Support for Analysis, Design, and Implementation Stages with MASDK

Vladimir Gorodetsky, Oleg Karsaev, Vladimir Samoylov, and Victor Konushy

St. Petersburg Institute for Informatics and Automation
39, 14 Liniya, St. Petersburg, 199178, Russia
{gor,ok,samovl,kvg}@mail.iias.spb.su

Abstract. In spite of much research and development on agent-oriented software engineering methodologies and supporting software tools, the problem remains of topmost importance. Many efforts are still needed to make such methodologies and software tools practically applicable at an industrial scale. This paper proposes extension of the Gaia methodology with a formal specification language, making it possible to implement Gaia as a model-driven engineering process supported by a corresponding agent-based software development environment, MASDK 4.0. The paper outlines MASDK 4.0 through the extended Gaia, and demonstrates the technology supported by MASDK 4.0 on the basis of a fragment of a case study on autonomous air traffic control.

1 Introduction

Over the last decade, different tools, software libraries, frameworks and program languages [1] have been developed for multi-agent system (MAS) development. Among them, the most widely used are Living SystemsRTechnology Suite [18] AgentBuilder [17], agentTool [5], Coguaar [8], JADE [2], INGENIAS IDE [12], etc. These differ in methodologies used (MaSE [6], Tropos [11]), Gaia [20], etc.), architecture of target applications (BDI, reactive architecture, etc.), and in levels of maturity achieved and classes of applications they are able to develop. However, there is no single methodology and software tool that is the best one.

Among other agent-oriented software engineering methodologies, the main peculiarity of Gaia is that it is not explicitly goal-oriented, although this feature does not mean that goal-oriented MAS applications of BDI architectures cannot be developed on the basis of Gaia. It focuses on organizational abstractions of applications with subsequent explicit separation and specification of internal and external behaviour of agents composing MAS, and exactly this feature determines its specificity as opposed to many other existing methodologies. External behaviour determines agent interactions in various use cases that is specified in terms of interaction protocols and constrained by organizational rules. Explicitly introduced protocols actually make it much simpler to represent collective behaviour of agents, and the behaviour itself is more predictive in comparison with the goal-oriented BDI approach. In the BDI approach, the emerging behaviour of individual agents and collective behaviour results from rich knowledge and reliable beliefs of the particular agents, as well as from sophisticated inference mechanisms that need to use, as a formal basis, modal and temporal logic. In many cases, the BDI approach works well, but the problem of complexity should be managed carefully.

M. Luck and J.J. Gomez-Sanz (Eds.): AOSE 2008, LNCS 5386, pp. 272–287, 2009.

In contrast, Gaia attempts to transfer MAS development complexity into more careful analysis of the system's organizational issues, and the explicit design of the MAS interaction model thus makes collective behaviour of agents more predictable and understandable. In general, such an approach simplifies agent-oriented software engineering and there exist many classes of applications where Gaia-based technology could outperform purely goal-oriented ones.

The objectives of this paper are twofold. One is to extend Gaia in order to make it more applicable by enriching it with a formal specification language that can make it possible to practically implement Gaia as model-driven engineering. The second objective is, based on the extended Gaia and a design process and formal specification language, to introduce the MAS development environment that implements a graphical MAS development process, according to the extended Gaia methodology. Accordingly, the paper is organized as follows. Section 2 presents a general view of the Multi-agent System Development Kit (MASDK 4.0) components and their interactions in the development procedure. Section 3 sketches Gaia as it was proposed by the authors [20] and presents generally in what aspects it is extended in this paper. Section 4 introduces the Agent System Modeling language, ASML, that supports model-driven engineering technology of the Gaia implementation. Section 5 describes the stage-by-stage development process supported by MASDK 4.0 bridging it with Gaia and describing the particular products. This is done using a fragment of a case study for an autonomous air traffic control system. Section 6 surveys related work on the existing extensions of Gaia. The conclusion summarizes the main paper contributions, emphasizes the advantages of the proposed Gaia extensions and the MASDK 4.0 development environment implementing Gaia. Future work is also outlined.

2 Architecture of MASDK 4.0 Environment

MASDK 4.0 is a software tool implementing an extended version of the Gaia methodology. It is provided with a graphical and formal specification language that supports thorough and consistent conceptual analysis, detailed design, code generation and deployment of target multi-agent applications, including those operating in dynamic and Peer-to-Peer (P2P) environments.

