Component-based models and simulation experiments for multi-agent systems in James II

Jan Himmelspach
jh194@informatik.uni-rostock.de

Mathias Röhl
mroehl@informatik.uni-rostock.de

Adelinde M. Uhrmacher
lin@informatik.uni-rostock.de

Faculty of Computer Science and Electrical Engineering
University of Rostock
18059 Rostock, Germany

ABSTRACT

The architecture of the modelling and simulation framework James II facilitates its reuse for a broad range of applications, including agent-based modelling and simulation. Simulation studies can be done by using component based models, which may have been defined in an external IDE and saved in an standardized way. We created a customisable middleware for James II, which can be easily extended to read various experiments and models from different sources, to dynamically instrument the created structures, to execute models on different hardware infrastructures efficiently, to store simulation data into diverse data sinks, which can be used to create specialised IDEs, and which can be fully integrated into other applications.

1. INTRODUCTION

In multi-agent research, particularly in bridging the gap between conceptual modelling and implementation, simulation has played an important role from the very beginning and since then features central for those simulation environments to support an easy evaluating of multi-agent systems has been subject of discussion [10]. Early approaches of utilising simulation in the context of agent design have been characterised by ad-hoc implementations or by concentrating on one test scenario only, e.g. Tileworld [22]. Over the years the attention has turned towards exploiting state of the art modelling and simulation (m&s) methods, or entire frameworks for agent design. The list of existing simulation systems is rather long, they offer quite diverse features for supporting the design of diverse models. Some of these systems try to make models interoperable, e.g. MÖBIUS [2], some focus on an efficient distributed simulation, e.g. μsik [21], some on specific application areas (e.g. NS/2 [5], SWARM [19]), some on web-based simulation (e.g. D-SOL [14]), and some on certain formalisms or description languages, e.g. JDEVS [6] and SESAM [16]. Several of these systems are extensible, few have a clear separation between (declarative) model and simulator, and even less address the problem of a declarative experiment description.

Especially the translation process, from a declarative model representation to an executable model is handled quite differently by the available systems. Sometimes models are directly created in a general purpose programming language, an approach pursued in Repast, or they are transformed into source code (from a symbolic description), compiled and executed afterwards, others directly interpret the symbolic model. All techniques have advantages as well as disadvantages [9]. On the one hand, the expressiveness of a general purpose programming language is considered to be higher and the resulting code most often faster (but this heavily depends on the skills of the author). On the other hand, model languages may ease the creation of models, may prevent errors, and may even be better reusable. Compiling means speed up, interpretation eases observation and interaction.

Despite all this efforts, so far only few answers to the question on how to transform declarative models into efficient executable code and thus combining the benefits of a declarative, composite model definition with an efficient and sound execution, do exist. Whereas the benefits of an explicit model description appear obvious and most agent-oriented approaches distinguish between model and simulation, one other aspect has achieved too little attention so far: supporting the simulation experiment itself. However, an unambiguous description of experiments is a pre-requisite for a systematic experimentation with multi-agent systems and their repeatability. Again we find declarative descriptions and executions tightly connected. Thus, simulation systems face a set of interrelated challenges with respect to

Modelling research: Modelling research most often deals with the problem of how to describe a model. Thereby the focus is either on how to describe a problem (which most often leads to new formalisms) or on how to make models reusable and exchangeable. Especially the often neglected validation of models is of high importance for getting reliable results by simulation.

Simulation research: The development of simulation algorithms is strongly related to the modelling formalisms used and platforms to be supported. Thereby the achieved results are most often not comparable to results which have been previously achieved because most of these algorithms have strong interrelationships with the frameworks they are embedded in.

Simulation experiments: Quite many simulation systems have been developed for concrete application domains, or even for concrete simulation experiments [19], often by non-experts in the area of modelling and simulation. For being able to execute a broad range of different applications

Jung, Michel, Ricci & Petta (eds.): *AT2AI-6 Working Notes, From Agent Theory to Agent Implementation, 6th Int. Workshop,* May 13, 2008, AAMAS 2008, Estoril, Portugal, EU.

Not for citation

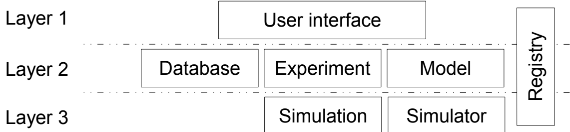

Figure 1: Modules of James II.

a variety of modelling formalisms and platforms have to be supported in an efficient manner and the software must be accessible for non m&s experts. In addition experiments and thus the computed results must be repeatable.

The two aspects (1) how to combine a composable model construction with efficient simulation engines and (2) to pursue the question what is an experiment, and how can it be supported shall form the nucleus of our paper. A plugin-based architecture provides a suitable base for our endeavour. In the following we will show the general experimentation process of James II and how symbolic model descriptions can be integrated. The proposed mechanism allows the integration of any model description language – as long as a so called *ModelReader* (and target model classes) exist in the system – but both can be easily extended as well. This process can be considered as a new way (for the generation of executable models) – symbolic models are read and mapped onto executable model components.

An example of a multi agent-based application will be used to illustrate the applicability and usability of the achieved solution in regards to component based modelling of multi agent systems and the usage of efficient distributed simulation for experimenting with the model.

2. BACKGROUND

A general modelling and simulation framework usable for different applications by various users has to be very flexible: all parts, and even sub parts, of the framework have to be exchangeable. For being reusable in the large James II has been split up into several modules (see Figure 1). Each of these modules realises a part of the functionality required for an m&s framework. These modules are clearly separated from each other – this ensures their interchangeability and reuse. The most apparent one may be the strict distinction between model and simulator. A model cannot directly call simulator functions – this makes it fairly easy to use different simulators, or even to exchange the simulator used during a simulation run – e.g. if another one is assumed to be more efficient. For being able to reuse a model, it is useful to have generic model descriptions which are independent from a concrete simulator implementation and can be converted into executable models on demand (e.g. [1]). These model components, sometimes called building blocks [30], can be described by, e.g., using XML.

