DRAMS – A DECLARATIVE RULE-BASED AGENT MODELLING SYSTEM

Ulf Lotzmann
Institute of Information Systems Research
University of Koblenz
Universitätsstraße 1, Koblenz 56070, Germany
E-mail: ulf@uni-koblenz.de

Ruth Meyer
Centre for Policy Modelling
Manchester Metropolitan University
Aytoun Building, Manchester, M1 3GH, U.K.
E-mail: ruth@cfpm.org

KEYWORDS
Agent-based Social Simulation, Rule Engine/Production System, Declarative Modelling, Policy Modelling.

ABSTRACT

This paper presents a snapshot of the Declarative Rule-based Agent Modelling System (DRAMS), which is currently being developed with the goal of providing declarative rule engine functionality to agent-based simulation models. While the incentive for this effort is to allow stakeholders in policy modelling activities to better understand and, hence, to be more deeply involved in modelling and simulation of policy processes, the approach chosen appears to be valuable for many other areas.
After a short literature review on similar approaches, the paper briefly outlines the FP7 project OCOPOMO, the context in which DRAMS is developed, in order to underpin some of the requirements and related design decisions presented subsequently. On the basis of this information, the software architecture is described in some detail, followed by a few notes about implementation and user interfaces. Finally, a small "toy" model illustrates the use of DRAMS.

INTRODUCTION

Artificial intelligence of software agents in social simulation is often represented by rules. While there are many different ways to specify and process these rules, all approaches have in common that individual agents are equipped with their "own" set of rules, where rule processing is triggered by events occurring within some kind of environment and perceived by the agent (some period of time has elapsed, an obstacle appeared in an agent's direction of movement, a new fact has been added to some data base etc.). The result of a rule execution might then be an action, changing the state of the environment. Such rules can be represented for example by stochastic decision trees (as used e.g. for normative simulations in EMIL-S; EMIL 2009), or by deterministic condition-action rules (like e.g. "if event1 and not event2 then action1").
In the latter case, rule processing is often done by simple production systems (Gilbert and Troitzsch 2005), but for models exceeding a certain degree of complexity the use of more sophisticated technologies is indicated. This can be declarative rule engines, a technology widely applied in expert systems, but so far not very common in agent-based simulation.
The fact that declarative rule engine technology has high potential for the field of agent-based modelling can be demonstrated by the following two examples:

- The Smalltalk-based tool SDML (Strictly Declarative Modelling Language, Moss et al. 1998) has been used for water demand management models within the FP5 project FIRMA (Freshwater Integrated Resource Management with Agents, Barthelemy et al. 2001).
- A model of the social impacts of HIV/AIDS in villages of a rural province in South Africa was created in the context of the FP6 project CAVES (Complexity, Agents, Volatility, Evidence and Scale; Alam et al. 2007; Moss 2008) using the Java-based rule engine JESS (http://www.jessrules.com).

This technology has also been chosen to be applied for model development within the FP7 project OCOPOMO, which focuses on open collaboration in policy modelling (Wimmer 2011). One of the major goals of this project is to show how policy modelling processes can be made explicit and better understandable in order to allow deeper involvement (via open collaboration technologies) of various kinds of stakeholders beyond modelling experts. For this, it is inevitable to find a modelling approach which, on the one hand, uses a language close to the everyday language of policy domain experts, and which, on the other hand, is powerful enough to cope with the degree of complexity inherent in serious policy problems.
As a technical outcome of this project, an integrated toolbox is to be implemented, which supports (and generates traceable documentations of) the transition process from unstructured raw data (e.g. scenario documentations, scientific papers, statistical reports) via qualitative data analysis, resulting conceptual models, through to formal policy models. For the final transition step – the development of formal models – the declarative modelling paradigm appears to be the best technique due to its more natural representation of the human approaches to reasoning and problem solving.
During the first stage of the project, several examples of existing rule engine software including JEOPS (Santos da Figueira Filho and Lisboa Ramalho 2000) and JESS have been evaluated, mainly with respect to several

