Expert Systems with Applications 36 (2009) 7630–7643

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

An integrated infrastructure for monitoring and evaluating agent-based systems ☆

Christos Dimou *, Andreas L. Symeonidis, Pericles A. Mitkas

Department of Electrical and Computer Engineering, Aristotle University of Thessaloniki, 54124 Thesssaloniki, Greece

ARTICLE INFO

Keywords:
Software agents
Performance evaluation
Automated software engineering
Fuzzy measurement aggregation

ABSTRACT

Driven by the urging need to thoroughly identify and accentuate the merits of agent technology, we present in this paper, MEANDER, an integrated framework for evaluating the performance of agent-based systems. The proposed framework is based on the Agent Performance Evaluation (APE) methodology, which provides guidelines and representation tools for performance metrics, measurement collection and aggregation of measurements. MEANDER comprises a series of integrated software components that implement and automate various parts of the methodology and assist evaluators in their tasks. The main objective of MEANDER is to integrate performance evaluation processes into the entire development lifecycle, while clearly separating any evaluation-specific code from the application code at hand. In this paper, we describe in detail the architecture and functionality of the MEANDER components and test its applicability to an existing multi-agent system.

1. Introduction

The intriguing properties and characteristics that intelligent agents and agent communities exhibit have given them a prominent place among the most promising programming techniques for the future. Indeed, agents provide an excellent modelling abstraction for autonomous or otherwise intelligent entities that exhibit human-like characteristics, including reasoning, proactivity, communication and adaptive behavior. Moreover, groups of cooperating or competing agents, conglomerated into MAS, provide efficient solutions for simulating, implementing and studying large-scale, complex systems.

Nevertheless, agent technology has not progressed as expected. When moving from the 'wild euphoria' to the 'down-to-earth' phase, researchers realized that the agent paradigm cannot/should not be applied to any given problem. The lack of consensus on the definition of an agent and its related entities has led to disputes within the research community (mainly between the Software Engineering and Artificial Intelligence disciplines). In turn, this has lead to the delayed release of appropriate development tools, creating a reluctance in the software engineering community to adopt and support agents with the appropriate theoretical and practical evaluating infrastructure. Thus the initial agent-phrenia of the last decade has now radically diminished (Knowles, 2002; Ruttens, 2002).

Further elaborating on the issue, it is rather evident that only few efforts attempt to address the engineering issues related to agent technology as a whole. Most publications explore the future potentials of agents or describe implementations, leaving many fundamental engineering issues unaddressed. Little effort has been spent towards both evaluation methodologies and software tools for integrating and automating the evaluation process. Even in the few foundational works that describe interesting methodologies for designing and developing agent-based systems (e.g. Bresciani, Perini, Giorgini, Giunchiglia, & Mylopoulos, 2004; DeLoach, 1999; Jonker, Klusch, & Treur, 2000; Wooldridge, Jennings, & Kinny, 2000), no reference is made on one of the most important aspects of agent software engineering, namely *formal performance evaluation*.

Evaluation is an integral part of any scientific or engineering discipline. Evaluation methodologies allow researchers to identify defects and advantages of a system, as well as to assess the quality of their findings. It comprises a necessary step towards better understanding of the nature of a newly proposed technology or of any implemented system. Focusing on agent technology, state-of-the-art evaluation is conducted in one of two manners: developers either use one of the existing methodologies for "traditional" software engineering (e.g. Huget, 2004) or, in the majority of cases, devise their own ad-hoc methodologies. The former most of the times prove to be inadequate for tackling emergent or unpredictable behavior performance; the latter are often tailored to specific applications and may, therefore, produce biased results. Consequently, existing monitoring and evaluating tools are of limited

☆ This paper is part of the 03ED735 research project, implemented within the framework of the "Reinforcement Programme of Human Research Manpower" (PENED) and co-financed by National and Community Funds (25% from the Greek Ministry of Development-General Secretariat of Research and Technology and 75% from E.U.-European Social Funding).

* Corresponding author.
E-mail address: cdimou@issel.ee.auth.gr (C. Dimou).
URL: http://issel.ee.auth.gr (C. Dimou).

0957-4174/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2008.09.061

scope and are almost always detached from a generic, fully-structured methodology.

We argue that the development and proliferation of sophisticated evaluation tools would identify and cement the quantitative advantages of the AOSE paradigm, while at the same time expose the nature of the disadvantages of agent-based systems. In either case, evaluation of agent systems performance would give the necessary boost to agent-related research, based on solid engineering foundations, correctly placing agents in the advanced programming setting.

Based on these arguments, we have developed MEANDER (Monitoring and Evaluating Agent Environment), an integrated framework that provides a set of tools for performing the evaluation tasks in a semi-automated, interactive manner. Adhering to the rules and constraints of a complete, generalized evaluation methodology for MAS, MEANDER provides a standardized software-assisted evaluation environment that will enable: (a) quick selection of evaluation metrics and related parameters, (b) semi-automatic collection of measurements, (c) easy organization of experiments and (d) aggregation and graphical representation of the results of each evaluation session. MEANDER provides an integrated environment of various interacting components, that include automatic generation of evaluation-specific code, distributed measurements collection, and Graphical User Interfaces (GUIs). It should be denoted that focus should be given on the demonstration of the efficiency of the framework in automating the parts of an evaluation process in agent-based/intelligent systems and not the implementation itself. Thus, one may utilize different existing technologies to build a different implementation following the same evaluation methodology.

The remainder of this paper is structured as follows: Section 2 presents the body of literature that is related to evaluation and monitoring issues of agent-based systems; Section 3 briefly outlines our generalized methodology for agent performance evaluation; in Section 4, MEANDER is described in detail and analyzed in its structuring blocks; Section 5 illustrates a demonstrator for the functionality of the proposed framework, while the paper concludes with Section 6 that summarizes the work conducted and indicates future directions.

2. Related work

Automated performance evaluation is part of the *Automated Software Engineering* (ASE) field. ASE is an umbrella term that encompasses a holistic view for automation on any part of the software development lifecycle, from conceptualization to software product maintenance. Within this field, a plethora of automated tools and methods has been proposed. Although most of the below referenced tools often cross boundaries, they can be roughly categorized in the following aspects of software development:

(i) Theoretical: specifications and design (Cai, Huynh, & Xie, 2007; Compare, D'Onofrio, Di Marco, & Inverardi, 2004; de la Riva & Tuya, 2006), requirements elicitation (Lurya, 2005; Rafla, Robillard, & Desmarais, 2007), model checking (Flanagan, 2004; Godefroid & Jagadeesan, 2002),
(ii) implementation/testing: development (Smith, 2007), debugging (Andrei, Chin, Cheng, & Lupu, 2006; Auguston, Jeffery, & Underwood, 2002), fault-tolerance (Delgado, Gates, & Roach, 2004; Stoller & Schneider, 1996), product quality (Berki, Georgiadou, & Holcombe, 2004; Chu & Dobson, 1996), maintenance (Sellink & Verhoef, 1999; Veerman, 2007).

In this paper, we focus on the automated performance evaluation tools. Since the early days of computing, software performance evaluation of systems has focused on the efficient use of computing resources, such as memory, CPU power and I/O devices (Hellerman & Conroy, 1975; Wang, Bennett, Lance, & Woehr, 2006). With the advent of distributed computing, network resources utilization was added to the above list (Coelli & Lawrence, 2007). Nowadays, efficient computing resources utilization is still a major performance factor for any implementation, including newly proposed paradigms, such as the Grid (Huedo, Montero, & Llorente, 2005), Autonomic Computing Systems (Huebscher & McCann, 2004) and P2P networks (Benevenuto, Ismael, & Almeida, 2004; Gummadi et al., 2003). Automated tools in all the above cases deal with monitoring a system at runtime and collecting measurements on selected resource utilization indicators.

However, in agent-based computing and, generally, in intelligent systems (IS) that deal with complex, unpredictable and dynamic behavior of software components, traditional performance evaluation methods and tools have been rendered inadequate. Indeed, resource utilization is only one of several performance issues arisen in domains that agent-based systems address. In practice, researchers in the field of agent technology often develop their own *ad-hoc* evaluation methods and tools for every newly proposed agent-based system they develop, making practically impossible for third parties to repeat and validate their findings. Moreover, these methods and tools are often inseparable from the source code of the original application, blurring the borders of development and testing.

In the field of IS evaluation, two main approaches can be identified: the bottom–up and the top–down. The former, as elaborately presented by Gudwin (2000) and Zadeh (2002), advocates the definition of formal constructs and languages that will enable the definition of the appropriate terms and scope of IS. Evaluation methods and tools will thereafter be gradually built upon these formal foundations. The latter approach observes that existing or newly implemented systems urge for evaluation methodologies and it is therefore, preferable to instantly evaluate them at any cost. According to this approach, experiences from different ad-hoc evaluation attempts will be generalized into a concise domain-independent methodology, which in turn will be established at the time that IS reach a sufficient maturity level.

3. The agent performance evaluation methodology

3.1. Objectives

The proposed MEANDER framework is intended to provide a series of tools that foster, implement and automate parts of the Agent Performance Evaluation, a generic evaluation methodology that has been previously sketched in Dimou, Symeonidis, and Mitkas (2007). The main objective of APE is to provide readily available guidelines and representation tools for evaluators in the field of agent technology. APE is mainly targeted to researchers and developers in this field; however, the methodology could also be generalized in other types of intelligent systems (i.e. intelligent web services, SOAs, etc.). APE comprises a set of predefined steps for evaluators to walk-through in order to realize a complete evaluation session, leaving implementation and measurement-specific issues to the choice of the evaluator. The methodology is not tied to specific application domains; it rather comprises more generic steps, guidelines, and tools that can be adjusted to any specific application domain. APE follows the *top–down* approach, mentioned in the previous section, and it is therefore, applicable to existing applications or applications that meet current agent-oriented software engineering concepts.