The user interface encapsulates model creation, model execution (control + visualisation), data analysis, and so on. Up to now we only have a rudimentary graphical user interface (GUI), i.e. we only have a model view, a simple experiment editor, and very simple model editors. The GUI is only loosely coupled to the remaining simulation system and it is even possible to use James II without a GUI at all (either in a kind of batch mode or interactively) or to embed James II into another application. The GUI has been designed by using the model-view-controller paradigm which allows, e.g., various views on the same model. The basic coupling between a running simulation and the GUI is realised by the Observer-pattern.

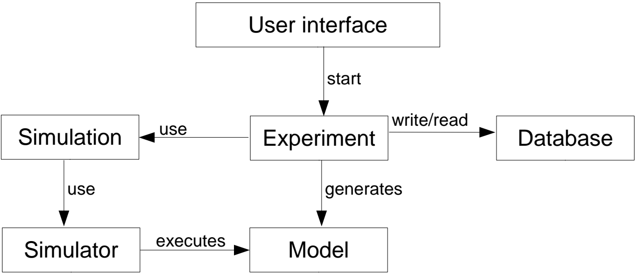

Figure 2: Relationship of the core modules of James II.

The data sink used during a simulation run may have to be able to handle many observations, such that the run can be analysed and visualised afterwards. The database interface makes no restriction in regards to the database to be used – thus everything starting from a plain ASCII file up to a modern database system can be used in principal. The interface is standardised, thus any sink implementing the interface can be used interchangeable for collecting runtime data.

The Model module encapsulates the modelling formalisms supported by James II. All executable models must be descendants of the base class `Model`, all externally visible functions should be placed in an interface on which the simulators can operate. These model classes can be directly used for creating models by coding in Java, for other model sources instances of these classes need to created while reading a model. Models are explicit and they are separated from the execution (simulation algorithms). The computer on which a model is computed does not matter, e.g. the mobility of agents is separated from moving models to other computers – if the latter is done, the reason is load balancing and not model logic.

The simulation module in James II contains the simulation management part. If a simulator executes a model this is called a simulation. In this package methods and classes for simulation setup (simulator selection and creation, partitioning, management of distributed computing resources), run time control (pause, ...) as well as simulation related run time jobs (e.g. load balancing) are located. Simulations in James II can either be executed sequentially on a single machine, several simulations can be executed in parallel on different machines, or simulation can be executed using parallel distributed algorithms. If they are to be executed in a parallel distributed manner, the integrated (and extensible) partitioning and load balancing sub packages come into play.

The Simulators are selected by and embedded into a simulation. This selection depends firstly on the model class of the model to be executed and secondly on other criteria. Currently the list of available criteria is extended by a new "intelligent" criterion, which tries to automatically select the best of the algorithms usable in principal. In James II several simulators for each of the different formalisms can coexist. E.g. currently we have six different simulation algorithms for PDEVS, including sequential and parallel variants [11].

Experiment is the central term in James II. Simulations

may be carried out with a focus either on conducting experiments with models or on experimenting with simulation algorithms. This package provides support for creating flexible experiments with support for a variety of simulation jobs, e.g. for parameter optimisation and validation.

An experiment is defined as the execution of a set of simulation runs for answering a concrete question. The experiment module has to combine the various, previously introduced modules (see Figure 2).

A PlugIn mechanism [13] allows the flexible extension of the simulation framework without the need to modify the code of the core later on. Due to the strict separation between models and simulators, simulation algorithms can be easily exchanged and thus evaluated. This makes the PlugIn mechanism a base for a reliable evaluation of new simulation algorithms. The adaptation of a modelling and simulation framework for certain user groups (especially of the user interface) is crucial for its usability. This adaptation can be easily done by the PlugIn mechanism or by embedding the complete JAMES II core (with all installed plugins) into another JAVA application [18]. Modelling and simulation applications whose individual requirements may even contradict (e.g. the use for demonstration purposes in the field of teaching simulation algorithms versus highly efficient implementations of algorithms for efficient experiments) can coexist in the framework [12, 11].

3. DIFFERENT MODEL SOURCES

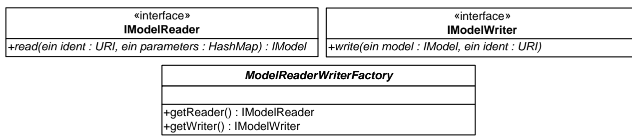

Figure 3: Interfaces and base factory class for reading / writing models.

JAMES II does not only allow the integration of different modelling formalisms / languages but also the usage of any description languages for them. I.e. in JAMES II symbolic and executable models are differentiated and only in the case of models coded in Java both might be the same. Interfaces defining implemented models are used for accessing executable models. This allows different implementations of the classes for executable models, and thereby allows efficient and adoptable model realisations. This is a highly required feature: different models may have completely different characteristics, such that different data structures in combination with different simulation algorithms might prove beneficial. For each description language a special reader has to be designed (a model reader) which maps a model description on instances of executable model classes. Depending on the reader any source (database, code files, ...) can be used for retrieving a model.

The interface, which is provided by the framework and has to be implemented by a model reader is shown in Figure 3. A model reader becomes accessible as a plugin by JAMES II by setting up a simple plugin description file. The plugin type description is given in Figure 4. If an XML plugin file for the ModelReader extension point is found, JAMES II automatically installs the plugin and if later on an experiment definition links to a model readable by the newly defined model reader, this plugin is automatically used.

```
<?xml version="1.0" encoding="UTF-8" ?>
<plugin xmlns="http://www.informatik.uni-rostock.de/
    mosi/cosa/plugintype">
  <id name="model␣reader/writer␣plugins" version="
      1.0" />
  <abstractfactory>james.core.data.model.
      AbstractModelReaderWriterFactory</
      abstractfactory>
  <basefactory>james.core.data.model.
      ModelReaderWriterFactory</basefactory>
  <description>Support of diverse model readers/
      writers.</description>
</plugin>
```

Figure 4: XML based Plugin type description file for model readers and writers (defines an extension point). Each reader/writer plugin must provide a factory which is a descendant of the specified basefactory.

