

From Requirement Specification to Prototype Execution: a Combination of a Multiview Use-Case Driven Method and Agent-Oriented Techniques

E. Astesiano M. Martelli V. Mascardi G. Reggio

Email: {astes, martelli, mascardi, reggio}@disi.unige.it

Abstract

In this paper we discuss how to combine a multiview use-case driven method for the requirement specification of a system with an agent-oriented method for developing a working prototype. The rationale behind this combination is to cover the complete software development cycle, while the two methods it originates from only cover a part of it. The prototype execution allows to obtain useful feedbacks on the coherence of the UML artifacts produced during the requirement specification phase.

Keywords: Automated Software Specification, Requirement Specification, UML, Use-Case Driven Methods, Validation and Verification.

1 Introduction

Research on methods for modelling and programming multi-agent systems, where autonomous, distributed and heterogeneous components interact following sophisticated protocols, spans across the fields of artificial intelligence and software engineering. The metaphor underlying that research is that any component of the application is an intelligent agent, namely, according to [8], “... a computer system, situated in some environment, that is capable of flexible autonomous actions in order to meet its design objectives.”

The correct and efficient engineering of real-world applications conceptualized as multi-agent systems (MASs) is the main goal of agent oriented software engineering (AOSE), a very active research area seeking abstractions, languages, methodologies and tools for MAS modelling, verification, validation and implementation.

Agent-oriented methodologies can be broadly divided into two groups [6]: those that take their inspiration from object-oriented (OO) development and those that adopt knowledge engineering or other techniques.

To take advantage of the potential of both approaches, a third, orthogonal category of AOSE methods is recently

emerging, based on the idea of composing method features to create new AOSE methodologies [9, 10]. The benefit of the modular approach is obvious: instead of creating incompatible techniques, models and CASE tools for each method, modular and reusable solutions can be created once, and shared within different methods.

In this paper we show how to apply the modular AOSE approach to two existing software engineering methods. The first method, presented in sect. 2, concerns the requirement specification, is UML-based, proposes a stringent way of structuring and representing the Requirement Specification artifacts, and presents a number of novelties w.r.t. the best-known current methods. The second method (sect. 3) concerns with architectural design, detailed design and prototype implementation of MASs. The composed method covers the software development cycle from early requirement specification to prototype implementation and allows to check the coherence of the UML artifacts produced as an output of the requirement analysis stage thanks to the execution of the working prototype. Checking the coherence of UML diagrams is a well known and still open problem. By composing the two methods we obtain a tool for facing this issue almost for free. A case study, first used in sect. 2 to introduce the method for requirement specification, practically exemplifies in sect. 4 how this is possible. Sect. 5 concludes.

2 A multi-view, use-case driven and UML-based method for requirement specification

The work presented in [2] (see [3] for an updated and extended version) refines and extends existing proposals of use-case driven methods for requirement specification combined with object-oriented techniques, particularly in connection with visual notations such as UML [12].

The proposed method, rigorously multiview, use-case driven and UML-based, presents some novelties, somewhat departing from traditional object-oriented dogmas and incorporating some good, but perhaps lost, ideas found in

well-known methods, such as Structured Analysis [17], and in the work of some pioneers, such as M. Jackson's [7]. Indeed, central to this approach are the following three concepts:

- the total separation of the Domain Model from the System, a distinction somewhat blurred in many object-oriented approaches;
- the distinction between the System and the environment, formalized in a Context View, that will be the basis for the definition of the requirements about the interaction between the System and the context, in conjunction with the use case diagram;
- the use of a very Abstract State, instead of the many optional use case states, to allow expressing abstract requirements about the interaction of the System and the context, without providing an object-oriented structuring at a stage when such a structure is not required and can be premature; such System concept would disappear in the following Design stages.

It is worthwhile to note that the proposal relies on the use of a well-chosen subset of UML that can be given a rigorous semantic foundation, see, e.g., [16, 15].