The APE methodology provides the means for addressing aspects of system performance at different levels of granularity,

depending on the application scope and the evaluation needs. In a fine-grained level, it is usually desirable to isolate single agents or even, at a more detailed level, agent-specific tasks and observe their performance and impact of their actions on their environment. In a coarse-grained level, one may focus on the overall behavior of an agent society and the outcome of transparent collaborative problem solving, competition or negotiation. In the former case, we consider each participating agent as a black box, whereas in the latter case we regard the entire MAS as a black box. In both cases, as well as in the many in-between cases, the methodology supports the isolation and measurement of certain characteristics of the observable behavior for each "black box".

Following this approach, one may assess the impact of the actual performance of agents and MAS, bypassing the methods of implementation, algorithms and other intrinsic characteristics. Implementation independence makes the proposed methodology a powerful tool for measuring both directly the actual efficacy of a system as a whole, and indirectly the performance of the intrinsic methods employed. In other words, the algorithmic decisions are indirectly handled and revised in an iterative manner, if and only if the results of the observable behavior evaluation are not within acceptable limits. Moreover, two systems implemented in a different way can still be compared against each other, solely in terms of performance, having the underlying mechanisms implicitly compared at the same time.

The APE methodology identifies and analyzes three aspects of a complete evaluation session: (a) metrics, (b) measurement[1] and (c) aggregation. These three aspects are briefly presented in the following sections.

3.2. Metrics

3.2.1. Metrics definition

Metrics are standards that define measurable attributes of entities, their units and their scope. In practice, they represent essential aspects of a system, aiming to identify and reveal some interesting system properties. The process of defining a set of appropriate metrics establishes the specific goals of a performance evaluation session. In the subsequent measurement phase, measured values are assigned to metrics and therefore, conclusions can be reached for the degree to which a specific system attribute has met its required goals. In the case of significant deviation from a desired range of acceptable values, the specific system aspect and related components should be revised, modified, re-designed or re-implemented. With respect to a complete evaluation methodology, a metric is the answer to the question: *'What should I evaluate?'*.

3.2.2. Metrics in APE

The APE methodology introduces the Metrics Representation Graph (MRG), a metrics representation tool for metrics organization and categorization in the form of an acyclic directed graph (Fig. 1). The graph is organized in hierarchical layers or views of granularity from the general to the specific. The bottom-most layer of the graph comprises a set of *Simple metrics*, i.e. metrics that are directly measurable and are therefore, assigned specific measurement values. Moving upwards in the graph, Simple Metrics can be integrated into *Composite Metrics*, i.e. metrics that are not directly measurable, but represent higher-level concepts of performance, such as *scalability*, *modularity*, etc. Composite Metrics may be composed of any number of Composite and Simple metrics, leading to the root node of the overall *System Performance*.

[1] Although the terms *Metric* and *Measurement* (or *Measure*) are often used interchangeably, in this paper, we follow the popular convention that the former term refers to the attribute of a system's performance, whereas the latter term refers to the observed or collected at runtime value for a specific attribute.

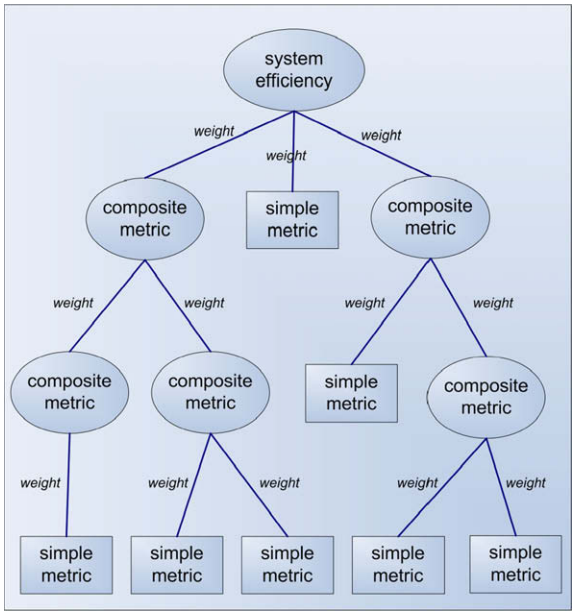


Fig. 1. The basic structure of the metrics representation graph.

Following the choice of metrics in the MRG, a list of parameters for each metric has to be defined, either manually by the domain expert or semi-automatically (see Section 6 on automation concepts). The parameters include a number of attributes such as the preferred scale of measurement, frequency of measurement, time intervals etc. Moreover, for each metric, a weight value is defined, signifying the contribution of the metric to its parent metric.

3.3. Measurement

3.3.1. Measurement definition

Measurement is defined as 'the process of ascertaining the attributes, dimensions, extend, quantity, degree of capacity of some object of observation and representing these in the qualitative or quantitative terms of a data language' (Krippendorff, 1986). Within the scope of an evaluation methodology, a measurement method refers to the parametrization and execution of actual experiments and the subsequent assignment of observed values to the selected metrics. A measurement method is the answer to the question *'How should I perform the experimental evaluation?'*.

3.3.2. Measurement guidelines in APE

Organizing and executing experimental measurements can be particularly complex, but the process provides a considerable degree of freedom for parameter choices to the evaluator. Indeed, the properties and nature of two experiments may be significantly different, even in the case of evaluating a single implementation with respect to different aspects. Consequently, it is impractical to provide detailed guidelines within the context of a generic, domain-independent evaluation methodology, due to the significant diversity of applications in various domains.

The APE methodology employs a taxonomy of experimental methods, introduced by Kitchenham (1996), in order to provide the evaluator with generic guidelines for his/her experiments. These experimental options are summarized in Table 1.

While APE provides only a simple, generic list of methods for the evaluator to choose from, the actual contribution to measurement aspects of an evaluation session is achieved by organizing and automating parts or all of the measurement tasks. The MEANDER framework aims exactly at organizing various measurement aspects, without restricting any of the domain- or implementation-specific

Table 1
Categories of experimental measurement

(a) Quantitative methods
Experiments, case studies, surveys, screenings
(b) Qualitative methods
Experiments, case studies,survey, effects analyses
(c) Benchmarking

choices that the evaluator makes in order to customize their experiments.

3.4. Aggregation

3.4.1. Definition

Aggregation, often referred to as *multicriteria composition*, is the process of summarizing multiple measurements into a single measurement in such a manner that the output measurement will be characteristic of the system performance.

Aggregation groups and combines the observed measurements, using weights, functions and/or operators, in order to result into atomic characterizations, either for complex aspects of the evaluated system or for the system as a whole. The process of aggregation is an important post-processing task, combining heterogeneous or even contradicting measurements. It is very common for a system to perform well in some aspects while performing poorly in other aspects. Aggregation designates the contribution of each aspect to the overall system performance and signifies whether actions should be taken to improve or modify defecting components. Aggregation answers to the question: *'How to combine the results of the evaluation process?'*.

3.4.2. Aggregation in APE

Within the context of the APE methodology, aggregation takes place with the assistance of the MRG. Following the completion of the experiments and the collection of measurement values, the MRG is traversed in a bottom–up fashion. At each level of the graph, aggregation occurs by combining the measured values of the corresponding metrics based on the weights associated with each edge of the graph. While the MRG figure gives a general guideline for the aggregation process, it does not specify the details of the aggregation methods employed. Generally, the evaluator is free to implement and utilize any possible aggregation method. In Section 4.9, we present and describe in detail the concepts of *Fuzzy Aggregation*, a specific aggregation method, based on fuzzy sets and fuzzy operators.

4. The MEANDER evaluation framework

The APE methodology is fully supported by the MEANDER evaluation framework that undertakes all the tasks related to the automation of the evaluation process. The general architecture of the framework, as well as the analysis of the core components are described next.

4.1. General architecture

The goal of MEANDER is to provide a complete set of interactive tools that automate the steps of the APE methodology. These tools are expected to speed up the design and configuration of the evaluation process, while also reducing human-generated errors. Through the automation that MEANDER provides, developers are able to configure experiments and organise them into batch jobs that may be executed with no human intervention. Measurements for specified metrics are automatically collected, aggregated and presented to the developer by graphical means.

The overall architecture of MEANDER is depicted in Fig. 2, where the core components may be identified:

(i) **MRG GUI**: Interactively creates and/or modifies the application MRG.
(ii) **Evaluator API**: Implements basic logging, communication and measurement methods, in a general, domain- and implementation-independent way.
(iii) **Jar Generator**: Generates a domain-specific Java archive (.jar file) based on the previously defined MRG and the Evaluator API.
(iv) **JAR Invocation**: Imports the above generated Java archive into the MAS application code and produces an evaluation-enabled MAS.
(v) **Runtime**: Organizes/executes batch experiments on the MAS and produces measurement data.
(vi) **Logger**: Collects and organizes distributed measurement data.
(vii) **Aggregation**: Assigns measurements to Simple Metrics and composes them into Composite Metrics measurements, according to the aggregation information of the MRG.
(viii) **Results Presenter GUI**: Interactively presents the evaluation results using graphical means.