JAMES II uses a "late" reading mechanism. Thus models are not read and instantiated if a new simulation configuration is created by the experiment but if the simulation is about to the started. Thereby the model reader is used on the computer the model must be instantiated on – this reduces network load (in distributed setups), and prevents the central (distributing) instance to become a bottleneck.

In addition to reading models JAMES II provides an interface for the integration of model writers (*ModelWriter*). Based on the interfaces of the model in memory a model description can be written to any supported target. This includes different storages and all description languages for which writers exist.

3.1 Component-oriented Modelling in XML

While models can always be directly described in Java it is possible to describe them by using a special XML-based syntax as well. The XML-based variant eases the import and export of models specified in a standard exchange format. Furthermore, XML descriptions enable us to simply generate and describe a broad range of experiments.

XML handling is based exclusively on entities that are bound to XML Schema Definitions [25]. Schema Definitions mainly define the syntax of an XML document and thereby provide the means for rendering XML documents valid or invalid. JAXB [28] is used to generate Java classes that conform to XML Schema Definitions. Thereby, the bound entities are able to transparently unmarshal XML documents to Java objects and marshal Java objects back to XML documents. This process has to be encapsulated into ModelReader and ModelWriter implementations for integrating the XML descriptions into JAMES II.

Execution of XML model components builds upon the simulators as described above. To this end, the declarative model definitions are automatically transformed to executable ones [25].

Based on XML-descriptions models can be specified in a component-oriented manner [23]. Provided and required interfaces of each model component have to be explicitly specified. Thereby internal details of a model are hidden and direct dependencies between models eliminated. A set of components may become customised and arranged to form a composition according to the aim of an experiment. Parameters set on component instances are evaluated and dependencies between components resolved.

3.2 Agents and James II

In JAMES II agents can either be modelled (i.e. JAMES II is used as a testbed for multi agent scenarios) [24], JAMES II can be used to create an environment in which "real" software agents can be plugged into and evaluated [8], or agent technology can be used to extend the simulation middleware of JAMES II.

JAMES II strictly separates between models and simulators. A model is a "pure" picture of the system under study and does not contain any simulation related information. Every agent model in JAMES II has always to be based on a supported modelling formalism. Agent specifics (e.g. communication protocols, migration issues) are either inherently supported by the modelling formalism used or they have to be explicitly modelled. I.e. in JAMES II agents and the environment they reside in are models. Agent models described outside of JAMES II have to be converted into an executable model of JAMES II before they can be simulated. The model reader schema allows the usage of arbitrary agent model repositories and it allows the usage of different agent modelling languages, if there is a translation into executable model classes of JAMES II. Later on we'll present an example of a multi agent simulation model, defined by using model components which are stored in XML (as described above).

JAMES II is well suited for the simulation of (large scale) multi agent models because the usage of different simulation algorithms for a model, e.g. different sequential and distributed ones, is supported. E.g., in JAMES II models can be executed by using a fine-grained parallel and distributed setup if the simulation of a model on a single machine is no longer possible (as long as there is an appropriate simulator plugin). If several simulations shall be executed in parallel (e.g. replications required because of stochastics) the model reader schema takes care of creating agent models directly on the hosts they shall be simulated on.

Thus the model reader schema enables flexibility for the simulation of multi agent systems in regards to model sources, computation schema used and up to a certain degree in regards to the modelling formalism used.

4. EXPERIMENTS IN JAMES II

Experimenting is a difficult, time consuming (sometimes event too time consuming [7]) and error prone task. Different types of experiments have to be conducted in a simulation study: runs for exploration are followed by validation experiments, for making sure that the model is valid. In addition, we might have to adjust some parameters by optimisation, or even re-design the entire model. Finally, we will execute the experiments to answer our initial questions we had in mind while creating the model. Consequently, a model must be independent from the type of experiment it is used in: we decided to create an explicit experiment description, which just links the model to be experimented with.

Reading/writing experiments.

JAMES II is not restricted to a concrete experiment definition language. Any experiment definition is fine as long as there is a suitable plugin for reading and converting the experiment definition. For the plugin to be accessible by JAMES II the interface *IExperimentReader* has to be implemented. A user interface may initiate the reading of an XML-based experiment definition from a database. The reader retrieves XML data and initialises the experiment instance according to the experiment definition. Changes to an experiment or a newly created experiment may be saved by JAMES II using a so called experiment writer, which implements the *IExperimentWriter* interface.

Job creation.

The support of different hardware infrastructures can reduce the overall time needed for an experiment if corresponding hardware is available. Consequently, an experiment definition in JAMES II is independent from the system an experiment will be executed on. An experiment definition usually comprises the parameters to be modified from simulation run to simulation run, how they shall be modified (e.g. by an optimisation method), the number of replications (e.g. for achieving statistical reliability), and so on. An experiment definition creates *simulation configurations* ("model and simulation parameter combinations") which are transferred to a *simulation runner*. Depending on the simulation runner used, the simulation configurations will be executed sequentially on a single host or by using a coarse- or fine-grained parallel simulation on any (supported) hardware infrastructure.

Simulation creation.