The basic components of MASDK 4.0 and their interaction are shown in Fig. 1. The *Agent-based System Modeling Language* (ASML), based on UML, is exploited for in-progress-design specification of MAS formal models. ASML-based specification of MAS is supported by a *Visual Design Environment* (VDE) component providing for user-friendly graphical notation of ASML. The final formal model or model-in-progress is stored in the *MAS Model Description* (MMD) component. The resulting MAS formal model serves as input for producing the agent classes' source code. This functionality is performed by the *Source Code Builder* (SCB) component. Two more components, *Generic Agent* (GA) and *Agent Platform* (AP), are implemented as reusable solutions (source code). The code of the GA component realizes application-independent functionality of agents behaviour, implemented as a reusable library producing the agent classes' source code. The AP is FIPA compliant developed according to the reference model [8], and realizes distributed "white" and "yellow" pages services and provides

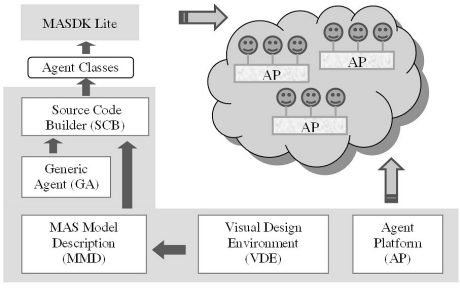

Fig. 1. MASDK environment components and their interaction

for communication. The AP is a self-dependent software component that can be exploited separately from MASDK 4.0. Its description and downloadable run-time code can be found in [15].

MASDK Lite is an auxiliary component intended for agent deployment. Specification of agent classes (source code of agent classes) is used as its input. The agent instances deployment requires identification of their names and addresses, and, when necessary, specification of their initial mental models and services they possess. MASDK Lite, like AP, can be used separately from MASDK 4.0 for application maintenance.

MASDK 4.0 supports analysis, architectural and detailed design of MAS applications by Gaia, but also software implementation using the SCB component, and deployment using MASDK Lite. The ASML-based formal specification of MAS models is used by the SCB component to produce source code of agent classes. The source code contains fully specified behaviour logic and mental models of agent classes. Therefore, the implementation stage is limited to encoding of the agent class activities. This code should be manually developed by programmers using the MS VC++ environment.

3 Methodological Basis: Extended Gaia

As stated above, the Gaia methodology for agent-oriented software engineering [20] is focused on organizational abstractions of complex systems with explicit separation and specification of internal and external behaviours of agents. External behaviour determines agent interactions in various use cases, specified by interaction protocols. Interaction protocols are distributed algorithms performed by a subset of agents each of which performs, within particular protocol, a specific role. The scenario of the role performance and some other role activities correspond to what is above called “internal agent behaviour”. Accordingly, Gaia explicitly determines the decomposed system organization components, the subtasks to be solved by each component in various use cases (roles), and interaction protocols associated with the use cases. The results of the Gaia analysis constitute the input of the subsequent architectural and detailed design.

In fact, Gaia, as described by the authors [20], specifies a general methodological framework for analysis and architectural design, whereas methodology applicability needs detailed development and formal support for all aforementioned stages and implementation, and deployment issues as well. In other words, applicability of Gaia requires

its technology-oriented extension. Extensions of various kinds were proposed in [10], [3], [13], [14], etc. reviewed in Section 6. This section outlines the applicability-oriented Gaia extensions proposed in this paper that then constitute the methodological basis for the agent-oriented software engineering technology fully supported by MASDK 4.0 components presented in Fig. 1.

3.1 Analysis

Subdivide System into Sub-organizations. The objective of this stage is description and analysis of the organization aiming at discovery of appropriate decomposition of the whole organization into more or less weakly-coupled sub-organizations.

Identify Environmental Entities. This activity aims at detection of environment conditions, constraints and environment entity interfaces. Examples of entities include databases, external services, user interface, sensors (e.g. controllers of an assembly line), etc.

Discover Roles, Create their Preliminary Internal and Interaction Models. Role discovery is fulfilled by an expert. Preliminary role model creation includes description of their *permissions* (e.g. regarding interaction with environment, etc.), *responsibilities and activities* together determining the role internal behaviour. The responsibilities are specified by *Liveness Expressions* (LE) using the formal language *Fusion*. The discovered roles are represented by the *Role Schemata*. Role interaction protocols are also identified. These models are preliminary and may be refined later.

Determine Organizational Rules. Organizational rules determine global system behaviour policy that must be satisfied by all the agents. E.g., security policy rules regulate access to confidential information. Safety policy rules in air traffic control determine admissible movements of the aircrafts in various situations when separation distances between aircrafts may be violated. In Gaia, organizational rules are introduced informally, constituting inputs for the architectural design stage.

An extension of the Gaia analysis proposed regards using a specific formal specification language, *Agent-based System Modeling Language* (ASML), supporting all the analysis-related activities and specifying formally the results. It is described below. These extensions are fully supported by MASDK 4.0.

3.2 Architectural Design

Gaia determines two kinds of activities that should be performed at this stage.
Select Organizational Structure. Decomposition is a subject of the analysis stage, while organizational structure is developed at the architectural design stage so that various sub-organizations can be structured differently. E.g., some components can be structured in P2P mode, whereas others can be hierarchical.
Final Role and Interaction Models. The organizational structures can require refining of the preliminary role schemata (i.e., models of roles and interaction protocols).

The proposed Gaia extension supports formal specification of the design results using ASML and supported by MADK 4.0 (see below).