When specifying a new evaluation process through MEANDER, one follows the steps shown in the MEANDER UML Activity Diagram (Fig. 3).

4.2. Graphical user interfaces

The end components of MEANDER are two Graphical User Interfaces that enable the evaluator to define and graphically manipulate (a) the input MRG, and (b) the output measurements collected throughout the experiments. Both GUIs share some common characteristics, in order to provide a unique and uniform *look* & *feel* to the interaction between the evaluator and the framework. Both GUIs have been developed as stand-alone applications, as well as Eclipse plugins, so that they can be integrated in the development environment. This way, evaluation is included in the development process. Moreover, both GUIs treat information in a unified, structured manner, employing W3C standards and technologies for ontology representation and information exchange in distributed environments.

4.2.1. MRG Editor GUI

The MRG Editor GUI provides a fully interactive means for the creation of a new Metrics Graph or the modification of an existing one. A screenshot of the GUI is depicted in Fig. 4.

One key feature of the GUI is its ability to import and export metrics graphs in standardized format. The MRG manipulation tool loads and saves the MRGs adhering to the ontology structure for metrics, presented in detail in Section 4.8.

Once loaded or created, a graph can be fully edited. Besides the basic functionality of creating and deleting nodes and edges, editing options allow customization of the measurement specific attributes (edge weights, preferred scales of measurement, frequency of measurement and time intervals).

Finally, fuzzy aggregation is also supported by the GUI by means of interactively selecting, defining and editing fuzzy variables and their corresponding fuzzy membership functions. These variables can be linked with specific metrics of the MRG.

4.2.2. Results presenter GUI

The other GUI of MEANDER is intended for the graphical representation of the collected measurements. Following the Aggregation component that is described in Section 4.9, the Results

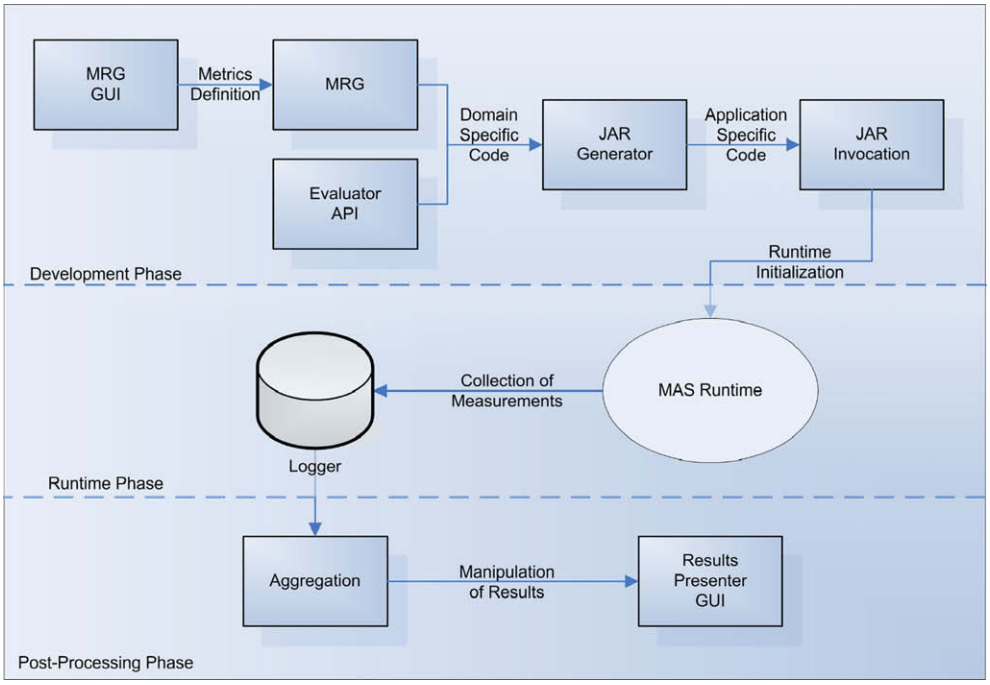


Fig. 2. MEANDER's overall architecture.

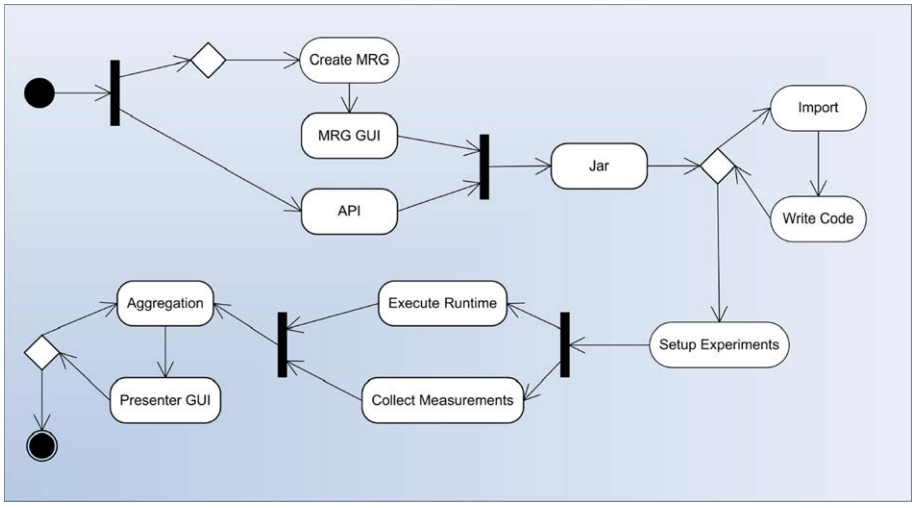


Fig. 3. UML activity diagram for the MEANDER framework.

Presenter GUI accommodates the evaluator needs for applying several statistical or other methods to select, organize and summarize the results of the evaluation process. A list of the GUI's main features is presented in Table 2.

Similar to the MRG Editor GUI, the Results Presenter manipulates files containing the required information structured in a well-defined XML format. Once a file is loaded, the evaluator is able to select the subset of the existing measurement data to be included for visualization, as well as to assign labels (such as *X*-axis and *Y*-axis) to groups of measurements. Finally, the evaluator may select a graph type from a list of available graph types.

4.3. Evaluation API

A key component of the MEANDER framework is the Evaluation Application Programmers Interface (Ev-API), which has been designed and implemented for measurement collection purposes. The API provides both complete and abstract classes in Java, serving as a general blue-print within the context of which the overall evaluation process takes place. More specifically, the Ev-API provides methods for identifying, collecting and communicating the required measurements, as well as facilitating the seamless incorporation of evaluation code into the MAS application. When moving to the next step of the APE methodology, this generic API is combined with the user-created MRG to automatically generate domain-specific code for carrying out any evaluation process at hand.

As illustrated in Fig. 5, the developed API consists of the *Eval* (root) Java package and four children packages, namely: (a) the *Logging*, (b) the *Communication*, (c) the *Module* and (d) the *Measurements* package. The two first packages are complete, while the other two contain both complete and abstract classes and

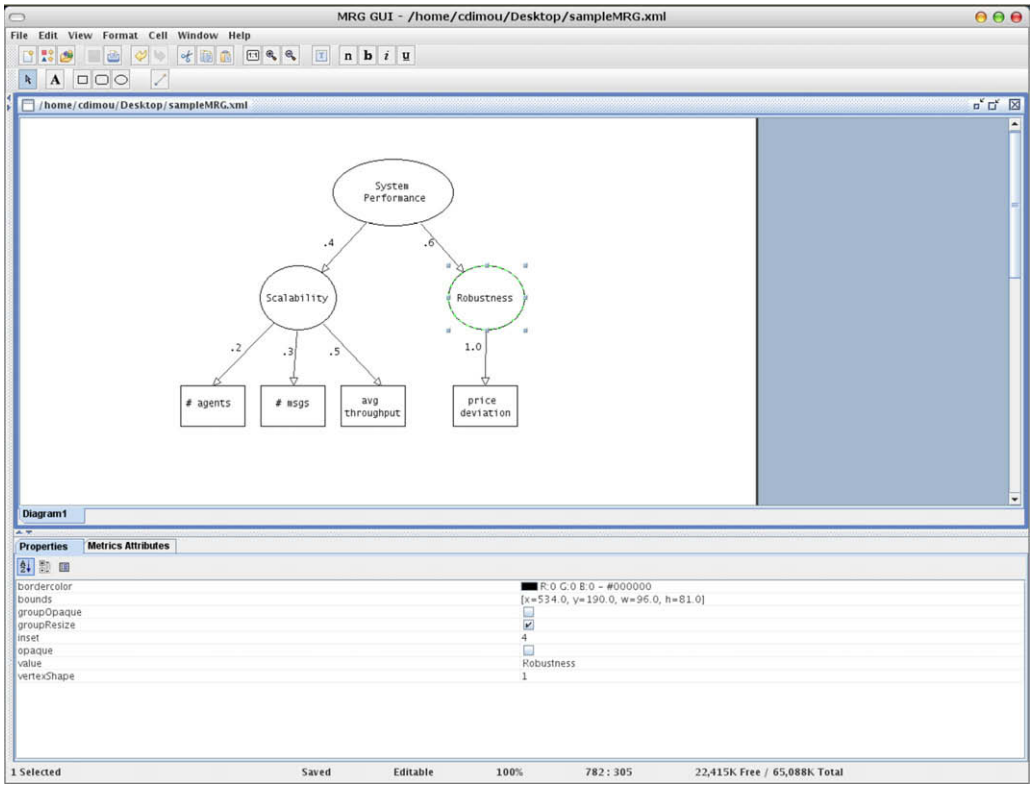


Fig. 4. The MRG Editor GUI.