Proceedings 25th European Conference on Modelling and Simulation ©ECMS Tadeusz Burczynski, Joanna Kolodziej Aleksander Byrski, Marco Carvalho (Editors)
ISBN: 978-0-9564944-2-9 / ISBN: 978-0-9564944-3-6 (CD)

given requirements like licence model (open source), the possibility to be integrated in a Web-based toolbox and, most importantly, the ability to efficiently cope with intensely dynamic data during simulation runs.
Most of the existing systems are optimized for expert systems, i.e. frequent requests on fact bases with rather static data configurations. They employ a variant of the Rete algorithm (Forgy 1982), fixedly matching facts to rules by compiling the rule base in order to speed up performance. If a fact is added, modified or deleted, the rule base is recompiled. In a simulation environment where the fact bases represent the working memory of agents and are changing frequently, recompiling becomes more or less continuous and, therefore, time-consuming as well as highly memory-intensive. For this reason, it was decided to initiate the development of a new tool optimized for the special purposes of simulations, integrating properties of tools like SDML and JESS. In the following sections the development state of this software called DRAMS (Declarative Rule-based Agent Modelling System) is presented.

OVERVIEW OF DRAMS

A rule engine is a software system that basically consists of:

- A fact base, which stores information about the state of the world in the form of facts. A fact contains a number of definable data slots and some administrative information (time of creation, entity that created the fact, durability).
- A rule base, which stores rules describing how to process certain facts stored in fact bases. A rule consists of a condition part (called left-hand side, LHS) and an action part (called right-hand side, RHS).
- An inference engine, which controls the inference process by selecting and processing the rules which can fire on the basis of certain conditions. This can be done in a forward-chaining manner (i.e. trying to draw conclusions from a given fact constellation) or backward-chaining manner (i.e. trying to find the facts causing a given result).

DRAMS is designed as a distributed, forward-chaining rule engine. It equips an arbitrary number of agent types with type-specific rule bases and initial fact base configurations. For each agent type, an arbitrary number of agent instances (objects) with individual fact bases can be created. All individual fact bases are initialized according to the agent type configuration, but may be adapted individually.
There is also a shared global fact base, containing “world facts”, e.g.

- a (permanently updated) fact reflecting the current simulation time,
- one fact for each agent instance present in the “world”, providing some information (e.g. reference ID) about the agent,
- model-specific environmental data, and
- (public) inter-agent communication messages.

Heart of the inference engine is the rule schedule, an algorithm deciding which rules to evaluate and fire at each point of time. The pseudo code in Figure 1 shows the basic structure of a possible algorithm. In order to decide (for each fact base configuration without recompiling the rule base) which rules to evaluate for which agent instances, the scheduler relies on a data-rule dependency graph. This is constructed once from all specified rules and initially available data; the graph does not change unless rule bases are modified. As to detecting fact base modifications, the schedule keeps track of all (writing) fact base operations. A more detailed description of this method is given in the following section.

```
processSchedule(time t){

   while new facts are available at time t
   loop

      find all agent instances for which
      new facts are available;

      foreach agent instance
      loop
         find all rules for which new facts
         are available at time t;

         foreach rule
         loop
            evaluate LHS;

            if evaluation result==true then
               execute RHS;
               // e.g. generate new facts
            end if;

         end loop;

      end loop;

   end loop;

}
```

Figure 1: The schedule algorithm.

At each point of time, the rule processing within an agent (intra-agent process) is performed for all possible rules. At first, the conditions of a rule are checked, i.e. the LHS is evaluated. Each LHS consists of one or many (LHS) clauses, pertaining to the following basic categories:

- Clauses for retrieving data from fact bases. These operations are similar to data base queries, where as a result a set of facts (with 0, 1 or many elements) is retrieved. Each retrieved fact is called an instance of this clause, and all subsequent clauses of the LHS have to be evaluated for all instances. Thus, this clause type is spanning an evaluation tree. If one or more facts are retrieved, the evaluation result for this clause is true, otherwise false. In the latter case, the evaluation of this tree branch is terminated.

- Clauses which test whether data from the retrieved facts hold for specified conditions. If such a test fails, the evaluation of this tree branch is terminated.