3.3 Detailed Design

The core of this stage is identification and detailed design of agent class models and services. Each agent class is allocated one or several identified roles. The products should correspond to a fully developed agent architecture, including its services, providing the specifications necessary for software implementation.

Agent Model. All the agents allocated the same roles determine a particular agent class. Detailed design of agent class software architectures is the main subject of this stage in Gaia, but Gaia gives no concrete methodology for this activity.

This paper extends Gaia in two aspects. First, it proposes the use of ASML for specification of the in-progress and final products of the agent model detailed design and, second, it proposes a *generic agent* architecture implemented as reusable software within MASDK 4.0.

Service Model. Gaia defines agent services vaguely. In this paper, an agent service is understood as a single activity or a sequence of activities (a scenario) that provides an agent with functionalities that do not necessarily fall into the block of functions invoked by an agent interaction protocol. They can be composed of LEs, proactive behaviour invoked by internal agent state and/or by the state of the environment (e.g. when agent requests for resources from environment).

Implementation and deployment. These very important stages are outside the scope of Gaia. The proposed extension, and the corresponding software engineering means implemented in MASDK 4.0 are described below.

4 Introduction to ASML Language

The ASML language is based on UML notation. It is supported by a user interface providing, for ASML, a graphic notation that allows representing and editing of MAS models via the VDE component (Fig. 1) by creating diagrams formally representing in-progress and final analysis and design solutions. The set of diagram types, their interrelations and use at the respective stages of MAS design correspond to Fig. 2. Below we provide a brief outline of ASML.

MAS macro model diagram. This diagram type is specified using ASML concepts: *tasks, agent roles*, *protocols, active entities (components),* and *agent class.*

The notion of *task* is used for assigning names to use cases. *Agent role*, in the macro model, is identified by the name and template containing the list of LE names defining the role internal behaviour and also the list of identifiers of the rules triggering proactive agent behaviour[1] (triggering rules, for short). Each of these rules is linked to the LE it triggers. *Active entity* specifies the interface of agents to external components that are not agents. Examples of active entities are interfaces to sensors, effectors, etc. The macro model is constituted of instances of the above notions and the relations between them. The following types of relations are used in ASML: *task – is initiated by – triggering rule / functionality of active entity; LE – is initiated by – protocol / triggering*

[1] Here proactive behaviour is initiated by something other than a protocol.

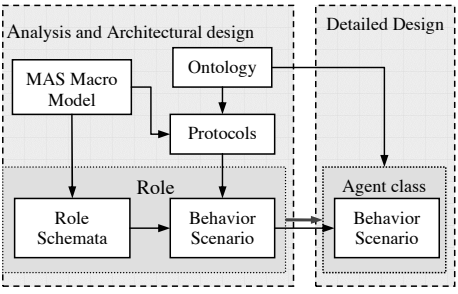

Fig. 2. Development process models and their interactions

rule; protocol – is initiated by – active entity / Role / LE / functionality of active entity; functionality of active entity – is initiated by – protocol; agent class – plays – role. Corresponding diagrams are intended for specification of the organizational structures and interactions, and a MAS application is specified by a single macro model that can contain several diagrams.

Protocol diagram. ASML provides the following protocol specification concepts: *(role) lifeline of the protocol participant, communication act, (role) lifeline end, alternative combined by a fragment* comprising two or more *operands*. Use of auxiliary dialogs allows specifying communication acts in terms of *cardinality of the protocol respondents, cardinality of the respondents* for each operand of combined fragments as well as for each message respondent. Communication act is specified in ACL language [9]. Specification of message content is done in terms of ontology concepts, while protocols are specified using the diagram editor. These diagrams specify interaction protocols among the roles and among the roles and active entities.

Role scheme. Detailed specification of each LE is performed at two abstraction levels using *Role scheme* and *behaviour scenario* diagrams respectively. The former is used to specify LE decomposition if necessary and scheme of inter-role behaviour coordination. *Role scheme* diagrams are represented in terms of such notions as *scenario, event, triggering rule* and *protocol.*

The LE decomposition is performed using two types of the designing rules, mandatory and optional. The former correspond to a case when the LE is linked to protocols or/and to triggering rules. In this case, the LE is decomposed into simpler scenarios with one-to-one relation to protocols, and triggering rules should be linked to a simple scenario. Optional rules allow experts to elaborate more detailed decompositions via adding new simple scenarios. The role scheme comprises scenarios reflecting LE decomposition and relations of type *scenario-uses-scenario.*

Each role's LE is run within a separate control thread that admits concurrency of role performance but this may require coordination. It is done using the notion of *event* and additional relations, i.e. *event – is generated by – scenario, event – is used by – scenario* and *event – is used by – triggering rule.* The relation of the first type identifies the behaviour scenario generating the corresponding event. The next type is intended

for specification of the scenario interrupt processing. The same relation determines the events initiating continuation of the interrupt processing. The last type of the relation determines the conditions initiating triggering rules.