Table 2
Basic menu options in the results presenter GUI

#	Menu category	Menu option	Description
1.	File		
1.1.		New	Creates a new blank document
1.2.		Open	Opens an existing document from an XML file
1.3.		Save/Save As	Saves the currently active charts into an XML file
1.4.		Close	Closes the working documents
1.5.		Export/Import	Exports/Imports data to/from PDF/PNG/CSV formats
1.6.		Print	Prints the currently active charts
1.7.		Page Setup	Configures page properties
1.8.		Exit	Exits the application
2.	Edit		
2.1.		Select All/None/ Inverse	Manages selections on the currently active graph
2.2.		Undo/Redo	Cancels/repeats previous actions
2.3.		Cut/Copy/Paste	Manages buffer selections
2.4.		Delete	Deletes currently selected chart items
3.	View		
3.1.		Zoom	Adjusts zoom level for current page
3.2.		Grid On/Off	Enables/disables grid lines on page
3.3.		Grid Size	Specifies number of grid lines
3.4.		Options	Configures various aspects of the GUI functionality
4.	Chart		
4.1.		Wizard	Initiates a wizard for creating a chart
4.2.		Gallery	Displays a list of available chart types
4.3.		Data Ranges	Specifies the data range to be displayed in a chart
4.4.		Chart Elements	Edits the properties of any chart element
4.5.		Properties	Edits the properties of the chart appearance
5.	Help		
5.1.		Contents	Displays a list of help topics
5.2.		About	Displays versioning and implementation information

methods. While logging and communicating the measurements is domain- and application-independent, the measurement phase (whether it is what to measure or how to measure it) is heavily dependent on the specifications of each domain and application.

The root *Eval* package, identified by the qualified name *gr.auth.issel.eval* contains classes that describe the basic notions of any evaluation process and are subsequently used throughout the Ev-API. In this package, we provide an implementation of the concepts of *Metric*, *Measurement*, *Experiment*, as well as their attributes and interconnections.

The *Logging* package, identified by the qualified name *gr.auth.issel.eval.log*, contains a set of classes that undertake the task of organizing the collected measurements in a structured format. Within the context of this framework, we have chosen XML-related technologies for representing and storing evaluation related information. Hence, the *Logging* package provides a full XML manipulation functionality, including creating, parsing, and editing XML documents.

The *Communication* package, identified by the qualified name *gr.auth.issel.eval.com*, provides the middleware infrastructure over which measurements are communicated between run-time entities and logging services. More specifically, each entity of the application designated to identify and collect measurements, is thereafter required to send this information over the network to an appropriate logging service. The *Communication* package adheres to the W3C Simple Object Access Protocol (SOAP) and provides methods for constructing, sending and receiving SOAP envelopes that contain the measurement information.

The *Module* package, identified by the qualified name *gr.auth.issel.eval.mod*, implements a middleware infrastructure that enables a seamless incorporation of evaluation-specific code into the MAS application. As further explained in Section 4.5, the generated code that collects and handles measurements must be separated from the application code to the furthest degree possible. The *Module*

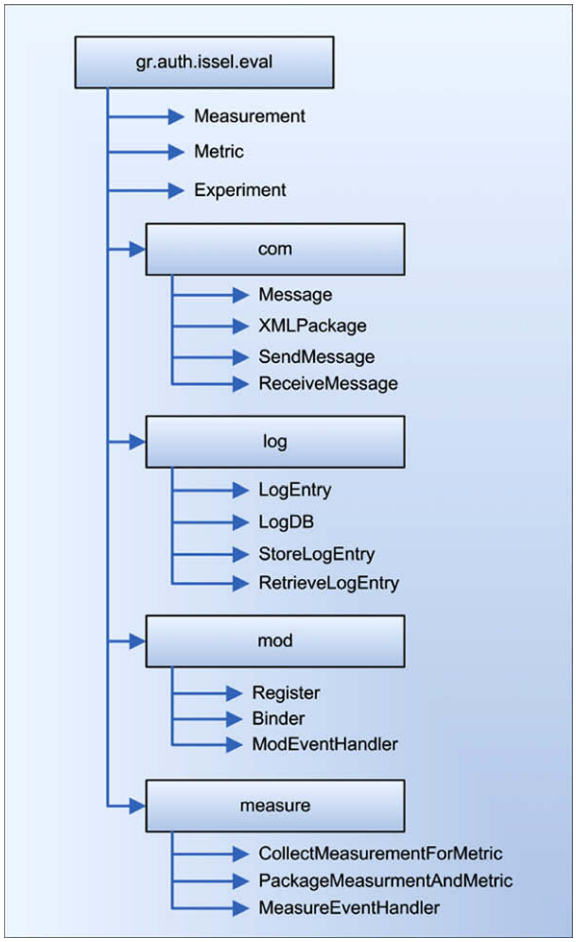


Fig. 5. API structure.

provides methods for registering the evaluation-specific code to the application and then for assigning of events on specific Java objects to metrics. For example, a desirable measurement for the *Bid Value* metric may be contained withing a message exchanged between two agents; the *Module* package allows a user to bind the Message object to the *Bid Value* metric and trigger a measurement event on the appearance of such message.

Finally, the *Measurement* package, identified by the name *gr.auth.issel.eval.measure*, contains a blue-print for the collection of measurements, with respect to abstract metrics. At this point, prior to having determined the domain-specific metrics, the API supplies the evaluator with abstract classes, interfaces and methods of the form *collectMeasurementForMetric()*. In the JAR generation phase that follows, these constructs are further specialized to produce *N* abstract classes, interfaces and methods of the form *collectMeasurementForMetric1()*,...,*collectMeasurementForMetricN()*, having assumed that the domain specific MRG contains the set of metrics [*Metric1*,...,*MetricN*].

4.4. Automatic JAR generation

This component of MEANDER deals with the automated generation of domain-specific code using the MRG and Ev-API. The goal of this component is to specialize the generic evaluation API and enforce correspondence between the specified metrics and the evaluation-specific API.

In order to provide this correspondence automatically, we have developed a mapping mechanism that receives inputs from both the Ev-API and the user-defined MRG. The menu option *"Generate JAR..."* mechanism is provided as a function of the MRG GUI. This GUI functionality transparently incorporates the JAR generation component into the integrated interactive environment for evaluation.

It must be noted that the generated code still contains abstract classes, interfaces and methods that have to be filled-in by the evaluator at the next step of the process. At this point, the generated evaluation code is *domain-specific*, always with respect to the selected metrics. If we assume that the evaluation of any single domain (e.g. electronic auctions in Supply Chain Management) is described by a standard set of metrics, then based on these metrics, Java code can be generated to be included in any application of the domain.

4.5. JAR invocation

In a simpler implementation, the evaluator would have to manually incorporate appropriate calls of the classes of the given JAR file, at various points throughout the code of his/her application. This approach adds extra burden to the evaluator, or even makes the entire procedure practically infeasible in the case the evaluator is not the developer of the application code.

In MEANDER, a modular approach to the invocation of the generated JAR file is adopted, in order to ensure that the incorporation of the appropriate functionality will require the minimum user intervention possible. The design is based on three basic points:

(i) *Import and module registration:* The evaluator imports the jar file and registers the module at the *main* or other initiating method of the application. Thus, the application is aware of the module, so that it will be dynamically loaded whenever needed.

(ii) *Module invocation means:* After registration, the application needs to be aware of the conditions under which the module is loaded. In our case, certain evaluation-specific methods will be triggered upon the appearance/creation of new Java objects of specific type. For example, in an auction application, if the required measurement information is contained in a *BidMessage*, then an appropriate method will be called upon the appearance of a *BidMessage* to the code of the auction seller agent. Thus, the evaluator has to provide a list of mappings between generated JAR methods and MAS Java objects.

(iii) *Compilation and addition of code:* At this point, by simply compiling the MAS application, the Java compiler indicates a list of classes, interfaces and methods that the evaluator must either implement or extend. Based on this skeleton code, the evaluator enters an iterative process of adding application specific code in order to indicate *how* the specified measurements are going to be collected. The process ends when all abstract elements are completed with the appropriate code, producing a ready-for-execution MAS.

4.6. Run-time measurements collection

During this phase several experiments are batched in a configuration shell script and are executed consecutively. While the augmented MAS is at run-time, the added fragments of code dynamically collect, log and send the required measurements to the logging service (described in the next session). The details of MAS run-time and measurement collection are better illustrated in Section 5, where MEANDER is used and the APE methodology is applied and demonstrated through a test case.

4.7. MAS Runtime Logger

The *MAS Runtime Logger* is the component where experimental measurements are pushed and stored for further post-processing. Since most MAS applications consist of agents or other software entities that are distributed over a computer network, the *Logger* operates as a web service that can be dynamically located and used on demand.

Fig. 6 shows the basic architecture of the *Logger*, which follows a simple web service model, extending the client/server model. The first step of the logging process concerns the location, identification and incorporation of the service by the MAS. This stage occurs at the initialization phase of the MAS, where the MAS searches for evaluation services by sending requests to a predefined UDDI web directory service.[2] If the *Logger* service is available, then a match is returned and a binding takes place between the service and the MAS. Each MAS receives a unique identification number for subsequent authentication purposes. If the *Logger* is unavailable, the measurements are stored locally in the filesystem of each node participating in the experiment.