The Requirement Specification structure and activity proposed by the method assume that the Problem Domain Modelling produces as an artifact a UML package, thus describing in an object-oriented fashion the part of the real world that concerns the System, but without any reference to the System itself nor to the problem to which the System provides a solution. The Requirement Specification artifacts consist of different views of the System, plus a part, Dictionary, needed to give a rigorous description of such views. Its structure is shown in fig. 1 by a UML class diagram. Some of the above views (e.g., Internal View and Context View) are new w.r.t. the current methods for the OO UML-based specification of requirements. They play a fundamental role to help ensure the consistency among the various use cases and of the whole specification.

We present directly the resulting specification of the requirement on a simple running system (shortly System in the following), whose natural language description is given in fig. 2; refer to [2] for seeing how it has been produced.

Dictionary The Dictionary lists and makes precise all the entities appearing in the various views of the System to help guarantee the consistency of the concepts used in such views. We do not report here the Dictionary of the running example for lack of room; it can be found in [2].

Use Case Diagram The Use Case View, as it is now standard, shows the main ways to use the System (*use*

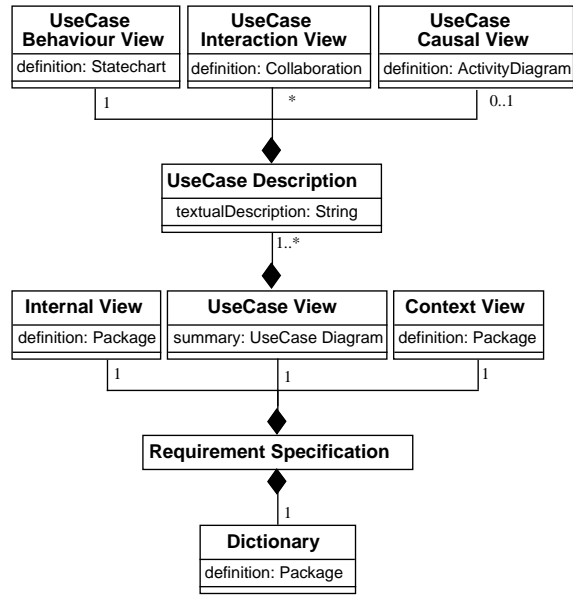


Figure 1. Requirement Specification artifacts

The system involves an “algebraic lottery” where tickets are numbered by integer numbers, winners are determined by means of an order over such numbers, and a player buys a ticket by selecting its number. Whenever a player buys a ticket, she/he gets the right to have another free ticket, which will be given at some future time, fully depending on the lottery manager decision. The number of a free ticket is generated by the set of the numbers of the already assigned tickets following some law. Thus a lottery is characterized by an order over the integers determining the winners and a law for generating the numbers of the free tickets. The tickets must be bought and paid on-line using credit cards with the help of an external service handling them. Possible clients must register with the lottery system to play, becoming players; and players access the system in a session-like way. An external service takes care of the registration of the players and of the distribution of the session keys.

Figure 2. Specification of the running example

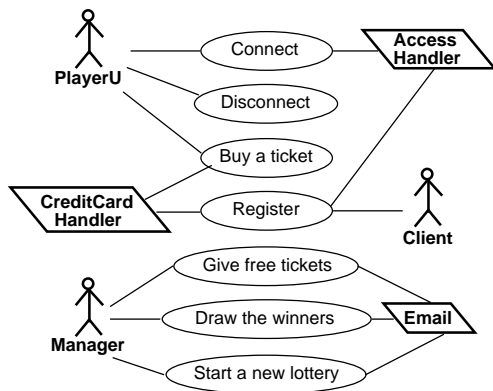




Figure 3. Use Case Diagram for System

cases), making clear which actors take parts in them. Such actors are just *roles* (*generic instances*) for some context entities depicted in the Context View.