Role behaviour scenario. Diagrams of this type are used for specification of simple scenarios as a set of nodes and transitions between them, while representing behaviour logics of each scenario. ASML introduces the following self-explanatory types of notions defining classes of scenario nodes: *activity, complex activity, generating / waiting event, event handling, sending / waiting message, message handling, interface with agent platform, control node,* etc. E.g., in the nodes of type *control node*, different variants of the behaviour scenario performance continuations are specified. In the nodes representing a complex activity, the conditions invoking the behaviour embedded scenarios are determined, etc.

Ontology. Ontology diagrams are used for description of domain notions and relationships between them. Ontologies are used as the language for message content specification and, in the detailed design of agent classes, it is used to specify their behaviour scenarios in terms of model variables and attributes. Ontology relations are specified using three standard relations types: *generalization (inheritance), association and composition.*

Agent class behaviour scenario. When an agent class is allocated roles (at architectural design) the former inherits the roles' behaviour scenario list of these roles. Detailed design assumes formal specification of agent variables, scenario interface, scenario variables and scenario node calls represented using scenario and agent variables.

5 Support of Gaia in MASDK 4.0

5.1 Case Study: Autonomous Air Traffic Control

Due to the ever increasing intensity of air traffic and stiffening safety requirements, air traffic control relying on purely human-based control needs to reconsider its basic organizational principles. According to current opinion, some complex real time control responsibilities of air traffic control operators should be assigned to aircraft software to make control more autonomous. The case study of an agent-based autonomous air traffic control system, used for explanation of the developed software tool, MASDK 4.0, is outlined below.

The air traffic safety is provided by two measures. The first is structuring the airport airspace according to a topology that determines all the admissible trajectories of aircraft arrivals, landing and departure. It encompasses two zones (Fig. 3): (1) *arrival zone* and (2) *approach one.* The arrival zone comprises *arrival schemes*, which begin with entry points and are specified as sequences of legs ending with a *holding areas*. The airport airspace (AA) topology also determines admissible echelons, i.e. admissible altitude ranges for passing through leg exit points. The AA topology also contains departure schemes but departure control is omitted in the case study. An example of an AA topology for JFK airport of New York City (NYC) is depicted in Fig. 3.

The second measure of the air traffic safety provision assumes the separation standards that determine the minimal admissible distances between aircrafts along each of

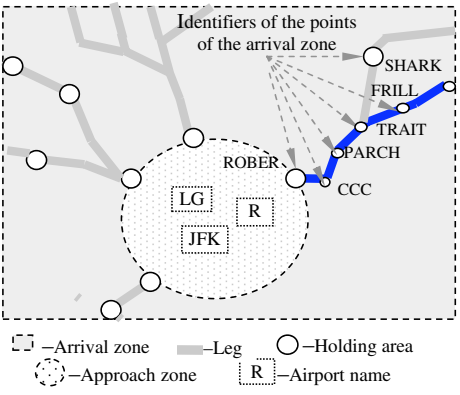

Fig. 3. Airspace topology within NYC area (horizontal projection), and arrival / approach zones

three spatial dimensions. These standards may be different for various air traffic-related situations. In the case study, meeting the separation standards is achieved by using a rule-based distributed safety policy that has to be followed by every aircrafts operating within airport airspace.

The idea of the autonomous air traffic control in the arrival zone is to delegate, to the aircraft's pilot-assisting software, the right to autonomously compute the safe landing trajectory, predict potential conflicts (violation of the separation standards) with other aircraft and to autonomously resolve these conflicts using a distributed safety policy and peer-to-peer negotiations with the potentially conflicting aircraft.

5.2 MASDK 4.0 Products Supporting Gaia

The whole air traffic control system (ATC) organization contains two sub-organizations: ATC in arrival zone and ATC in approach zone. Below, the latter sub-organization is not considered. The environmental model includes the model of the AA topology, the real time model, and visualization model of the whole situation in the AA. These models are composed in the component called *Simulation server* that, in terms of ASML, is an *active entity.*

In the considered fragment of the ATC system, the set of use cases is developed for individual aircraft, since all operate equally. The following use cases are considered below: *ATC in arrival zone*, corresponding to elaboration (computation) of the landing plan by an aircraft; *Aircraft grouping*, intended for rough evaluation, by an individual aircraft, of a subset of other aircraft that potentially can be the sources of conflicts. The use cases involving the *Simulation server* include *Airliner initialization* when an aircraft enters the arrival zone, instead of *Simulation* when the current time variable is increased for one simulation duration. Each of the above use cases assumes interactions according to the corresponding protocols.