An executable model is created on the machine the simulation will be executed on by using the model reader schema. For creating a simulation from a job based on the XML model components as described above, a *ModelReader* retrieves the XML-data of the components from the database, unmarshals it, and configures the component(s) according to the provided model parameters (e.g. create 200 or 400 agent instances in the model). If a remote access to the model source (database) is possible, the model, or parts of it, might reside anywhere in the world. The executable model is created by using a ModelFactory which converts the model components into an executable instance of the target modelling formalism. Further details of the model creation process can be found in [26].

If the model has been created, an *instrumenter* will take care of attaching observers to the model. The selected data sink will be attached and the simulators will be created and instrumented. Afterwards the simulation is executed.

5. A MULTI AGENT EXAMPLE

Mobile ad-hoc networks (MANETs) are computer networks based on wireless communication. MANETs are characterised by dynamic network topologies. Nodes induce topology changes by appearing, disappearing, and moving in a spatial environment. To provide fast and reliable connections poses a severe challenge for MANETs, because MANETs lack central infrastructure and bandwidth as well as energy are strictly limited [3].

Network devices operating independently of the mains, typically have limited capabilities in terms of memory, battery and performance. Complex operations require cooperation among network nodes. To make cooperation possible, resources and capabilities of nodes, in the following referred to as services, have to be announced by providers and need to be locatable be a requester. From the perspective of human users, services should be accessible in a transparent

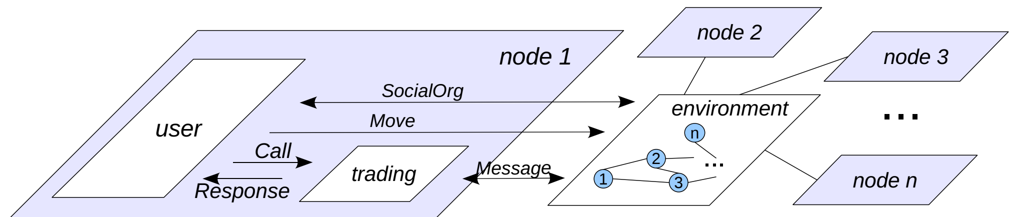

Figure 5: Conceptual model for evaluating service trading in mobile ad-hoc networks.

manner. Therefore, service descriptions, service matching algorithms, incentive schemes, and distributed reputation systems are developed in the project DIANE [4]. However, these mechanisms need to be thoroughly evaluated [15].

Conducting experiments to evaluate service trading in MANETs requires user models to represent network nodes as autonomous actors, which move in a spatial environment and announce and request services. Developers of service trading protocols are mainly interested in the cost-benefit ratio of protocols, if these are confronted with different kinds of user models. However, MANETs are typically simulated with special simulation systems, i.e. network simulators [17], which concentrate on the lower layers of the OSI protocol stack, e.g. routing protocols on the third layer. Within theses simulation systems, models for representing movement and service behaviour are simply calculated based on stochastic distributions [29]. As network traffic in MANETs is sensitive to local accumulation of nodes and temporal accumulation of network usage, more complex user models are needed, e.g. to allow a consistent modelling of motion and service behaviour and to incorporate social aspects. Thereby, users need not merely move individually but may form groups and move as clusters aiming at same destinations and partly synchronising their schedules.

Please note, that the purpose of this simulation scenario is not to test agents, but to use agent models for evaluating service trading protocols. Thereby, the agent models are part of the experimental setup.

Figure 5 shows a conceptual model for evaluating service trading in MANETs. The network consists of mobile nodes. Network connection between nodes depend on their positions. Each node comprises a user model and a service trading model such that the trading protocol mediates all network interactions and thereby provides transparent access to services available in the network. Models are supposed to exchange the following types of events:

Call A user initiates to announce, to revoke, or to search a certain service.

Response The trading protocol reacts, after having performed all necessary actions, to calls of the user with according responses.

Message Trading protocols communicate over the network with messages of arbitrary content.

Move Movement information in a two-dimensional spatial environment.

SocialOrg Social organisation requests and responses.

User models initiate communication by sending calls to the trading protocol. A call may indicate the publication of or search for services. Calls are passed to the protocol model, which answers user calls with *Response* events.

The concrete type of the user model and the trading protocol should be a variation point of the simulation model. Different kinds of user and protocol models should be exchangeable independently of each other within one node. For simulating service trading in MANETS with James II, user and protocol models have been realized as model components.

In the following we'll show how the experiment definition together with the model reader schema facilitates the flexible experimentation with this component-based model.

5.1 Defining composition structures

Figure 6 shows the composition structure of the Manet component. The *Manet* component can take parameters to initialise the number of nodes to be simulated, the type of the user model to be used, and the type of the protocol model to be used inside each node.

The spatial environment represents a certain geographical area containing streets and buildings and it keeps track of all user positions. The network component models the transport layer of the network. In real applications the whole OSI stack is part of each node. Since we are interested in higher level protocols the four lower OSI layers are pooled in a centralised network component, which delivers messages to nodes. The connectivity of each node is calculated according to its position with respect to other nodes.

Each node contains a *Protocol* component and a *User* component. From the point of view of the node component, the user and protocol components are black boxes whose couplings are defined by interfaces. The parameters *user* and *protocol*, which may be set on a node component, determine the type of sub components to be used. Each of both can be easily replaced by another one which provides the same interface. Three different versions of the user component have been implemented up to now. They can be composed into a node alternatively. We will now take a look at different implementations of user components and protocol components. All of these are realised themselves as composite components.

The simple user contains an *activity* sub component that manages login and logout behaviour. If logged in, the service component becomes notified to start publishing services and searching for services randomly according to a uniform distribution. Furthermore, the *activity* sub component selects destination points randomly and calculates routes to them. Routes are propagated to the *Motion* sub component, which executes them with a certain walking speed. After reaching a destination, a new route is requested from the *activity* component.