The second step of the logging process deals with the actual utilization of the service. All entities collecting measurements in the MAS are now aware of the location of the logging service and forward each measurement to it. The service is then responsible for unpacking the delivered message, identifying and storing the measurement-related information. Logger's storage is organized in folder structure, with each top folder corresponding to a specific MAS implementation. Their sub-folders correspond to the various experiments conducted for each MAS application, identified by an experiment ID and the date of experiment execution. The folder structure is necessary to enable concurrent execution of different experiments, as well as to facilitate easy access to measurement data by the evaluator.

The stored measurements can be accessed and retrieved by any authenticated node over the network, via an appropriate *pull* service request. This particular service is employed by the *Aggregation* component, which is described in Section 4.9.

4.8. XML file formats

To ensure the seamless integration of information exchanged among the participating entities of a MAS (agents and services), we have defined two basic XML-related formats: (a) the Metrics Ontology and (b) the XML Measurements.

(i) The *Metrics Ontology* is a generic, structured representation of the Metrics Representation Graph. While the MRG provides a useful visual means for assisting human evaluators, the Metrics Ontology conceptualizes and represents metrics information formally. It defines the basic concepts related to metrics, as well as the relationships between them. For each application domain, the Metrics Ontology has to be instantiated with respect to its specific metrics and relationships. The resulting ontology instance is then readily available for manipulation in the MRG GUI and for incorporation into the other components of the MEANDER (Ev-API and the JAR Generating Mechanism). A simplified class diagram of the generic Metrics Ontology is presented in Fig. 7.

(ii) The *XML Measurements* refers to the formal structuring of the collected measurement during runtime. In this case, formal structuring works as a *lingua franca* throughout the system and is required for efficiently exchanging and handling all measurement values collected at runtime. *XML Measurements* are created as temporary data structures by runtime collecting entities, using the Apache WebAxis/SOAP API.[3] This information is then sent over the network to the Logger, which in turn stores it in a filesystem. Finally, the XML files are retrieved, processed and presented to the end-user by the Results Presenter GUI. Below, a sample fragment of XML Measurement is displayed, containing three measurement values for two metrics:

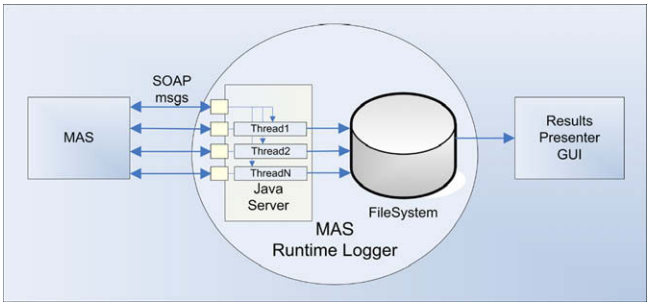


Fig. 6. MAS runtime logger architecture.

```
<?xml version="1.0'' encoding="ISO-8859-1"?>
   <sender id='agent42' uri='155.207.19.48'/>
   <receiver id='logService' uri='155.207.19.150'/>
   <application id='0701237612' name='Symbiosis.v1.1'/
   >
   <experiment id='Symbiosis.012' timeStarted='15:43:38'
dateStarted='03/03/2008'/>
   <measurementData>
      <metric name='usr'/ timestamp='120432'>
      <measurement value='0.472' scale='percent'/>
      <metric name='usr'/timestamp='126393'>
      <measurement value='0.431' scale='percent'/>
      <metric name='fcr' timestamp='128217'/>
      <measurement value='0.139' scale='percent'/>
   </measurementData>
```

4.9. Aggregation

The Aggregation component of MEANDER is responsible for automatically executing the selected aggregation method to the collected measurement data. Being a generic term, aggregation encompasses a variety of specific methods, ranging from simple operators to complex multicriteria-based decision-making. Within the context of MEANDER, we have chosen *Fuzzy Aggregation* methods for combining measurement values for related simple and complex metrics. Fuzzy Aggregation exhibits two main advantages:

- It provides a means of uniformly representing and combining heterogeneous information. With the use of fuzzy variables and fuzzy sets, we are able to uniformly treat measurement information that correspond to different intervals, scales, units or scale types.
- It provides linguistic performance indicators that are more easily understood by the human evaluator. For example, a domain expert would better comprehend the characterization *Moderate_low_accuracy*, than its numerical equivalent of, say, 42%.

In the current version of MEANDER, *Fuzzy Aggregation* requires manual translation between empirical measurements and fuzzy

[2] http://www.uddi.org/pubs/uddi_v3.htm.

[3] http://ws.apache.org/axis.

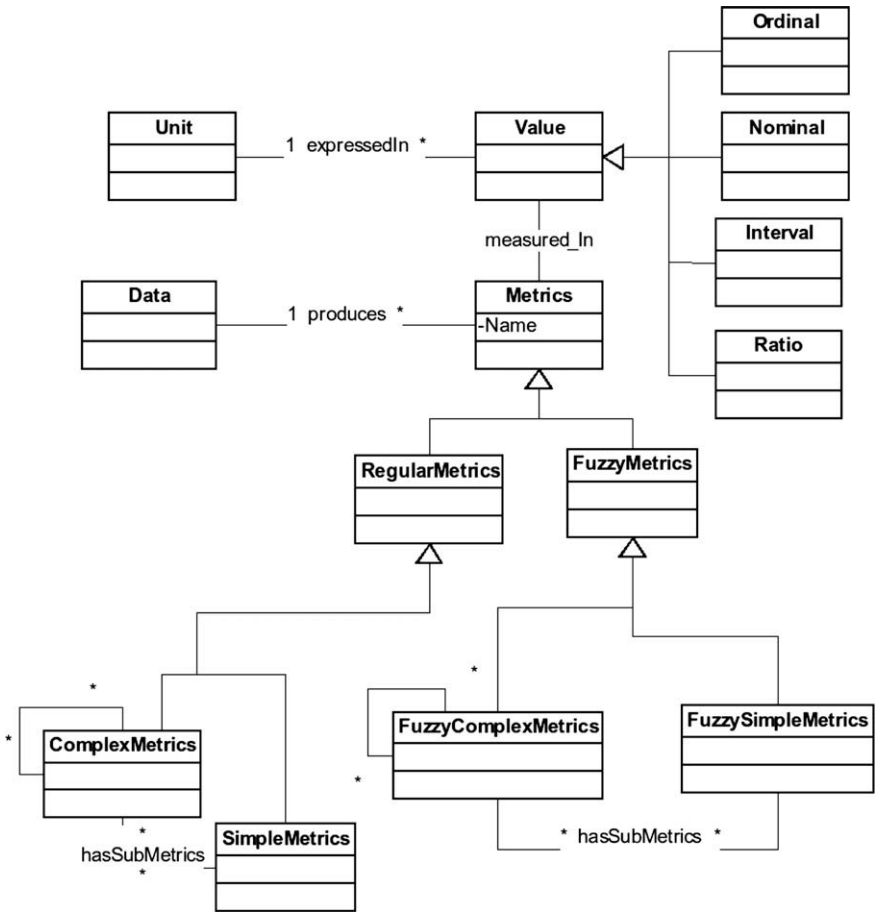


Fig. 7. The metrics ontology.

variables (see Section 6 for plans on automating this process). The domain expert is required to provide the following:

(i) A set of weights for every edge of the MRG, if not already present.
(ii) A characteristic fuzzy variable for every metric, either simple or complex, that exists in the MRG.
(iii) A fuzzy membership function for every fuzzy variable.
(vi) A set of fuzzy operators that apply on two or more sibling nodes of the MRG that combine into a single composite metric.
(v) A complete defuzzification mapping.

MEANDER incorporates fuzzy-specific functionality in various components, in order to help the domain expert to interactively or semi-automatically produce the required fuzzy elements. More specifically, the MRG GUI supports the introduction of weights in the graph, as well as defining fuzzy variables, correlating them to metrics and specifying fuzzy membership functions. Moreover, the Aggregation component automatically applies the selected fuzzy operators to the collected measurements.

4.10. Implementation and synopsis of MEANDER

All the above presented components of MEANDER have been based on Java and W3C-related components. Table 3 provides a synopsis of implementation and I/O aspects of the MEANDER components.

5. Demonstrator

In this section, we demonstrate the applicability and validate the efficiency of the proposed framework by using it in an already-implemented MAS, the *Symbiosis* simulation environment. We first describe the domain and implementation details of *Symbiosis* and then follow the methodological steps of the MEANDER framework.