The set of leaves of the evaluation tree is considered a set of possible input data configurations for firing the RHS of the rule. The RHS consists of one or many (RHS) clauses with the purpose of executing fact base operations (adding or removing a fact) or other actions (e.g. printing a statement to a log).

Accordingly, the expressiveness of the system is determined mainly by the number and capabilities of available clause types. In DRAMS, the following functionality is available for the LHS of a rule:

- Data from fact bases can be obtained either by retrieve or by query clauses. In both cases, a query on (in principle any existing) fact base is performed, and a number of facts, for which the slot values specified in the query match and optional time-related conditions hold, are returned. In the case of retrieve clause, these facts are used to create a corresponding number of instantiations, and for each instance a number of requested variables are bound with specified slots. In the case of query clause, only one instance is created, and a list of retrieved facts is bound to a result variable.
  These two clause types are representative for the first category defined above, whereas the following LHS clause types belong to the second category.
- A unary BIND operator, which binds a specified variable with the evaluation result of an expression. The expression can be another single (already bound) variable, or a more complex arithmetic expression (with a standard repertoire of math operations, including generation of random numbers).
- A full set of binary logical operators, where the operators can be any expressions allowed for the BIND operator. The result is either bound to another variable or, alternatively, used as clause evaluation result.
- A set of LIST operators, including generation and modification of lists, counting and extracting of list elements, and several accumulator operations (sum, avg, min, max etc.).
- A set of SET operators, including creation and modification of sets, number or existence of elements, as well as union and intersection of two sets.
- A NOT clause, inverting the evaluation result of the specified inner clause. The inner clause can be any other LHS clause.
- A COMPOSITE clause, which can be seen as an encapsulated LHS with its own variable name space. For processing the specified inner clauses, the evaluation mode can be chosen between AND, OR and XOR.

RHS clauses dedicated to perform actions comprise:

- Asserting new facts to (in principle any existing) fact bases.
- With some restrictions, retracting existing facts from (in principle any existing) fact bases.

There is one special clause defined which can be part of both the LHS and RHS, providing two kinds of actions:

- printing formatted text (including values of variables) to a log (either to a console window or to a file);
- calling a method on the underlying model part; the peculiarity of this functionality is explained in more detail in the following section.

An important aspect of any multi-agent simulation system concerns the means by which agents can communicate with each other (inter-agent process). Technically, DRAMS provides three options:

- communication via the global fact base in a blackboard-like manner;
- writing facts to fact bases of other (remote) agents; this can be interpreted as the way humans typically communicate with each other, via speech or written messages;
- reading facts from fact bases of remote agents; this is conceptually similar to mind-reading, and should thus be avoided in most cases, but can be useful to find out (public) properties of another agent

To facilitate creating a model using any of the described features of DRAMS, an appropriate user interface ought to be available. This is discussed in the next but one section, while the following section sketches the software architecture of DRAMS.

ARCHITECTURE AND IMPLEMENTATION

Figure 2 shows the overview class diagram of the DRAMS core. Main class and control hub is the singleton `RuleEngineManager`. It manages all instances of class `RuleEngine` and controls the interactions between the three core components “Scheduler”, “Data Management” and “Rule Management”.

The “Data” component represents the fact base functionality and is responsible for storing all facts to be processed by any rule. Both `RuleEngine` and `RuleEngineManager` are using the class `FactBase` for this purpose. A fact base contains an arbitrary number of entries (class `FactBaseEntry`), which serve as fact templates for the actual facts to be stored. A fact template specifies the number and types of data slots for a specific fact. For each of the fact templates an arbitrary number of facts (class `Fact`) can be stored. A special type of fact is the so-called “shadow fact” (class `ShadowFact`), which does not store actual data but gains its content from external sources (such as Java objects; concerning utilisation of this feature, see last