The second user model realises an activity-based user be-

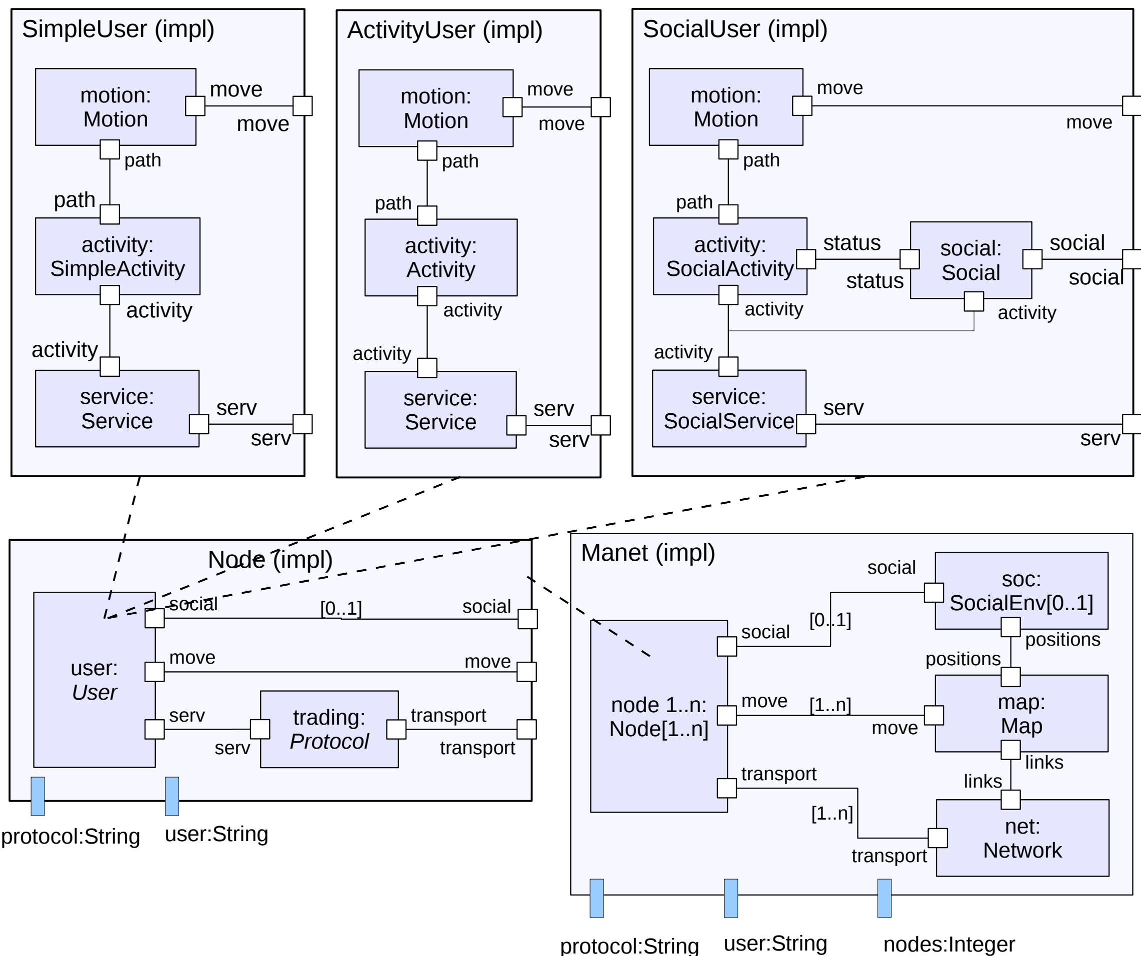

Figure 6: Structure of the manet model with three alternative user models.

haviour. Network usage and moving is not modelled independently of each other. Both depend on the activity a user is currently executing. The *Activity* model generates a schedule at the start of the day. The schedule contains fixed activities, e.g. attending a lecture as well as flexible activities, such as learning, with different priorities and durations. The current activity influence the *Motion* and *Service* sub component. The service component generates service offers and service requests according to the received activity. Activities are not independent of the spatial context, but each location is only suited for a certain set of activities. The current activity constrains the choice of the next destination and thereby the motion model. Thus, motion and service behaviour are both based on activities.

The third user model extends the activity-based model with social awareness. The sub component *Social* of each user announces planned activities to the social environment model. If users have planned similar activities and are spatially close the social environment forms a group and selects a group leader. The group leader chooses activities, which all group members may adopt. Because an activity does not uniquely define the location of performance, the group leader chooses a location out of a set of suited ones and communicates this choice to all group members. The group members are free to choose a path to the location. Thus, social users are synchronised with respect to the next joint activity and the location where this activity will be performed.

5.2 Defining model behaviour

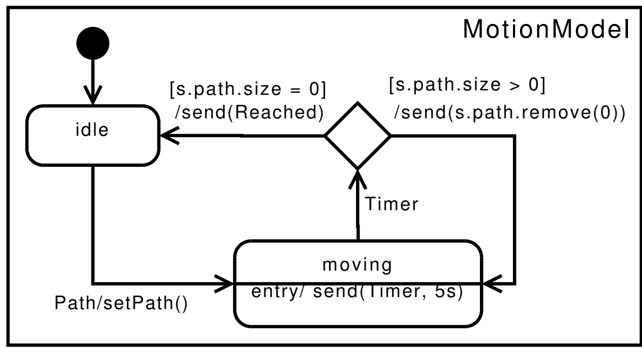

Figure 7: Motion model as a statechart.

Composition structures define the composition hierarchy of a model. The leaves of the model tree have to be equipped with behaviour, which in JAMES II may be done using different modelling formalisms. Statecharts are used in the

following to exemplify the definition of model behaviour.