Role discovery results in one role, *Pilot (aircraft) assistant* (PA). It should participate in these use cases to be responsible for autonomous planning and scheduling of landing trajectory, prediction of potential conflicts with other aircrafts and conflict

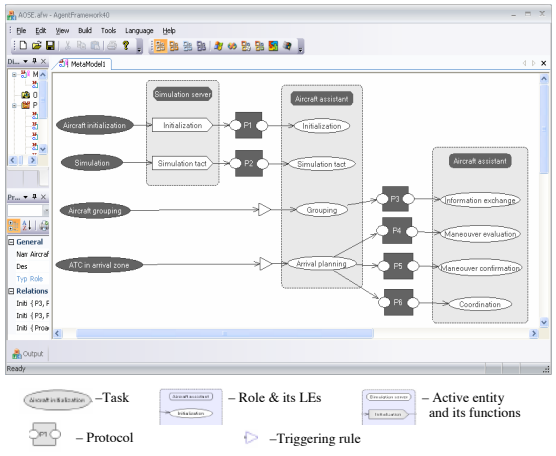

Fig. 4. Example of MAS macro model diagram

resolution through negotiation and reaching agreement. Accordingly, the listed use cases determine the list of LEs of the PA role while determining its interaction model and list of protocols. The single agent class, *PA-agent class* assigned the PA role is introduced.

An important component of the macro model is a set of *organizational rules*. In this case study, organizational rules represent the distributed safety policy, but since analysis of this is not the focus of this paper, its description is omitted.

MASDK 4.0, implementing ASML, supports all aforementioned analysis-related activities with graphical means. Fig. 4 shows the MADK 4.0 product, i.e. graphical notation of the *macro model* of agent-based autonomous ATC system and its components, representing PA role scheme and environment model, *Simulation server*. It is worth noting that this macro model is not only "a picture", but is also the formal model specified in ASML, and used as input for the next development stages. In general, architectural design should result in selection of the organizational structure, and refinement of the roles and interaction models. Below, these Gaia activities and their support in MASDK 4.0, as well as the products, are considered by example.

Organizational structure. The autonomous ATC assumes negotiations of PA roles in several use cases with no mediation by a centralized server, i.e. with no hierarchy. This requires use of P2P negotiations of the PA roles, so that agents have to negotiate using the distributed P2P platform that is a component of MASDK 4.0 environment [15].

Interaction model architectural design. Fig. 5 depicts an example of the protocol specification. It represents the protocol P4 identified in the macro model (Fig. 4). According to the latter, this protocol is used by a PA role when it agrees with maneuvers proposed by the same role of other PA agent class instances. In the diagram, these roles are denoted as *I-Airliner* (the protocol initiator) and *P-Airliner* (participant, or respondent of the P4 protocol) respectively. In the example, only one respondent of the P4 protocol

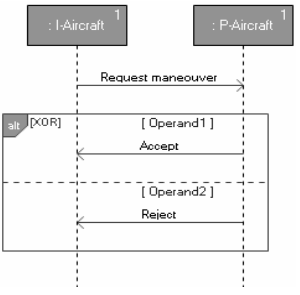

Fig. 5. Example of a protocol diagram

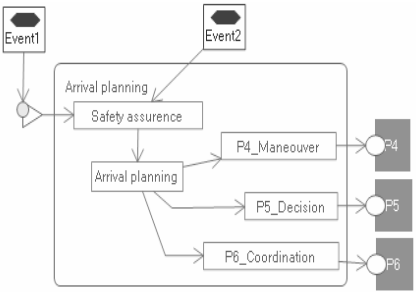

Fig. 6. Example of a scenario diagram

exists. It receives a request from *I-Airliner* to perform some maneuver, either accepting the request or rejecting it, while sending a corresponding message, either *Accept or Reject* respectively (Fig. 5). In Fig. 5 the protocol description consists of two participants and includes one combined fragment. In general, the protocol can consist of several participants and can include several nested combined fragments. It is worth noting that UML *Sequence diagrams* are used as prototypes of the protocol diagram used in MASDK 4.0. The limitation of such diagrams is that they do not allow for specifying nested protocols so far. This limitation is actually made up for by other types of diagrams aimed at specification of the role scheme.

Thus, the *second important product* of MASDK 4.0 in architectural design is the set of formal protocol models composing the MAS interaction model that is specified in terms of UML-like Sequence diagrams. Let us note that this specification is later used in automatic generation of the corresponding LE scenario scheme represented via the set of its nodes and transitions between them.

Role model: Liveness expression specification. Role scheme diagram. Architectural design of an agent role consists in specification of its LEs. It is done using two types of diagrams, *Role scheme* and *Behaviour scenario*. The former diagrams are used for decomposition of the LE into several simpler behaviour sub-scenarios, as well as for specification of the inter-roles behaviour coordination scheme.

Fig. 6 explains by example the decomposition rules described above. It represents the PA *Role scheme* diagram for the LE *Arrival planning* (Fig. 4). According to the macro model, this LE is capable of proactive behaviour, and participates in three interaction protocols, $P4$, $P5$ and $P6$. Accordingly, this LE scheme is decomposed to four simple behaviour scenarios denoted as *Safety assurance, P4_Maneuver, P5_Decision* and *P6_Coordination*. The fifth simple scenario, *Arrival planning*, is added to provide a specification of the logically complete sub-scenario that is the computation of the admissible set of landing plans. The arrows connecting the scenarios are interpreted as *"scenario – invokes – scenario"* imposing order on performance of simple scenarios.