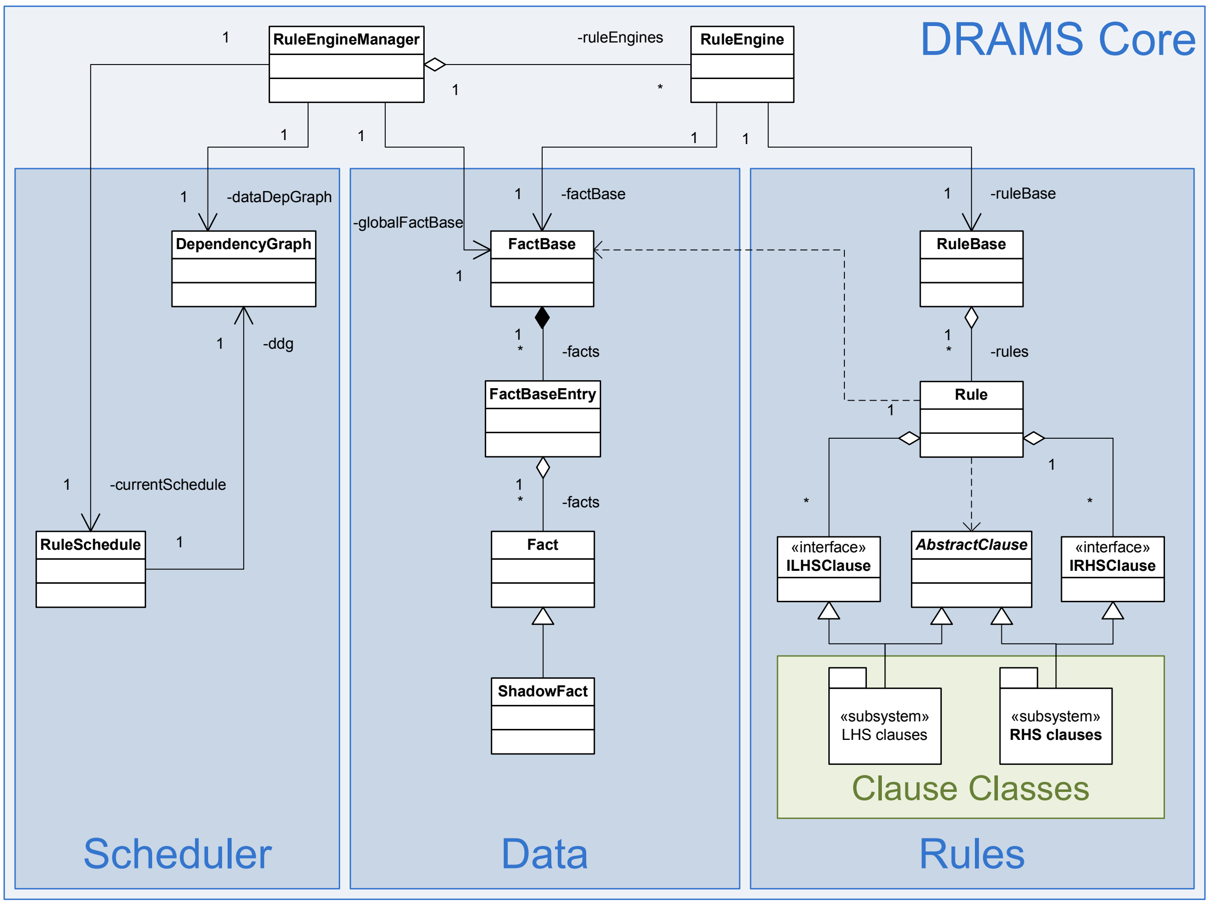

Figure 2: Overview class diagram.

paragraph of this section). Besides operations like inserting to or removing facts from a fact base, the key functionality of `FactBase` is to process queries, i.e. to retrieve all facts stored in the fact base that match all conditions specified by the query. Clauses for this purpose form the basic element of every rule.