Figure 7 visualizes the definition of the model behavior for the component *Motion* as a statechart. The model *Motion* starts in phase *idle* and waits for input. If path information is received, the model goes to phase *moving*. Each time phase *moving* is entered, a movement event is produced. After a certain time, triggered by a special send operation in phase *moving*, it is checked, whether the end of the path was reached. If this is not the case, the model enters phase *moving* again. If the end of a path is reached, the model produces an event of type *Reached* and returns to phase *idle*.

With SCXML [31] a proposal exists for representing statecharts in a style accessible not only to computers – like XMI, the exchange format for UML state machines [20] – but also to humans. Figure 8 lists the definition of the *Motion* model in SCXML.

```
<scxml
    xmlns="http://www.w3.org/2005/07/scxml"
    version="1.0" initialstate="idle">
 <datamodel>
  <data name="path"
      src="../PathProc:Path"/>
  <data name="posChange"
   src="../geometry:Position"/>
  <data name="move" src="../motion:Move"
      expr="create"/>
 </datamodel>
 <state id="idle">
  <transition event="../PathProc:Path"
      target="moving">
   <assign location="path"
       expr="_eventdata"/>
  </transition>
 </state>
 <state id="moving">
  <onentry> <send event="Timer"
      delay="5"/> </onentry>
  <transition event="Timer"
    cond="s.path.size() > 0"
       target="moving">
   <assign location="posChange"
       expr="s.path.remove(0)"/>
   <assign location="move"
       expr="setChange(s.posChange)"/>
   <send event="../motion:Move"
       namelist="move"/>
  </transition>
  <transition event="Timer"
   cond="s.path.size()==0" target="idle">
   <send event="../PathProc:Reached"/>
  </transition>
 </state>
</scxml>
```

Figure 8: Definition of the *Motion* model in SCXML.

5.3 Deriving Simulation Models

In the following 400 nodes are simulated with the trading protocol *Lanes* and different user components inside each network node. To simulate these different compositions, an experiment has to be defined. Figure 9 lists an according experiment definition in XML. The experiment induces three different parameter combinations ($|nodes| * |protocol| * |user|$). Each combination results in a separate simulation configuration.

```
<experiment
  xmlns="http://...de/cosa/experiment"
  xmlns:xsd="http://..org/2001/XMLSchema"
  xmlns:exp="unihro/diane/experiment"
  xmlns:net="unihro/diane/com/manet">
  <id>exp:experiment</id>
  <model>net:interface</model>
  <mparams>
    <param name="nodes">
      <value>400</value>
    </param>
    <param name="protocol">
      <value>Lanes</value>
    </param>
    <param name="user">
      <value>SimpleUser</value>
      <value>ActivityUser</value>
      <value>SocialUser</value>
    </param>
  </mparams>
  <targetFormalism>
    dynpdevs
   </targetFormalism>
  <platform>.../cosa/jamesii</platform>
  <observercfg>
    diane.experiments.v05.ObsVis
  </observercfg>
  <sparams>
    <startTime>0.0</startTime>
    <endTime>32400</endTime>
  </sparams>
</experiment>
```

Figure 9: Definition of an experiment in XML.

The experiment reader configures simulation job description, which are passed to an instance of the appropriate ModelReader, which reads and creates the model according to these.

Figure 10 shows the structure of a resulting simulation model that was derived using “400” nodes, the “Lanes” protocol and the “social user” component. The outcome is an executable PDEVS model [32], which was generated from the composition structures and model behaviours [27]. The parameter values are translated into a corresponding number of network nodes and are reflected in the internal structure of each node. After having created the model, observers for collecting experiment data are attached to the model.

Figure 11 shows three simulation runs, each with 400 nodes. Users appear uniformly distributed between 0 and 60 minutes and immediately log into the network. Subfigure 11 a) depicts the run which uses the “simple user” model for generating calls. For measuring the load of the network three different types of messages are distinguished. All messages that result from building up the lane structure are summarized by the *login* trajectory. Messages for keeping the lane structure valid are subsumed under *intra lanes* messages. Messages that are used to answer service calls of users are summarized within the *inter lane* message trajectory. Using simple user models results in a quite uniformly scattered number of service messages during the whole online period.

The trajectories in subfigure b) are produced using the “activity-based user” model. The effort for service related