5.1. Description of symbiosis

Symbiosis (Tzima, Symeonidis, & Mitkas, 2007) is a mutli-agent simulation framework that follows the animat approach, as proposed by Krebs and Bossel (1997). Animats represent autonomous, adaptive, learning entities that live and evolve in complex environments, in competition or collaboration with other animats. Symbiosis constitutes a virtual ecosystem, where two competing species of animats, preys and predators, co-exist and share the environment's limited resources. Preys consume natural resources, while predators go after prey. The goal of Symbiosis is to provide a simulation environment for testing and validating a number of emergent learning and adaptation techniques and the consequent effect of behavioral strategies, for both the prey and predator groups. The environment of Symbiosis is a $x \times y$ grid, where each cell can either be empty or occupied by:

– Natural resources that increase the energy of a prey agent, namely food;

Table 3
Component summarization

Component Name	Type	Impl.	Input	Output	Functionality
MRG GUI	GUI	Java, JGraph API	Existing MRG or none	Created/modified MRG	Create, load modify and export a MRG
Results GUI	GUI	Java, JFreeChart API	Measurement Data in XML	Charts of input in PDF, XML, CVS, PNG formats	Select data/Select and edit graphical representation style
Ev-API	API	Java	N/A	N/A	Implementation of fundamental Logging, Communication and Measurement classes
Jar Generator	Code generator	Java	Ev-API and MRG in XML	Java code with domain-specific evaluation code, in JAR format	Instantiates the Ev-API for the given domain, with respect to the input MRG
JAR Invocation	Module	Java	Generated JAR, MAS application and user provided code	Evaluation-enabled MAS code	Imports, registers and dynamically invokes evaluation-specific code into a MAS
Runtime	Executable	Java, XML, SOAP, Unix shell scripts	MAS Application executables and experimental parameters	Measurements in XML/SOAP	Executes the MAS runtime based on the experimental parameters and produces XML messages containing observed measurements
Logger	Service	Java/SOAP Axis	XML messages with measurements information	XML log files	Receives messages over the network, process and stores them in a filesystem
XML Representation	Data Modeling	Java Xerces/W3C OWL Protege 2000	Metrics or measurement information	XML files with MRG or measurement representation	Encapsulates input information in XML format
Aggregation	Executable	Fuzzy-J	MRG and XML file with raw measurements	XML files with aggregated information	Pulls, processes and aggregates measurement information based on the aggregation parameters stored in the MRG

– Natural resources that decrease the energy of an agent, namely an obstacle or a trap;
– A predator agent, or;
– A prey agent.

While natural resources are static, preys and predators are free to move in any neighbouring cell, aiming to maximize their energy, either by visiting energy enhancing cells (food cells for preys and prey cells for predators) or by avoiding energy reducing cells (predator cells for preys and obstacles and traps for both species). Each agent is born with an initial amount of energy, certain vision and communication capabilities, a decision-making mechanism and reproduction abilities.

The decision-making mechanisms employ genetic algorithms for the classification and evaluation of a set of action rules, either based on previous experiences or communicated from a neighbour of the same species. Finally, in order to reproduce conditions that occur in real-world environments, uncertainty has been introduced in Symbiosis, in the form of a parametrised vision error probability.

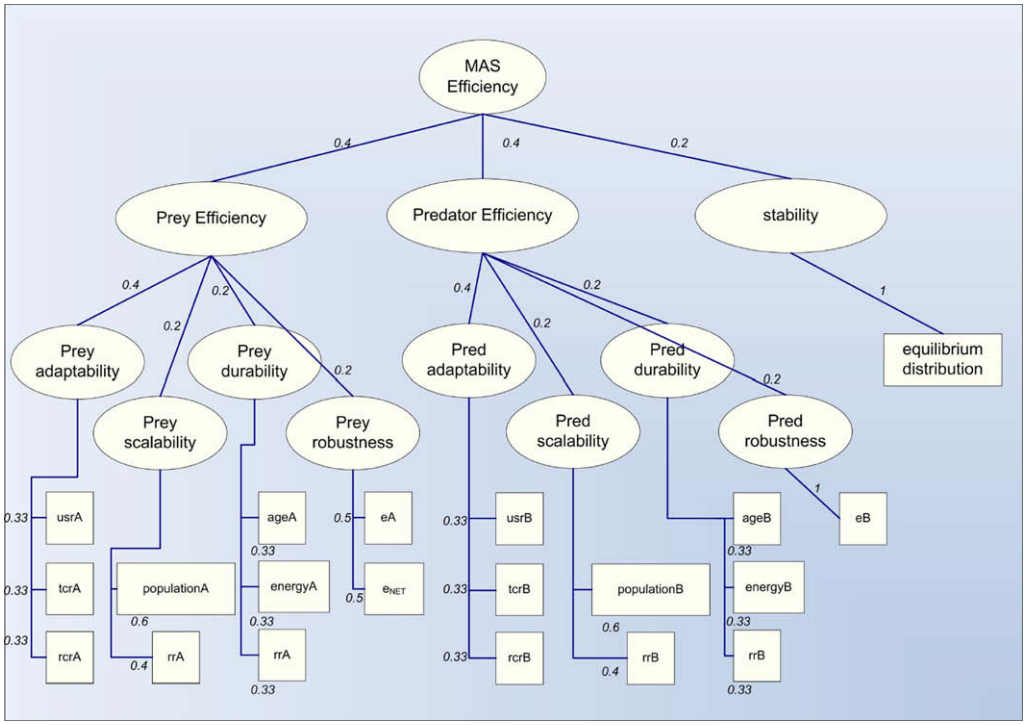


Fig. 8. The symbiosis MRG.

Table 4
Parameters for the executed experiments

Parameter	Experiment 1	Experiment 2
Environment settings		
Grid DV	50	50
Grid DH	50	50
Food percentage (%)	10	10
Obstacle percentage (%)	20	20
Trap percentage (%)	2	2
Food refresh rate 10	10	
Agent settings		
Number of rules	3000	3000
Number of preys	80	80
Number of predators	8	8
Initial prey energy	500	500
Initial predator energy	800	800
Ageing steps	2000	2000
Start reproduction age	400	400
Steps between reproduction	50	20
Communication	True	True
Steps between communication	200	200
Rule exchange	False	False
Rules percentage	0.1	0.1
Vision error (%)	5	5
Genetic algorithm constants		
Crossover rate	0.7	0.7
Mutation rate	0.01	0.01
GA steps	100	200
Naive predators	False	False
Naive preys	False	False
Max ages	10,000	10,000

Table 5
Collected measurements for Experiments 1 and 2

Metric	Experiment 1	Experiment 2
Environmental var	2.3334	2.3334
Resource avail	0.2866	0.2846
e	1.5782	1.4823
eNET	2.5025	2.3946
Population	17.3613	17.9495
fcr	38.2068	37.0400
tcr	1.2348	1.2296
usr	56.5224	57.0206
rr	9.8311×10^{-4}	0.0010

5.2. Evaluating symbiosis

5.2.1. Step 1: metrics definition

A typical evaluation procedure starts by defining the metrics for which measurements must be collected. Metrics definition and manipulation was handled using the MRG GUI. The domain-specific MRG that was created is illustrated in Fig. 8, with the appropriate Simple and Complex Metrics, as well as their corresponding weights.

5.2.2. Step 2: domain-specific evaluation code

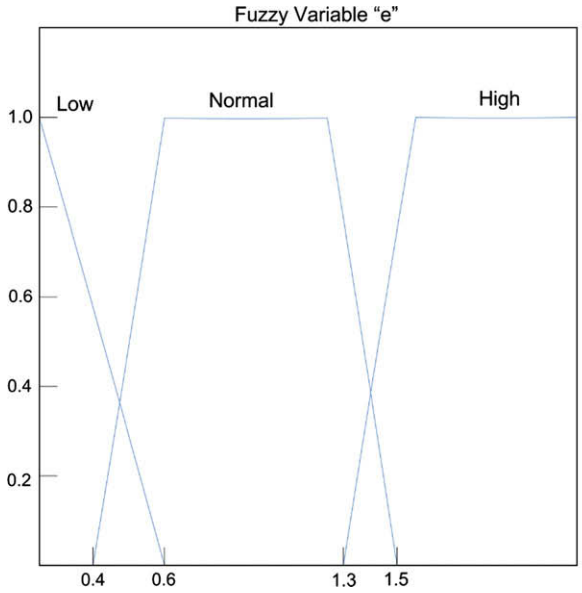
After defining the appropriate metrics and their relationships, the generic Ev-API was incorporated and combined with the Symbiosis MRG. The produced domain-specific evaluation code, in Java JAR format, comprised complete and abstract classes that could handle all the metrics present in the domain MRG.

5.2.3. Step 3: application-specific evaluation code

In this step, the JAR file was imported into the application and appropriate methods were invoked in crucial points of the application code, thus indicating the points at which the required measurement information resides. The imported JAR was first registered as a module so that the application was aware of its presence. It was subsequently linked to a number of specific Java objects and called dynamically upon the construction of any of these linked objects. We, finally, added some application-specific code for instantiating and implementing abstract classes and interfaces of the given JAR file.

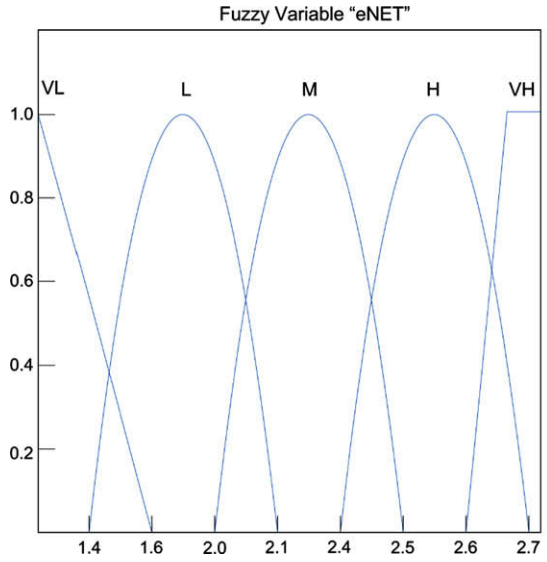
5.2.4. Step 4: runtime and collection of measurements

Having completed all parts of the code, we organized two sets of experiments and initiated the MAS runtime. While in execution, the imported evaluation-specific code collected all measurement information, packed it in XML messages and sent them to the *MAS Runtime Logger*. The goal of the experiments was to identify the effect of the Genetic Algorithm employed in the Prey Efficiency, as well as the performance of the entire MAS. The experiments were set up so as to differ only in the Genetic Algorithm Step parameter. The parameters of the executed experiments are summarized in Table 4, whereas the collected measurements for the Simple Metrics are presented in Table 5.