The rules specified within a rule engine are managed by the "Rules" component. Similar to the fact base, there is a `RuleBase` class available, which stores an arbitrary number of rules (represented by class `Rule`). A rule consists of several LHS clauses, constituting the condition part. All such clauses must implement the `ILHSClause` interface. The action part of a rule is composed of RHS clauses, for which the interface `IRHSClause` is designated. The abstract class `AbstractClause` provides some key functionality and must be extended by any clause class. Each of the LHS and RHS clause functions described in the previous section is implemented in a separate class, in Figure 2 subsumed under subsystems "LHS clauses" and "RHS clauses".
The third component, the "Scheduler" is responsible for the dynamic aspects of the rule engine, as described by the schedule algorithm presented in the previous section. DRAMS employs a data-driven rule schedule implementation, which relies on a specific dependency graph (class `DependencyGraph`). This directed graph describes both which fact templates are evaluated by any rule (linking facts to rules), and for which fact templates writing operations might be triggered by a rule (linking rules to facts). The basic concept behind the scheduling algorithm can be outlined as follows: all rules are scheduled for which new facts are available at a given point of time. The information about newly asserted data is collected by the `RuleEngineManager`, which in turn obtains this information from `FactBase`.
This data-driven scheduling mechanism is one of the major benefits of DRAMS with regard to applicability for agent-based simulations. While all available rule engine systems currently supported and known to us are optimized to efficiently perform requests on large but static fact bases (which is greatly desired in e.g. expert systems), changes in fact bases often require time-consuming "recompiling" operations. DRAMS with its presently naive, i.e. non-optimized rule evaluation algorithm offers already suitable rule execution speed for fact base sizes and rule quantity typical for mid-sized simulation models (with up to a few thousand agent instances). Furthermore, frequently applied assertions of new facts are processed extremely efficiently. Although only partly implemented so far, fast algorithms are conceivable also for other types of operations, such as retraction of facts (dependent on the intended formal logical model) or creation of new rules.

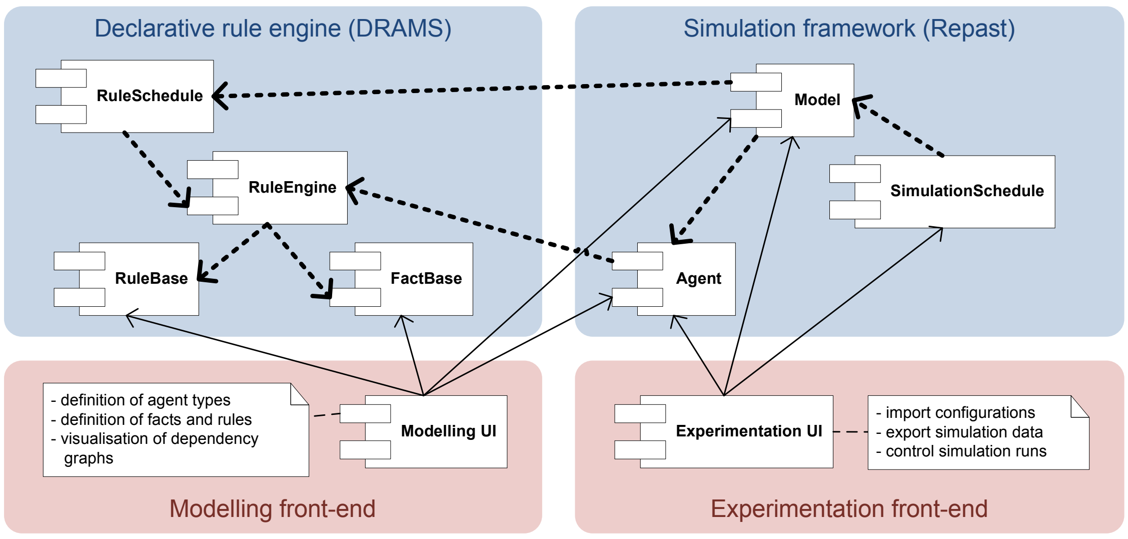
Declarative rule engine (DRAMS)
Simulation framework (Repast)
RuleSchedule
Model
RuleEngine
SimulationSchedule
RuleBase
FactBase
Agent
- definition of agent types
- definition of facts and rules
- visualisation of dependency graphs
Modelling UI
Experimentation UI
- import configurations
- export simulation data
- control simulation runs
Modelling front-end
Experimentation front-end

Figure 3: Components and dependencies of the Repast/DRAMS simulation system.