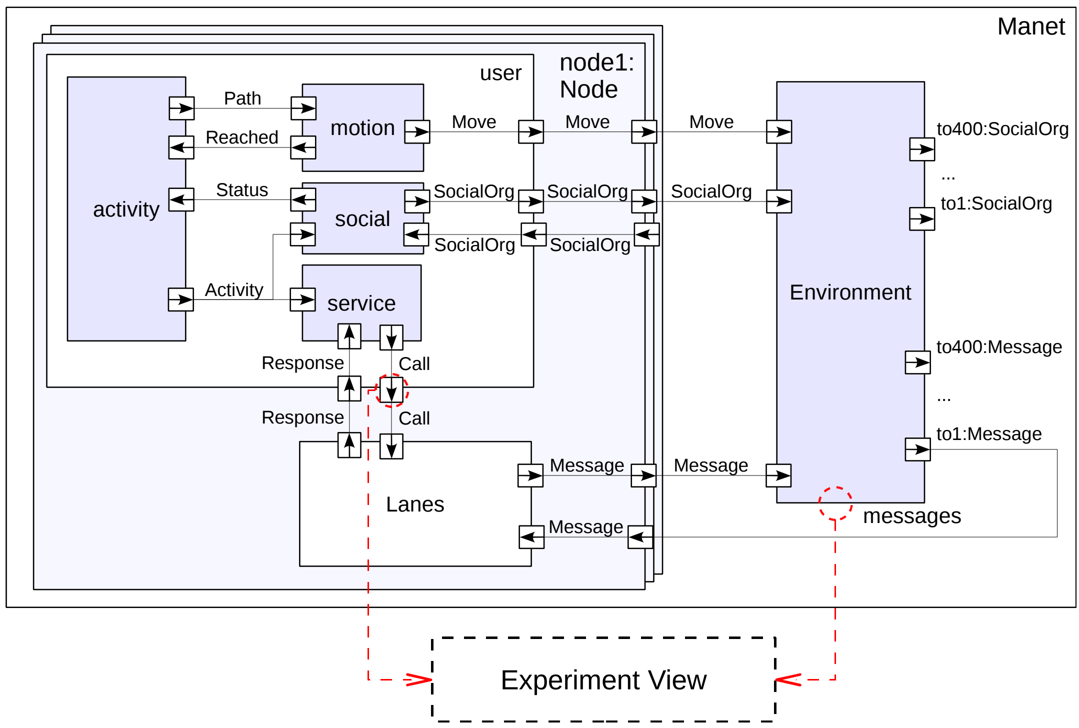

Figure 10: Structure of the derived simulation model.

messages varies not so much from minute to minute, but different periods with higher and lower numbers of service messages can be distinguished. The periods reflect globally scheduled user activities, e.g. attending a lecture. As indicated by subfigure c) "social user" models show less distinct global behaviour patterns. For further discussions of this experiment please refer to [24].

6. CONCLUSION

James II realizes a plugin-based architecture that allows a flexible re-use of concepts in the area of modelling and simulation. The core of James II provides standard functionality and the means to integrate additional functionality. If an experiment is conducted all those parts come together. Here we focused on the possibility to support different model descriptions in James II, taking a component-based declarative modelling of a multi-agent system as an example. Even though the model is based on a declarative schema the execution is still quite efficient. This is achieved by mapping the declarative model (by a ModelReader) on executable model classes, which can be executed efficiently.

We have shown how "external" component-based model descriptions (here "COMO") can be embedded into James II and that a component-based modelling of multi agent system is manageable and useful. This mechanism can be easily extended to integrate further different (component-based) model descriptions. In addition we voted for an explicit experiment definition and shortly sketched how an experiment definition can be converted into a James II experiment. The strict separation of models (agents) and simulators eases the support of different hardware and thus allows the efficient execution of (many) small and large scale simulations, and thus helps to avoid errors as discovered in [7].

7. REFERENCES

[1] V. Balakrishnan, P. Frey, N. B. Abu-Ghazaleh, and P. A. Wilsey. A framework for performance analysis of parallel discrete event simulators. In *WSC '97: Proceedings of the 29th conference on Winter simulation*, pages 429–436, New York, NY, USA, 1997. ACM Press.

[2] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster. The möbius modeling tool. In *9th international Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 241–250. IEEE, 2001.

[3] S. Corson and J. Macker. Mobile ad hoc networking (MANET): Routing protocol performance issues and evaluation considerations. `http://www.ietf.org/rfc/rfc2501.txt`, Jan. 1999. Network Working Group Memo (Request for Comments: 2501).

[4] Diane. Diane-projekt: Services in ad hoc networks (Dienste in Ad-Hoc-Netzen). `http://www.ipd.uni-karlsruhe.de/DIANE` (zugegriffen am 19. September 2007), 2007.

[5] K. Fall and K. Varadhan. *The ns Manual (formerly ns Notes and Documentation)*. The VINT Project, a collaboratoin between researchers at UC Berkeley, LBL, USC/ISI, and Xerox PARC., 2008. `http://www.isi.edu/nsnam/ns/ns-documentation.html`.

[6] J.-B. Filippi and P. Bisgambiglia. JDEVS: an implementation of a DEVS based formal framework for environmental modelling. *Environmental Modelling and Software*, 19(3):261–274, March 2004. Elsevier Science.

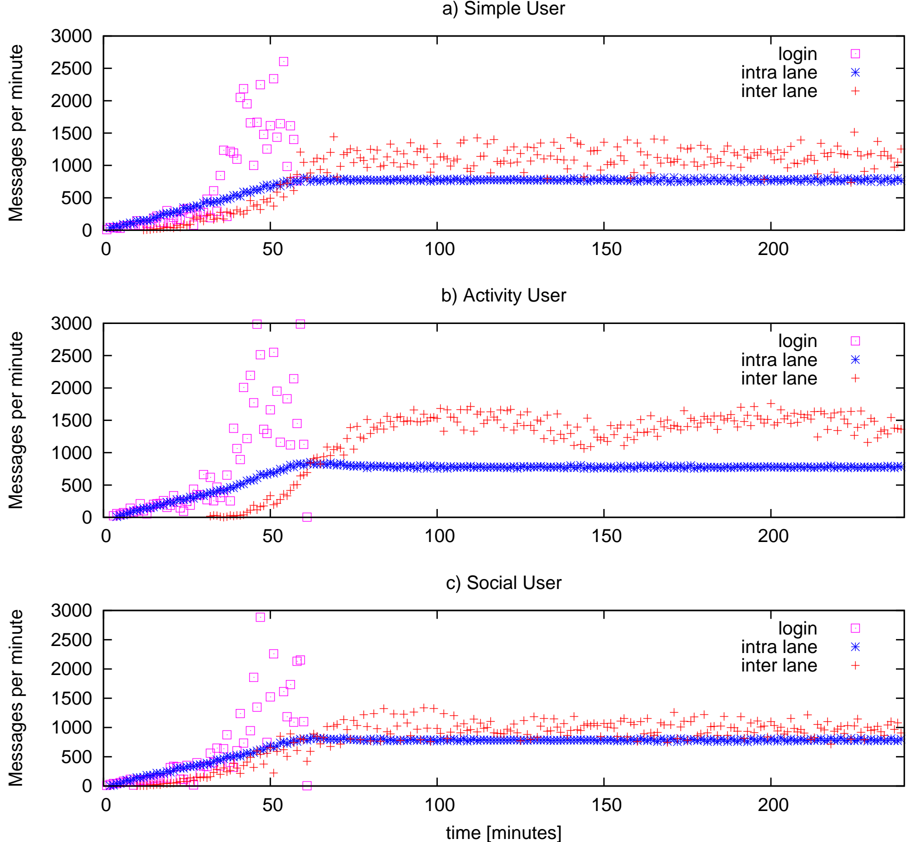

Figure 11: Number of network message with different user models.