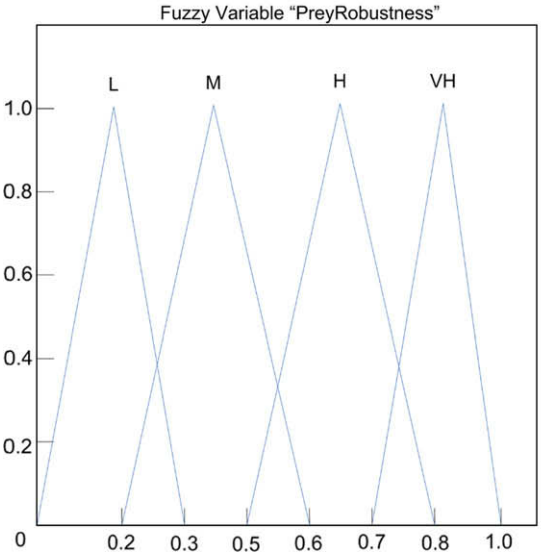
Fuzzy Values	Variable Range
Low	< 0.6
Normal	0.4 - 1.5
High	> 1.3

Fig. 9. Fuzzy variable *e* values and corresponding ranges.



Fuzzy Values	Variable Range
VL	< 1.6
L	1.4 - 2.1
M	2.0 - 2.5
H	2.4 - 2.7
VH	> 2.6

Fig. 10. Fuzzy variable e_{NET} values and corresponding ranges.



Fuzzy Values	Variable Range
L	0 - 0.3
M	0.2 - 0.6
H	0.5 - 0.8
VH	0.7 - 1.0

Fig. 11. Fuzzy variable *PreyRobustness* values and corresponding ranges.

5.2.5. Step 5: aggregation

In this step, we redefined a fuzzy variable as well as the corresponding fuzzy membership functions, for each metric in the MRG. In Figs. 9–11, we show the fuzzy variables and corresponding membership functions for the metrics *e*, e_{NET} and *PreyRobustness*.

It must be noted that an alternative approach for defining fuzzy metrics exists: the expert may choose only one fuzzy membership function per fuzzy variable. The advantage of this approach is simplification of calculation, since only one fuzzy set needs to be checked against crisp values. Our proposed approach, nevertheless, is more flexible, as it provides the evaluator with a range of options for fuzzy membership functions to choose from.

We proceeded by aggregating metrics *e* and e_{NET} into the Complex Metrics *PreyRobustness*. Since both metrics contribute equally to their parent metric, their aggregation was reduced to simple *IF... THEN* fuzzy rules, utilizing a fuzzy conjunction operator (fuzzy AND). The fuzzy rule set is displayed in Table 6.

Membership functions and rules were then applied to the actual measurements. This process is abstractly summarized in Figs. 12 and 13, where the two instantiated MRG for Experiment1 and Experiment2 are presented, containing the actual fuzzy values produced from the above tests.

Table 6
Fuzzy rules for calculating PreyRobustness

IF e is **Low** and e_{NET} is **M THEN** PreyRobustness is M
IF e is **Low** and e_{NET} is **H THEN** PreyRobustness is H
IF e is **Low** and e_{NET} is **VH THEN** PreyRobustness is H
IF e is **Normal** and e_{NET} is **VL THEN** PreyRobustness is L
IF e is **Normal** and e_{NET} is **L THEN** PreyRobustness is M
IF e is **Normal** and e_{NET} is **H THEN** PreyRobustness is H
IF e is **High** and e_{NET} is **VL THEN** PreyRobustness is L
IF e is **High** and e_{NET} is **L THEN** PreyRobustness is M
IF e is **High** and e_{NET} is **M THEN** PreyRobustness is H

After the aggregation process, a MRG instance is produced for each experiment. The detailed information that is contained at each node of the MRG can be used to compare different experi-

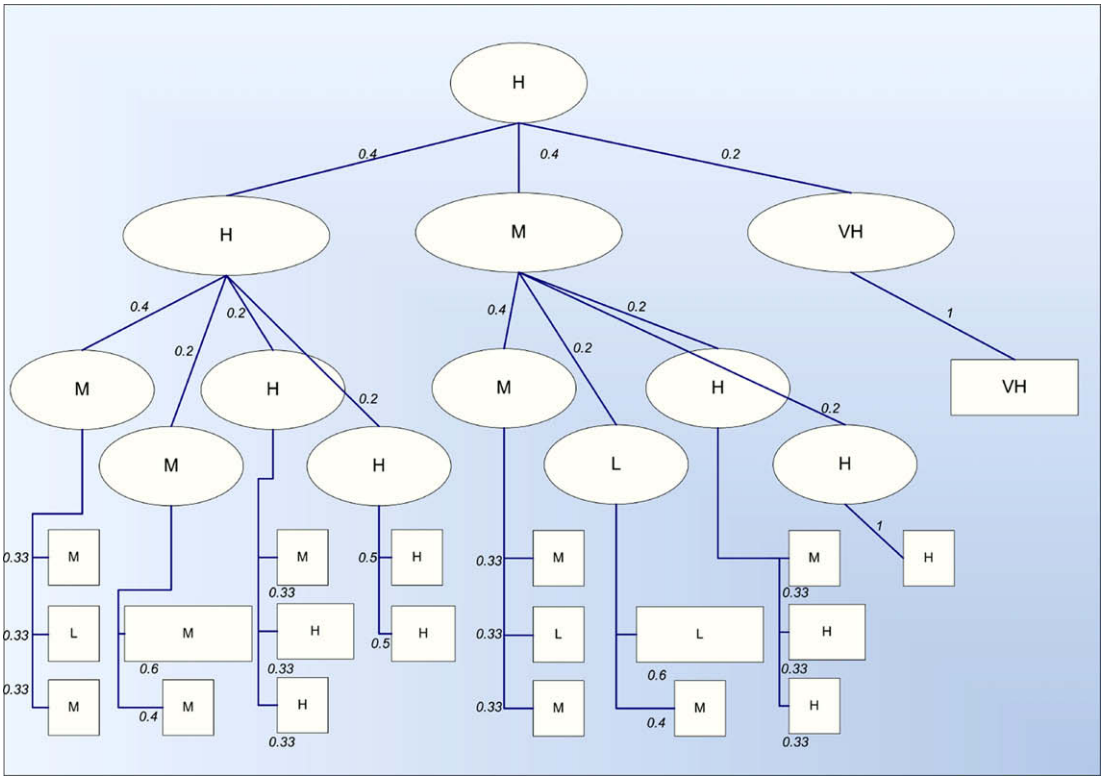


Fig. 12. Fuzzy MRG output for Experiment 1.

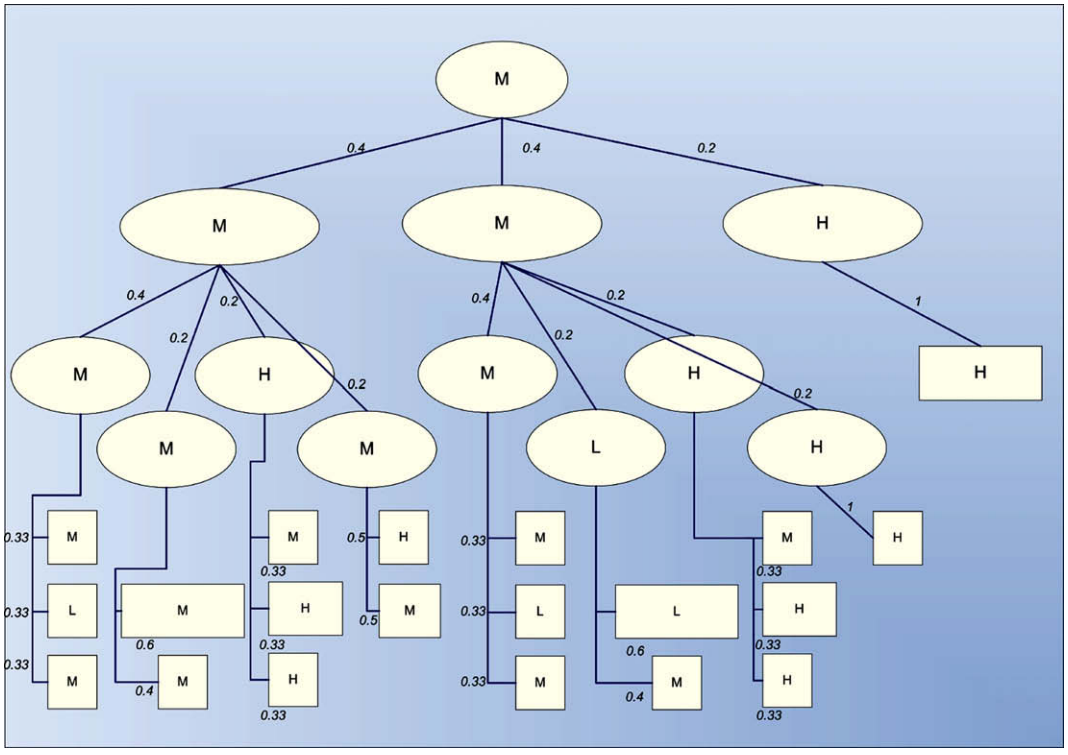


Fig. 13. Fuzzy MRG output for Experiment 2.

ments at different levels of granularity. For example, we may tune several parameters over the course of several experiments and then compare the results by examining either the root of the MRG or any other subgraph of the results.