DRAMS implements two operation modes for `RuleSchedule`.
The active time mode is used for time-driven simulation runs, where the (usually equidistant) time steps are set by a simulation scheduler (which controls the flow of simulation time, not to be confused with the rule schedule!).
The passive time mode provides means for event-driven simulations. No external time step is fed in by the simulation controller, instead the rule schedule proceeds with the next point of time at which some system state change is scheduled (e.g. indicated by deferred fact assertions).

When using passive time mode, complete simulation models might be specified and run solely with the functionality provided by DRAMS. However, if some kind of environment beyond fact base entries is to be included into a model designed for discrete time step operation, DRAMS only takes care of agent deliberation abilities, while the environment is "outsourced" to an external simulation tool. Since DRAMS is implemented in Java, in principle any Java-based simulation tool is qualified for this purpose. For example, DRAMS can be used as a rule engine extension for Repast (North et al. 2006). In this case, a Repast model class maintains a link to the DRAMS rule schedule, and each Repast agent instance maintains its own rule engine object (see Figure 3; the dependencies between and within DRAMS and Repast are depicted by dashed arcs). In order to guarantee seamless integration, DRAMS provides several means for accessing Java objects (e.g. shadow facts, Java action clauses). An overview on how to create such a model is presented in the following section, after a discussion on user interface aspects.

USER INTERFACE

For any simulation system, at least two types of typical use cases can be distinguished, for which specifically designed user interfaces should be provided:

- a modelling user interface, facilitating to the writing and testing of model code, and
- an experimentation user interface, allowing the user to configure, control and analyse simulation runs.

In the case of integrated Repast/DRAMS models, the involved components and anticipated dependencies are shown in Figure 3 (solid arcs). Obviously, the experimentation user interface as well as the Repast part of the modelling interface can make use of already available features supplied by Repast. Although it is desirable to have an Integrated Development Environment (possibly based on Eclipse; http://www.eclipse.org/eclipse/) for creating the DRAMS part of a model, it is rather simple to create models with the few basic instruments characterized below (which represent the current implementation state).
The entire process of designing and running a simulation model, using RepastJ 3.1 as simulation platform and DRAMS as deliberation extension, can be structured into the five steps described below. These steps are illustrated by a primitive example, modelling the process of a customer finding a shop that sells a certain product. There are a number of shops, each offering a specific set of products sold at a price chosen randomly within a range. Consumers have the intention to purchase a specific product and are willing to spend a certain amount of money for it. As a first step, they have to find all shops which offer the product at a price below that threshold.

1. A Repast model class must be defined, which creates all agent instances (for which the desired agent types have to be defined, see step 2), initialises the global fact base and handles the Repast time steps by triggering the rule scheduler. An abstract model super class providing the DRAMS related code is available, thus only model related aspects have to be added.

```
( defrule "find_shops"
   // LHS
   // retrieve all agents of type "Shop", bind name and instance of shop to variables
   ( global::$AGENT$ (agent_type "Shop") (name ?name) (ruleengine_instance ?shop_id) )

   // retrieve shop products;
   // only care about shops that have apples in stock, bind price to variable
   ( "Shop"@?shop_id::product (name "apples") (unit_price ?price) )

   // only care about prices below 2.20 Euros
   ( less_than ?price 2.20 )

   =>
   // RHS
   // print all found shops to the log
   ( print "Shop ?name sells apples for ?price Euros." )

   // for each found shop assert a fact in the consumers fact base
   ( assert ( apple_shop (shop ?shop_id) (price ?price) ))
)
```