E.g., according to the diagram given in Fig. 6, the simple scenario *Safety assurance* is initiated by a proactive behaviour rule. This rule is triggered when *event 1* has happened, for instance. The latter is generated as an output of another LE, *Simulation* (Fig. 4), when an airliner is approaching the airport airspace point connected to holding area. In

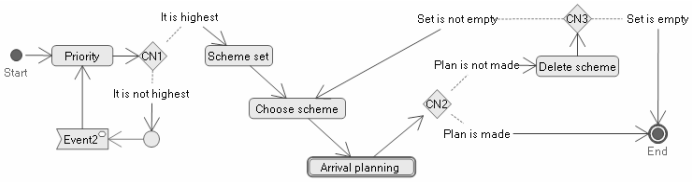

Fig. 7. Example of scenario diagram

that point, the PA role has to compute a flight plan for moving within the next sector of the arrival scheme. If, at the same time, other PA-agent instances begin computing their flight plans within the same sector then, according to the distributed safety policy, these PA-agent instances have to compute their priorities using the *Safety assurance* scenario. These priorities determine the order of their entry in the same sector, while granting entry permission to the highest priority aircraft. If an aircraft is not granted this permission, it interrupts execution of the *Safety assurance* scenario and waits for *event 2*, which arrives when the situation is changed in a way leading to a change of the aircraft priorities. In this case, the *PA-agent* instance repeats computation according to the above described scenario and, in case it has highest priority, continues performance of the LE and invokes the *Arrival planning* scenario, determining its behaviour within the next sector.

Role model: Liveness expression specification. Behaviour scenario diagram. Behaviour scenario diagrams are used for specification of simple scenarios. This type of diagram is an extension of UML activity charts [19]. Two types of simple scenarios are discerned, associated and not associated with the interaction protocols, and specified differently. Fig. 7 presents a self-explanatory example of a scenario that is not associated with any interaction protocol.

Behaviour scenarios of the second type, i.e. associated with participation of the role in an interaction protocol, must be consistent with the protocol specification, (protocol specification determines scenario behaviour logic). In the MASDK 4.0 environment, this dependency is automatically supported by two mechanisms. The first provides for automatic generation of scenario behaviour logic, and the second allows for preservation of this logic during further development (refinement) of the scenario by the designer. Both these mechanisms are explained in Fig. 8 where description of the *P4_Maneuver* scenario is shown. This scenario specifies the performance of the PA role (as initiator) in interaction protocol P4 (Fig. 4) represented by the diagram in Fig. 5.

Fig. 8. Diagram example of scenario related to protocol

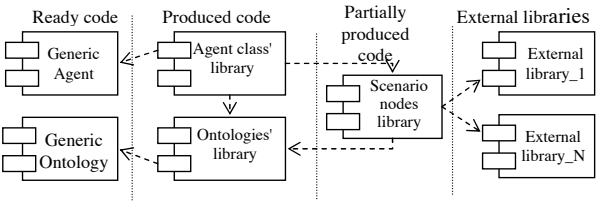

Fig. 9. Class libraries' scheme

The core of the detailed design is in depth development of the agent classes and service models. Let us outline how these activities are supported by MASDK. Agent classes inherit behaviours of the roles assigned. In particular, they inherit the roles' schemes and behaviour scenarios that require detailed specification of the agent class model (scenario nodes and transitions between them) in terms of variables and attributes.

This task of detailed design is aimed at more precise specification of agent class behaviour in terms of the agent class model variables, attributes, behaviour scenarios and functionalities of the scenario nodes. Variable and attribute types are defined either according to the ontology concepts used, or by standard data types usually supported by programming languages.

The resulting specification is used as an input to the SCB (Fig. 1), which checks MAS model specification correctness and either reports (through messaging) on the developed model incompleteness or inconsistency, or automatically builds source code of agent classes. In the former case, the messages inform the designer about what has to be redesigned in the model specification, e.g. the messages can be as follows: *"Description of behaviour scenario ⟨name⟩ is not completed"*, or *"Proactive behaviour of liveness expression ⟨name⟩ is not described"*. Model completeness and consistency can also be checked during the design process, i.e. independently of generation of the source code. This helps the designer to earlier fix the current state of MAS model development.

The architecture of the agent class software produced by SCB is depicted in Fig. 9. *Generic agent* and *Generic ontology* are provided by MASDK 4.0 environment as ready (reusable) components in terms of source code from the very beginning. They represent invariant behaviour of agent classes and abstract ontology. *Generic agent*, in particular, includes classes representing the abstract behaviour scheme of agent classes, the abstract proactive behaviour model of agent classes, abstract simple scenarios, etc.