5.2.6. Step 6: results manipulation

The final step of the MEANDER framework is to retrieve and present the aggregated measurements in graphical manner. For this purpose, we used the Results Presenter GUI, which interactively

specifies which measurements to include, how they are correlated and how they are graphically presented.

6. Conclusions

In this paper, we have presented MEANDER, an integrated framework for evaluating the performance of agent-based systems. The main achieved objective of MEANDER is to integrate performance evaluation processes into the entire development lifecycle, while clearly separating any evaluation-specific code from the application code at hand. MEANDER employs a number software tools to implement and automate the various steps of the APE methodology, including specification of metrics, collection of measurements and presentation of aggregated results to the user. The proposed framework was thoroughly described and applied to an existing MAS, namely Symbiosis. A step-by-step description of Symbiosis' evaluation with MEANDER was presented.

For future direction on this work, it must be emphasised that, since MEANDER only provides a complete framework for performance evaluation, it is necessary for developers and researchers in the agent community to provide their expertise by proposing domain metrics and results on diverse MAS applications. We underline the need for a *Performance Metrics Forum* where domain communities would propose and standardise MRGs for various application domains. It would also be valuable to collect experiences from actual results of applying the MEANDER to various existing or newly created applications.

Other directions that head towards a greater degree of automation within the MEANDER framework include the automated calculation of weights in the MRG. In the current version, a domain expert is required to provide specific, crisp weights for each metric node. We envision the next version of MEANDER improved in two directions with respect to metric weights: (a) the automatic generation of weights, based on historical evaluation data (for example, by utilizing neural networks) and (b) the specification of fuzzy, instead of crisp, weights. The latter would enable simpler metrics to contribute to more complex metrics by fuzzy factors.

Finally, we plan to integrate MEANDER into the Agent Academy (Symeonidis, Athanasiadis, & Mitkas, 2007), an environment for developing, training and retraining intelligent agents. Within this context, performance evaluation will play a crucially role in all parts of development or training of efficient agents.

References

Andrei, S., Chin, W. N., Cheng, A. M. K., & Lupu, M. (2006). Automatic debugging of real-time systems based on incremental satisfiability counting. *IEEE Transactions on Computers, 55*(7), 830–842.

Auguston, M., Jeffery, C., & Underwood, S. (2002). A framework for automatic debugging. In *ASE'02: proceedings of the 17th IEEE international conference on automated software engineering* (pp. 217–222). Washington, DC, USA: IEEE Computer Society.

Benevenuto, F., Ismael, J., Jr., & Almeida, J. (2004). Quantitative evaluation of unstructured peer-to-peer architectures. In *HOT-P2P'04: proceedings of the 2004 international workshop on hot topics in peer-to-peer systems (HOT-P2P'04)* (pp. 56–65). Washington, DC, USA: IEEE Computer Society.

Berki, E., Georgiadou, E., & Holcombe, M. (2004). Requirements engineering and process modelling in software quality management – Towards a generic process metamodel. *Software Quality Journal, 12*(3), 265–283.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., & Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems, 8*(3), 203–236.

Cai, Y., Huynh, S., & Xie, T. (2007). A framework and tool supports for testing modularity of software design. In *ASE'07: proceedings of the 22nd IEEE/ACM international conference on Automated software engineering* (pp. 441–444). New York, NY, USA: ACM.

Chu, H. D., & Dobson, J. E. (1996). *FAST: A Framework for Automating Statistics-based Testing.* Technical Report 564, Department of Computing Science, University of Newcastle upon Tyne.

Coelli, T. & Lawrence, D. (2007). *Performance measurement and regulation of network utilities*. Edward Elgar Publishing, Incorporated.

Compare, D., D'Onofrio, A., Di Marco, A., & Inverardi, P. (2004). Automated performance validation of software design: An industrial experience. In *ASE'04: proceedings of the 19th IEEE international conference on automated software engineering* (pp. 298–301). Washington, DC, USA: IEEE Computer Society.

Delgado, N., Gates, A. Q., & Roach, S. (2004). A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering, 30*(12), 859–872.

DeLoach, Scott A. (1999). Multiagent systems engineering: A methodology and language for designing agent systems. In *Agent-Oriented Information Systems '99 (AOIS'99),* 1 May 1999, Seattle, WA.

de la Riva, C., & Tuya, J. (2006). Automatic generation of assumptions for modular verification of software specifications. *Journal of Systems and Software, 79*(9), 1324–1340.

Dimou, C., Symeonidis, A. L., & Mitkas, P. A. (2007). *Soft computing for knowledge discovery and data mining. Chapter data mining and agent technology: A fruitful symbiosis*. Springer.

Flanagan, C. (2004). Automatic software model checking via constraint logic. *Science of Computer Programming, 50*(1–3), 253–270.

Godefroid, P., & Jagadeesan, R. (2002). Automatic abstraction using generalized model checking. In *CAV'02: proceedings of the 14th International conference on computer aided verification* (pp. 137–150). London, UK: Springer-Verlag.

Gudwin, R. R. (2000). *Evaluating intelligence: A computational semiotics perspective, 3*, 2080–2085.

Gummadi, K. P., Dunn, R. J., Saroiu, S., Gribble, S. D., Levy, H. M., & Zahorjan, J. (2003). Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP'03: proceedings of the 19th ACM symposium on operating systems principles* (pp. 314–329). New York, NY, USA: ACM.

Hellerman, H., & Conroy, T. F. (1975). *Computer system performance*. McGraw-Hill Education.

Huebscher, M. C. & McCann, J. A. (2004). Evaluation issues in autonomic computing. In *Proceedings of grid and cooperative computing workshops (GCC)* (pp. 597–608).

Huedo, E., Montero, R. S. & Llorente, I. M. (2005). An evaluation methodology for computational grids. In *HPCC* (pp. 499–504).

Huget, M.-P. (2004). Agent UML notation for multiagent system design. *IEEE Internet Computing, 8*(4), 63–71.

Jonker, C. M., Klusch, M., & Treur, J. (2000). Design of collaborative information agents. In *CIA'00: proceedings of the 4th international workshop on cooperative information agents IV, the future of information agents in cyberspace* (pp. 262–283). London, UK: Springer-Verlag.

Kitchenham, B. A. (1996). Evaluating software engineering methods and tool, part 2: Selecting an appropriate evaluation method technical criteria. *SIGSOFT Software Engineering Notes, 21*(2), 11–15.

Knowles, C. (2002). Intelligent agents without the hype: why they work best with well structured content. *Business Information Review*.

Krebs, F., & Bossel, H. (1997). Emergent value orientation in self-organization of an animat. *Ecological Modelling, 96*(1), 143–164.

Krippendorff, K. (1986). *A dictionary of cybernetics*. The American Society of Cybernetics.

Lurya, R. A. (2005). Automatic requirements elicitation in agile processes. In *SWSTE'05: proceedings of the IEEE international conference on software – science, technology & engineering* (pp. 101–109). Washington, DC, USA: IEEE Computer Society.

Rafla, T., Robillard, P. N., & Desmarais, M. (2007). A method to elicit architecturally sensitive usability requirements: Its integration into a software development process. *Software Quality Journal, 15*(2), 117–133.

Ruttens, P. (2002). Intelligent agents: Digital slaves or sci-fi dreamland? *City Information Group Newsletter*. Summer.

Sellink, A., & Verhoef, C. (1999). An architecture for automated software maintenance. In *IWPC'99: proceedings of the 7th international workshop on program comprehension* (pp. 38). Washington, DC, USA: IEEE Computer Society.

Smith, D. (2007). Toward automated software development. In *ASE'07: proceedings of the 22nd IEEE/ACM international conference on automated software engineering* (pp. 1). New York, NY, USA: ACM.

Stoller, S. D. & Schneider, F. B. (1996). *Automated analysis of fault-tolerance in distributed systems*. Technical Report TR96-1614, 19.

Symeonidis, A. L., Athanasiadis, I. N., & Mitkas, P. A. (2007). A retraining methodology for enhancing agent intelligence. *Knowledge Based System, 20*(4), 388–396.

Tzima, F. A., Symeonidis, A. L., & Mitkas, P. A. (2007). Symbiosis: Using predator–prey games as a test bed for studying competitive co-evolution. In *International conference on integration of knowledge intensive multi-agent systems, KIMAS 2007* (pp. 115–120).

Veerman, N. (2007). Automated mass maintenance of software assets. In *CSMR'07: proceedings of the 11th european conference on software maintenance and reengineering* (pp. 353–356). Washington, DC, USA: IEEE Computer Society.

Wang, M., Bennett, Jr. W., Lance, C., & Woehr, D. (2006). Performance measurement: Current perspectives and future challenges. Mahwah, N.J.: Erlbaum. 361 +xv. *Psychometrika, 72* (3) 455–456, September 2007. <http://ideas.repec.org/a/spr/psycho/v72y2007i3p455-456.html>.

Wooldridge, M., Jennings, N. R., & Kinny, D. (2000). The GAIA methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems, 3*(3), 285–312.

Zadeh, L. A. (2002). In quest of performance metrics for intelligent systems – A challenge that cannot be met with existing methods. In E. Messina & A. M. Mystel (Eds.), *Proceedings of the performance metrics for intelligent systems (PerMIS) workshop, NIST SP 990* (pp. 303–306).