Figure 4: Rule specification.

```
*********************** SCHEDULE AT TIME 0.0*******************

-------- TASKS FOR TIME 0.0 ---------
  ---> TASK NO 0
    * ACTION ActionRuleFire FOR RULE 'find_shops'
         REASON: NEW FACTS AVAILABLE FOR GLOBAL::$AGENT$; Shop::product;
         INSTANCES: Consumer (1);

*********************** RULE OUTPUT AT TIME 0.0*******************

[consumer-1.find_shops] Shop shop-19 sells apples for 2.09 Euros.
[consumer-1.find_shops] Shop shop-17 sells apples for 2.19 Euros.
[consumer-1.find_shops] Shop shop-1 sells apples for 2.19 Euros.
Time to run: 0.032 seconds

******************* BEGIN: DUMP OF FACTBASE OWNED BY 'consumer-1' *******************
-------- FACTS FOR NAME 'apple_shop' ---------
(apple_shop (price 2.09) (timeStamp 0.0) (shop_id dffef662-b62e-4f91-9c85-f32936fcd9c9) (permanent false) … )
(apple_shop (price 2.19) (timeStamp 0.0) (shop_id 4ac0802d-846a-4ffb-9c6a-1d490a59de68) (permanent false) … )
(apple_shop (price 2.19) (timeStamp 0.0) (shop_id 51cf255c-6c62-45f1-a75c-5973f894bd4a) (permanent false) … )
******************* END: DUMP OF FACTBASE OWNED BY 'consumer-1' *******************
```

Figure 5: Extract from a simulation run log.

2. Classes for all designated agent types (in the example `Producer` and `Consumer`) must be created. Besides environment-related functionality, code must be prepared for initialising the rule engine and for defining the agent (type and instance) specific fact templates, facts and rules. Similar to the model super class, an appropriate agent super class is delivered with DRAMS.
3. For each agent class, code for the declarative model part has to be written. Firstly, the fact templates have to be specified, after that the initial facts can be asserted to the fact base, and finally the rules can be written (example for `Consumer` in Figure 4). In general, all DRAMS related elements can be either specified by directly instantiating the appropriate Java objects (as shown for fact definitions), or by using the DRAMS modelling language (as shown for rule definition; in this case the Java objects are created by a parser).
4. The declarative model part can be checked for consistency, using a visualisation of the automatically generated dependency graph. Figure 6 shows the graph for the example.
5. The model can be executed using the Repast user interface. Figure 5 shows a possible output log (which includes a schedule trace, statements printed by the rule and a fact base dump).

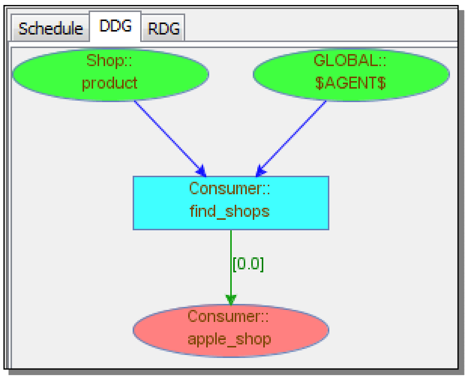

Figure 6: Screenshot of a data-rule dependency graph.

CONCLUSIONS AND OUTLOOK

The declarative rule-based agent modelling system DRAMS described in this paper has been designed with the aim to fulfil the most important requirements of a rule engine extension for agent-based social simulation purposes. While all core functionalities necessary for developing serious policy models have been completely implemented (e.g. the three pilot use case models for the OCOPOMO project are based on DRAMS; Moss et al. 2011), there is still rather a long way to go in order to achieve the goal of creating a versatile and easy-to-use declarative modelling environment. This challenge has been accepted by the original authors, together with a steadily growing community of developers and users. The most important tasks to tackle in the near future are:

- To provide a comfortable integrated user interface which allows for interactive and graphic definition of facts and rules, supported by an online debugging facility. It should be possible to import data from external qualitative text analysis tools (the development of such a tool is also a focus of OCOPOMO; Wimmer 2011; Bicking et al. 2010).
- To include meta-rules, i.e. rules which add new rules or change existing ones.
- If possible, to integrate DRAMS with declarative modelling approaches from several other groups across Europe.

The stable versions of DRAMS will be published under an open source licence model. Exact conditions will be decided during mid 2011.

ACKNOWLEDGEMENTS

OCOPOMO is co-funded by the European Commission under the 7th Framework Program. DRAMS has been developed in OCOPOMO. This publication reflects the view only of the authors and the project consortium, and the Commission cannot be held responsible for any use, which may be made of the information contained therein.
The authors would like to thank Prof. Scott Moss, Prof. Klaus G. Troitzsch and Dr. Michael Möhring for their valuable input to the development of DRAMS and their helpful comments on the first draft of this paper.