The source code of *Agent class* and *Ontology* components is generated by the SCB component. For this purpose, the SCB translates elements of the model specification of each agent class to the corresponding class libraries. In particular, simple behaviour scenarios of an agent class and scenario performance order are represented by the *Agent class library.* The *Ontology library* provides for access to data and knowledge storage of *Agent class*. The source code of these two libraries is automatically generated in full without any additional programmer efforts.

The *Scenario nodes library* specifies scenario node functionalities, whose classes and methods correspond to simple scenarios of agent class and their nodes, respectively. Specification of scenario nodes includes identification of their names, and informal description (comment) and strict specification of their input/output attributes. Therefore,

source code generated by SCB component is composed of the headers, the variables and the attributes of the classes and their methods, but the "bodies" of methods must be encoded by programmers manually. The agent class can use some external entities (e.g., applications, resources, etc). If they are not agent-based software, these components are developed externally, outside the MASDK 4.0 environment.

Development of MAS can be of iterative nature. In this case, the main problem is the maintenance of the source code developed earlier. This problem is solved by the SCB, which reports which classes and methods 1) are new, 2) which of them can be reused since they were developed earlier and do not require any modification, 3) which of them have to be rewritten according to MAS model modification, and 4) which of them should be deleted. It is worth noting that the source code maintenance problem concerns only the *Scenario nodes* component. The components *Agent class* and *Ontology* are generated by the SCB anew using the new MAS model specification.

6 Related Work

Since the first publication [20], Gaia was accepted as a valuable abstract methodology focusing on organizational issues. Many publications were devoted to its extensions and in-depth development to make it practically applicable.

Cernuzzi et al [3] proposed enriching Gaia with AUML for protocol specification. Indeed, the interaction model is a key issue of the organizational model, and adding the formal notation of AUML to Gaia can significantly enrich the methodology. The Agent Interaction Protocol (AIP) of AUML regards the protocol as an integrated entity combining roles, constraints, and communication acts, while clearly expressing, in UML-like notation, the formal protocol semantics. The paper pays specific attention to the problem of modeling the complexity of open MAS and emergent behaviours. It introduces a distinguished set of agent class instances allocated the same roles while Gaia just specifies the role. An additional argument for AUML is that it is capable of specifying timely message ordering that is important to implement concurrency.

Garcia-Ojeda, et al [10] propose a similar extension of Gaia, using benefits provided by AIP of AUML. In fact, UML is well developed and widely used as a de-facto software design standard and, therefore, would be accepted by practitioners. The authors proposed to refine the architectural design interaction model based on the two first layers of AIP in terms of AUML for more detailed interaction model representation. Role and service models (at architectural and detailed design) are refined via integration of the AIP third layer and extended UML Class Diagrams. Finally, they refine the organizational model by integrating all the Gaia models developed at previous stages using the Alaadin model proposed by Ferber et al [7], that naturally provides a basis for developing representational mechanisms for organizational concepts.

In order to make Gaia more practically applicable, Gonzalez-Palacios, et al [13] proposed an extension of Gaia in two aspects. First, they consider the design of the internal composition of agents that is omitted in Gaia. This activity uses, as the input, the organizational design results. The proposed agent model is composed of two parts, the *structure model* and the *functionality* model. The first model decomposes roles into classes, while the functionality model is intended to specify collaboration of the above

classes determining the expected role classes behaviour. An advantage of this approach is that such design activity is independent of the specific agent architecture (reactive, BDI, etc.). The second extension regards an iterative approach to large scale application development. The whole development process is divided into simply manageable units that are analyzed, designed and implemented one after another, thus extending previously produced executable deliverables. The final deliverables must contain all the functionality expected from the system.

The Roadmap methodology described in [14] is motivated by the desire to extend Gaia to engineering of large scale open systems, viewing the latter as computational organizations. Roadmap introduces use cases in order to discover requirements (like MASE [6]), make explicit agent environment and knowledge models, and also to enrich Gaia by interaction AUML-based models. However, the authors do not regard these refinements in the needed detail, while other issues remain unclear.

The main drawback of the reviewed and other existing works is that all of them deal mostly with particular stages or aspects of Gaia, extending its particular modeling issues. In contrast, in this paper we treat the whole lifecycle of Gaia, while proposing a formal specification language supported by a powerful software tool, MASDK 4.0, implementing a model- driven agent-based software engineering approach.