[7] J. M. Galan and L. R. Izquierdo. Appearances can be deceiving: Lessons learned re-implementing axelrod's 'evolutionary approach to norms'. *Journal of Artificial Societies and Social Simulation*, 8(3), 2005.

[8] M. Gierke. Coupling Autominder and James. Diplomarbeit, Universität Rostock, Jan. 2005.

[9] G. N. Gilbert. Environments and languages to support social simulation. In *Social Science Microsimulation*, pages 457–458, 1995.

[10] S. Hanks, M. E. Pollack, and P. R. Cohen. Benchmarks, testbeds, controlled experimentation, and the design of agent architectures. *AI Magazine*, 14(4):17–42, 1993.

[11] J. Himmelspach. *Konzeption, Realisierung und Verwendung eines allgemeinen Modellierungs-, Simulations und Experimentiersystems - Entwicklung und Evaluation effizienter Simulationsalgorithmen*. Reihe Informatik. Sierke Verlag, Göttingen, Dec. 2007.

[12] J. Himmelspach and A. M. Uhrmacher. Sequential processing of PDEVS models. In A. G. Bruzzone, A. Guasch, M. A. Piera, and J. Rozenblit, editors, *Proceedings of the 3rd EMSS*, pages 239–244, Barcelona, Spain, Oct 2006.

[13] J. Himmelspach and A. M. Uhrmacher. Plug'n simulate. In *Proceedings of the Spring Simulation Multiconference*. IEEE Computer Society, March 2007.

[14] P. H. M. Jacobs, N. A. Lang, and A. Verbraeck. Web-based simulation 1: D-sol; a distributed java based discrete event simulation architecture. In *WSC '02: Proceedings of the 34th conference on Winter simulation*, pages 793–800. Winter Simulation Conference, 2002.

[15] M. Klein, M. Hoffman, D. Matheis, and M. Müssig. Comparison of overlay mechanisms for service trading in ad hoc networks. Technical Report TR 2004-2, University of Karlsruhe, Oct. 2004. ISSN 1432-7864.

[16] F. Klügl and F. Puppe. The multi-agent simulation environment SeSAm. In H. K. Büning, editor, *Proceedings des Workshops Simulation in Knowledge-based Systems*, volume tr-ri-98-194 of *Reihe Informatik*, Paderborn, April 1998. Universität Paderborn.

[17] S. Kurkowski, T. Camp, and M. Colagrosso. MANET simulation studies: The incredibles. *ACM's Mobile Computing and Communications Review*, 9(4):50–61, 2005.

[18] A. Martens and J. Himmelspach. Combining intelligent tutoring and simulation systems. In P. Fishwick and B. Lok, editors, *Proceedings of the International Conference on Human-Computer Interface Advances for Modeling and Simulation (SIMCHI'05)*, pages 65–70, New Orleans, USA, Jan. 2005. SCS, The Society for Modeling and Simulation International.

[19] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The SWARM simulation system: a toolkit for building multi-agent simulations. Technical report, Santa Fe Institute, June 1996.

[20] OMG. Unified Modeling Language: Superstructure version 2.1 (document ptc/2006-04-02). `http://www.omg.org/cgi-bin/doc?ptc/2006-04-02`, Apr. 2006.

[21] K. S. Perumalla. μsik: A micro-kernel for parallel/distributed simulation systems. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 59–68, Washington, DC, USA, 2005. IEEE Computer Society.

[22] M. E. Pollack and M. Ringuette. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. In *AAAI-90*, pages 183–189, Boston, MA, 1990.

[23] M. Röhl. Platform independent specification of simulation model components. In SCS, editor, *ECMS 2006*, pages 220–225, 2006.

[24] M. Röhl, B. König-Ries, and A. M. Uhrmacher. An experimental frame for evaluating service trading in mobile ad-hoc networks. In *Mobilität und Mobile Informationssysteme (MMS 2007)*, volume 104 of *Lect. Notes Inform.*, pages 37–48, 2007.

[25] M. Röhl and A. M. Uhrmacher. Flexible integration of XML into modeling and simulation systems. In *Proceedings of the 2005 Winter Simulation Conference*, pages 1813–1820, 2005.

[26] M. Röhl and A. M. Uhrmacher. Composing simulations from xml-specified model components. In *Proceedings of the Winter Simulation Conference 06*, pages 1083–1090. ACM, 2006.

[27] M. Röhl and A. M. Uhrmacher. Composing simulations from XML-specified model components. In *Proceedings of the Winter Simulation Conference*, pages 1083–1090. ACM, 2006.

[28] Sun. Java architecture for xml binding (JAXB). `http://java.sun.com/xml/jaxb`, 2005.

[29] D. S. Tan, S. Zhou, J.-M. Ho, J. S. Mehta, and H. Tanabe. Design and evaluation of an individually simulated mobility model in wireless ad hoc networks. In *Communication Networks and Distributed Systems Modeling and Simulation Conference 2002*, San Antonio, TX, 2002.

[30] A. Verbraeck. Component-based distributed simulations. the way forward? In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS'04)*, pages 141–148, 2004.

[31] W3C. State chart XML (SCXML): State machine notation for control abstraction. `http://www.w3.org/TR/2007/WD-scxml-20070221`, 2007. W3C Working Draft 21 February 2007.

[32] B. Zeigler, H. Praehofer, and T. Kim. *Theory of Modeling and Simulation*. Academic Press, London, 2000.