REFERENCES

Alam, S.J.; R. Meyer; G. Ziervogel and S. Moss. 2007. “The Impact of HIV/AIDS in the Context of Socioeconomic Stressors: an Evidence-Driven Approach”. *Journal of Artificial Societies and Social Simulation* 10, No. 4, 7.

Barthelemy, O.; S. Moss; T. Downing and J. Rouchier. 2001. “Policy Modelling with ABSS: The Case of Water Demand Management”. CPM Report No. 02-92. Centre for Policy Modelling, Manchester Metropolitan University, Manchester.

Bicking, M.; P. Butka; C. Delrio; V. Dunilova; … and M.A. Wimmer. 2010. “D1.1 Stakeholder Identification and Requirements for Toolbox, Scenario Process and Policy Modelling”. Deliverable 1.1. FP7 Project OCOPOMO (Open Collaboration for Policy Modelling).

EMIL. 2009. “EMIL-T”. Deliverable 5.1. FP6 Project EMIL (Emergence in the Loop: Simulating the Two-Way Dynamics of Norm Innovation).

Forgy, C. 1982. “Rete: a Fast Algorithm for the Many Pattern/Many Objects Pattern Match Problem”. *Artificial Intelligence*, 19, No. 1 (Sept.), 17-37.

Gilbert, N. and K.G. Troitzsch. 2005. *Simulation for the Social Scientist.* 2nd edition. Open University Press, McGraw-Hill, Maidenhead.

Moss, S.; H. Gaylard; S. Wallis and B. Edmonds. 1998. “SDML: A Multi-agent Language for Organizational Modelling”. *Computational and Mathematical Organization Theory* 4, No. 1, 43-69.

Moss, S. 2008. “Simplicity, Generality and Truth in Social Modelling”. CPM Report No. 08-187. Centre for Policy Modelling, Manchester Metropolitan University, Manchester.

Moss, S.; R. Meyer; U. Lotzmann; M. Kacprzyk; … and C. Pizzo. 2011. “D5.1 Scenario, Policy Model and Rule-Based Agent Design”. Deliverable 5.1. FP7 Project OCOPOMO (Open Collaboration for Policy Modelling).

North, M.J.; N.T. Collier and J.R. Vos. 2006. “Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit”. *ACM Transactions on Modeling and Computer Simulation* 16, No. 1 (Jan.), 1-25.

Santos da Figueira Filho, C. and G. Lisboa Ramalho. 2000. “JEOPS — The Java Embedded Object Production System”. In *Advances in Artifical Intelligence*, Springer, pp.53-62 (Lecture Notes in Computer Science; 1952).

Wimmer, M.A. 2011. "Open Government in Policy Development: Form Collaborative Scenario Texts to Formal Policy Models". In: Natarajan, R.; Ojo, A. (Eds.): *Distributed Computing and Internet technology. 7th International Conference, ICDCIT 2011.* LNCS # 6536, Springer-Verlag Berlin Heidelberg, pp. 76 - 91.

AUTHOR BIOGRAPHY

ULF LOTZMANN obtained his diploma degree in Computer Science from the University of Koblenz-Landau in 2006. Already during his studies he has participated in development of several simulation tools. Since 2005 he has specialized in agent-based systems in the context of social simulations and is developer of TRASS and EMIL-S. He is involved in the FP7 project OCOPOMO (Open Collaboration for Policy Modelling) and is responsible for the DRAMS development in this context. Currently he is doctoral student at the University of Koblenz-Landau. His e-mail address is ulf@uni-koblenz.de.

RUTH MEYER graduated in computer science and biology from Hamburg University, Germany, and is in the final stages of completing her PhD in agent-based simulation. She is currently a Research Associate at the Centre for Policy Modelling, where she has been involved in the projects CAVES, EMIL and OCOPOMO. Her e-mail address is ruth@cfpm.org.