7 Conclusion

The MAS development environment, MASDK 4.0, thoroughly implements the proposed extension of the Gaia methodology. It supports user-friendly technology for MAS application development. Its advantages and novelties are as follows:

1. It is based on an extended version of the well founded *Gaia methodology*. The proposed *extensions* are intended for making Gaia applicable to practical use.
2. It realizes a *model-driven engineering approach*, providing automatic support for consistency and integrity of all the intermediate and final solutions produced by developers at all Gaia development stages. The use of model-driven engineering allows for substantial speeding up of the development process. This approach is supported by the *formal specification language, ASML,* that is provided with graphical notation, thus supporting the user-friendly graphical design style.
3. MASDK 4.0 *supports* the development activity *at all development stages*: at the *analysis stage* when organizational issues of the MAS macro-model are decided; at the architectural and detailed *design stages* where the development process is represented in graphical notation of the formal specification language; and at semi-automatic *code generation and deployment stages.*
4. MASDK 4.0 provides *automatic generation of agent behaviour scenarios* schemes using formal protocol specifications as input. This is one of the most important advantages of MASDK4.0, distinguishing it from other existing MAS development environments and tools.
5. MASDK 4.0 is integrated with a *FIPA-compatible distributed P2P agent platform*, supporting a service-oriented approach. It can provide a distributed P2P infrastructure for interaction of heterogeneous software agents running over various operating systems and using heterogeneous communication protocols.

The MASDK environment was tested on various applications during recent years. Its current runtime version and documentation are available on the web [16]. In addition, the FIPA compliant agent platform (runtime version) is freely available. This version, together with the documentation, can be found at [15].

Future work is aimed at: thorough testing of the current version on various applications, including embedded and mobile ones in order to determine the directions of Gaia and MASDK's further enrichment to make it of industrial level. The intended enrichment of the FIPA compliant P2P agent platform should also be done in order to provide it with more system services and capabilities supporting operation of heterogeneous nomadic agents. Development of the light versions of the MASDK environment specifically intended for mobile applications is also planned.

References

1. AgentLink. Agent Software, http://eprints.agentlink.org/view/type/software.html
2. Bellifemine, F., Poggi, A., Rimassa, G.: JADE — A FIPA-compliant agent framework, http://sharon.cselt.it/projects/jade/papers/PAAM.pdf
3. Cernuzzi, L., Zambonelli, F.: Experiencing AUML in the Gaia Methodology. In: 6th International Conference on Enterprise Information Systems — ICEIS 2004, Porto, Portugal (2004)
4. Coguaar web site, http://www.cougaar.org
5. DeLoach, S., Wood, M.: Developing Multiagent Systems with agentTool. In: Castelfranchi, C., Lespérance, Y. (eds.) ATAL 2000. LNCS, vol. 1986, p. 46. Springer, Heidelberg (2001)
6. DeLoach, S., Wood, M., Sparkman, C.H.: Multiagent systems engineering. International Journal of Software Engineering and Knowledge Engineering 11(3), 231–258 (2001)
7. Ferber, J., Gutknecht, O.: A Meta-Model for the Analysis and Design of Organizations in Multiagent Systems. In: Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS 1998), pp. 128–135. IEEE Computer Society, Los Alamitos (1998)
8. FIPA P2P NA WG6, Functional Architecture Specification Draft 0.12, http://www.fipa.org/subgroups/P2PNA-WG-docs/P2PNA-Spec-Draft0.12.doc
9. FIPA ACL Message Structure Specification, http://www.fipa.org/specs/fipa00061/SC00061G.htm
10. Garcia-Ojeda, J., Arenas, A., Perez-Alcazar, J.: Paving the way for implementing multiagent systems. In: Müller, J.P., Zambonelli, F. (eds.) AOSE 2005. LNCS, vol. 3950, pp. 179–189. Springer, Heidelberg (2006)
11. Giunchiglia, F., Mylopoulos, J., Perini, A.: The Tropos software development methodology: Processes, Models and Diagrams. In: Giunchiglia, F., Odell, J.J., Weiss, G. (eds.) AOSE 2002. LNCS, vol. 2585, pp. 162–173. Springer, Heidelberg (2003)
12. Gomez, J., Fuentes, R., Pavon, J.: The INGENIAS Methodology and Tools. In: Agent-oriented Methodologies, pp. 236–275. Idea Publishing Group, USA (2005)
13. Gonzalez-Palacios, J., Luck, M.: Extending Gaia with Agent Design and Iterative Development. In: Luck, M., Padgham, L. (eds.) Agent-Oriented Software Engineering VIII. LNCS, vol. 4951, pp. 16–30. Springer, Heidelberg (2008)
14. Juan, T., Pearce, A., Sterling, L.: ROADMAP: Extending the Gaia Methodology for Complex Open Systems. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002), pp. 3–10. ACM, New York (2002)
15. LIS Agent Platform, http://space.iias.spb.su/ap/

16. MASDK 4.0, http://space.iias.spb.su/masdk
17. Reticular Systems Inc.: AgentBuilder An Integrated Toolkit for Constructing Intelligent Software Agents. Revision 1.3 (1999), http://www.agentbuilder.com/
18. Rimassa, G., Kernland, M., Ghizzioli, R.: LS/ABPM — An Agent-powered Suite for Goal-oriented Autonomic BPM. In: Proceediongs of AAMAS 2008, Portugal (2008)
19. Unified Modeling Language: Superstructure, http://www.omg.org/docs/formal/07-02-05.pdf
20. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing Multiagent Systems: The Gaia Methodology. Transactions on Software Engineering and Methodology 2(3), 317–370 (2003)