, such as a power plant, it is sufficient to provide the `@NodeEntity` annotation. The example below also show how to define relations to other artifacts and agents. In order to define a relation it is sufficient to annotate the class member with `@RelatedTo` annotation and specify the name and the direction of the relationship.

```
@NodeEntity
public class PowerPlant {
  @RelatedTo(type = "technology", direction = Direction.OUTGOING)
  private PowerGeneratingTechnology technology;
  @RelatedTo(type = "owns", direction = Direction.INCOMING)
  private PowerProducer owner;
}
```

The above code is better explained by the diagram below.

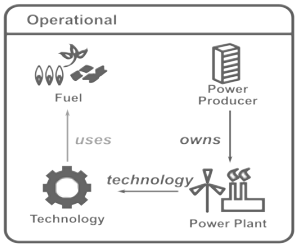

Fig.5. Relations between agents and artifacts.

4.4 Defining Queries

The queries are defined within Java interfaces annotated with with `@Repository` annotation and methods annotated with the `@Query` annotation containing the code of the query. Please see appendix B for the code of this example. It might look unconventional, but it is a simple, brief and a powerful method to express query logic. The query is expressed in Cypher query language. The method `findPowerPlantsByOwner` will return all power plants owned by the power producer supplied as the argument to the query. The name of the relation between the `PowerProducer` and `the PowerPlant` is specified in the square brackets and the direction is specified by the symbol `>`. This is consistent with the relationship definition within the PowerPlant class (see above).

The next two queries in the code are examples of calculations. The diagram on the next page shows the graph defining relations between the cash flows, agents and markets. Each cash flow object has a reference to a power plant that it is associated to. The queries find the the cash flow nodes that are related to the `agent` and the `plant` provided as parameters and sum up their `amount` properties.

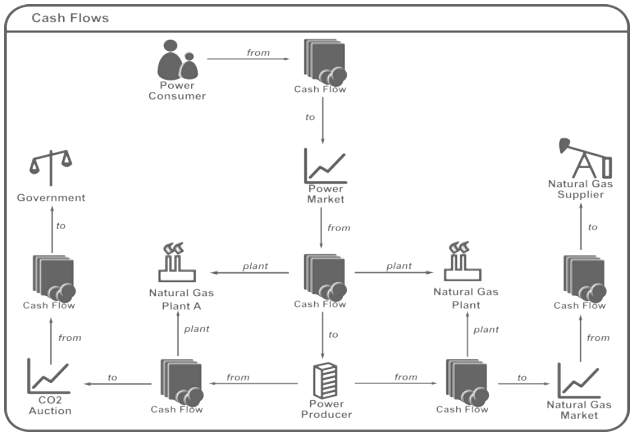

Fig.6. Cash flows between agents and markets.

4.5 Defining Behavior

Agent behavior is composed out of scripts. A script is a separate class that implements the Role interface provided by the framework. It is also annotated with `@ScriptComponent` annotation. The annotation can take parameters that determine the order of script execution in the simulation schedule. The modeler has to implement a single method “act” that contains the logic of the agent's behavior. The example in the appendix A demonstrates an investment script. In the example the agent iterates through its existing power plants and selects the type of the power plant (`PowerGeneratingTechnology`) that proved to be the most profitable.

The script demonstrates the use of graph query methods already discussed. The method `findPowerPlantsByOwner` returns a list of all power plants owned by the agent. Iterating through the list, the return on assets (ROA) score is calculated and the `PowerGeneratingTechnology` with the highest score is selected. ROA is calculated by summing up the cash flows received and paid by the agent per year, dividing it by the capital invested.

5 Conclusions

As agent-based modeling is applied to model and simulate socio-technical systems researchers find themselves in need of tools that would better support their goals. Commonly, the socio-technical systems are distinguished as having many heterogeneous actors and artifacts embedded in a network of versatile relationships. The authors of this article present a new ABM framework – AgentSpring – specifically designed to support modeling of such systems.

AgentSpring introduces a few novelties into the modelers toolkit. Firstly, it uses graph data structure to describe the model and store the simulation data. It enables the modeler to use multiple graph query languages to perform analyses of the simulation outcomes. Queries are also integral to the decision making of the agents who use them to asses the state of their environment. Secondly, the framework allows for modular composition of agent behavior. Such modularity aids the understanding of the model logic and allows the model to scale in complexity. Finally, AgentSpring features a rich web interface allowing to run the simulation and visualize the simulation results on-line. It also allows AgentSpring to seamlessly integrate with other tools in the modelers toolkit, such as R statistical package.

AgentSpring is already being used by multiple projects and institutions. It is open-source and continuously developed. In order to use it or contribute to the development, please find it at http://github.com/alfredas/AgentSpring.

Acknowledgments. This research is funded by the Energy Delta Gas Research (EDGaR) programme[7]. The focus of the interdisciplinary EDGaR program is to understand the challenges presented by the transition of the gas-based energy system to a more liberal, secure, affordable and sustainable state.

This paper as a part of a broader research commitment to understand and simulate the gas market interactions with other markets, infrastructures and industries. In the future the model presented in this paper will be combined with existing electricity market models to explore the inter-market relationships and complexities.

7http://www.edgar-program.com/

References

Angles, R. and Gutierrez, C. (2008), Survey of graph database models, *ACM Computing Surveys (CSUR)*. 40

Bondy, J.A. and Murty, U.S.R. (1976). *Graph theory with applications*, MacMillan, London.

Chappin, E. J. L. (2011). *Simulating Energy Transitions*, PhD thesis, University of Technology, Delft, the Netherlands

Chmieliauskas, A., Chappin, E., Nikolic, I. and Dijkema, G. (2012). New methods in analysis and design of policy instruments, in A. V. Gheorghe (ed.), *System of Systems*, Intech.

Eifrem, E. (2009). Neo4j - the benefits of graph databases, *no: sql (east)*.

Farmer, J. D. and Foley, D. (2009). The economy needs agent-based modelling, *Nature* **460**(7256): 685–686.

Hursch, W.L. and Lopes, C.V. (1995). *Separation of concerns*, Citeseer.

Johnson, R., Hoeller, J., Arendsen, A. and Thomas, R. (2009). *Professional Java Development with the Spring Framework*, Wiley-India.

Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model-view controller user interface paradigm in smalltalk-80, *J. Object Oriented Program.*1(3): 26–49.

Nikolic, I. (2009). *Co-Evolutionary Method For Modelling Large Scale Socio-Technical Systems Evolution*, PhD thesis, Delft University of Technology. ISBN 978-90-79787-07-4.

Ottens, M., Franssen, M., Kroes, P. and Van De Poel, I. (2006). Modelling infrastructures as socio-technical systems, *International Journal of Critical Infrastructures* **2**(2–3): 133–145.

Schank, R.C. and Abelson, R.P. and others (1977). *Scripts, plans, goals and understanding: An inquiry into human knowledge structures*, Lawrence Erlbaum Associates Hillsdale, NJ

Skiena, S.S. (1998). *The algorithm design manual,* Springer.

Appendix A. Investment Script.

```
@ScriptComponent
public class InvestInPowerGenerationTechnologiesRole extends
AbstractRole<PowerProducer> implements Role<PowerProducer> {

@Autowired
PowerPlantRepository plantRepository;

public void act(PowerProducer producer) {
  PowerGeneratingTechnology bestTechnology = null;
  double maxRoa = 0;
  for (PowerPlant plant : plantRepository.findPowerPlantsByOwner(producer)) {
    double revenue = plantRepository.calculateCashFlowsForPowerPlantToAgent(owner,
plant);
    double costs = plantRepository.calculateCashFlowsForPowerPlantFromAgent(owner,
plant);
    double averageAnnualIncome = (revenue - costs) / (getCurrentTick() + 1);
    double capital = plant.getInvestedCapital();
    double roa = averageAnnualIncome / capital;
    if (plantRoa >= maxRoa) {
      bestTechnology = plant.getTechnology();
      maxRoa = plantRoa;
    }
  }
  buildPlant(bestTechnology);
}
....
}
```

Appendix B. Query Repository.

```
@Repository
public interface PowerPlantRepository extends GraphRepository<PowerPlant> {

@Query("start owner=node({owner}) match (owner)-[:owns]->(plant) return plant")
Iterable<PowerPlant> findPowerPlantsByOwner(@Param("owner") PowerProducer owner);

@Query("start plant=node({plant}), agent=node({agent}) match (agent)<-[:to]-(flow)-
[:plant]->(plant) return sum(flow.amount)")
double calculateCashFlowsForPowerPlantToAgent(@Param("agent") PowerProducer agent,
@Param("plant") PowerPlant plant);

@Query("start plant=node({plant}), agent=node({agent}) match (agent)-[:from]->(flow)-
[:plant]->(plant) return sum(flow.amount)")
double calculateCashFlowsForPowerPlantFromAgent(@Param("agent") PowerProducer agent,
@Param("plant") PowerPlant plant);
}
```

Appendix C. Snapshot of AgentSpring User Interface.

Appendix D. Visualization of a Running Simulation. Different color represent different relations between agents and artifacts (around 650000 relations).