The Use Case Diagram for System is shown in fig. 3. It depicts that the lottery system will use external services (visually represented by ) for handling the credit cards and the accesses to the system. Furthermore, it will use the email to communicate with the players. The System will provide services to the users (visually represented by ) PlayerU, Client and Manager.

Context View The Context View describes the context of the System, that is which entities (*context entities*) and of which kinds may interact with the System, and in which way they can do that. Such entities are further classified into those taking advantage of the System (*service users*), and into those cooperating to accomplish the System aims (*service providers*).

The explicit splitting between the System and the context entities should help avoid confusions between what exists and needs just to be precisely described (context entities) and what instead has to be developed (System).

The Context View is a UML package importing the Dictionary containing at least a class diagram, where all classes are all of the following three stereotypes: `<<System>>` that stands for the System (exactly one class in the diagram with this stereotype), `<<SU>>` (System user) or `<<SP>>` (System provider). In such class diagram there is one binary association from the System class into each other class, and these are all the associations.

Because the actors in the Use Case Diagram are just generic instances of `<<SU>>` and `<<SP>>` classes appearing in this view, they should be in accord.

The operations of the `<<SU>>`-`<<SP>>` classes and of the `<<System>>` class model their mutual interfaces, that is in which way they may interact.

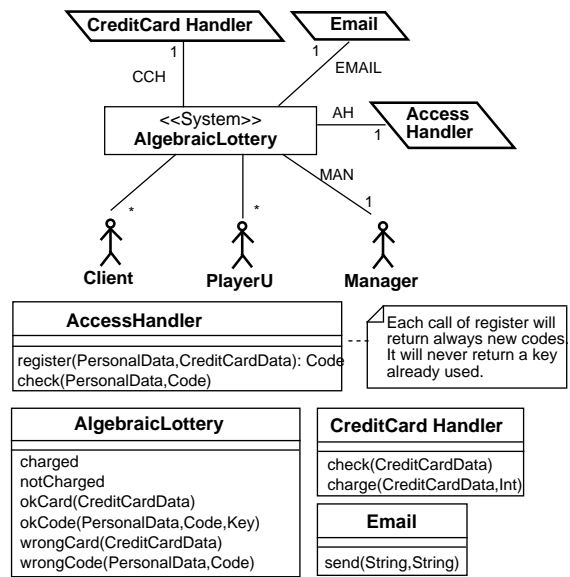


Figure 4. Context View

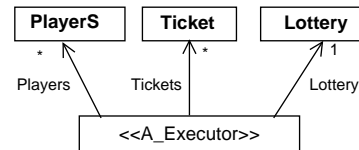


Figure 5. Internal View

Notice that the Context View presents also what we know on the context entities (assumptions on their behaviours, on how they interact, ...). It is important to do this task before describing the use cases, since the context entities are not under the responsibility of the System developer, but they are already existing.

Usually we factorize the presentation of the Context View in a diagram showing only the class names and the associations and in another one showing the operations of the various classes, and, if any, the property on the behaviour of the context entity classes (fig. 4).

Internal View The Internal View describes abstractly the internal structure of the System, that is essentially its Abstract State. It will help precisely describe the behaviour of the use cases, by allowing to express how they read and update it. UML allows a single use case to have a proper state, but in this method there is a unique state for all the use cases, to help model their mutual relationships (e.g., if two use cases update the same information, it is possible to detect and to handle possible conflicts).

The Internal View (fig. 5) of the running example is represented by a class diagram importing Dictionary that describes at an extremely abstract level the structure (archi-

ture) of the System. This structure consists of a unique active object able to perform the System activities (abstract executor) and by many passive objects abstractly describing the System Abstract State (Ticket, Lottery and PlayerS, note that these classes are defined in the Dictionary).

From an agent-oriented point of view, the abstract executor is the only entity in the System that fully deserves the name of “agent”. The other passive entities will be “agentified” to make the prototype execution possible.

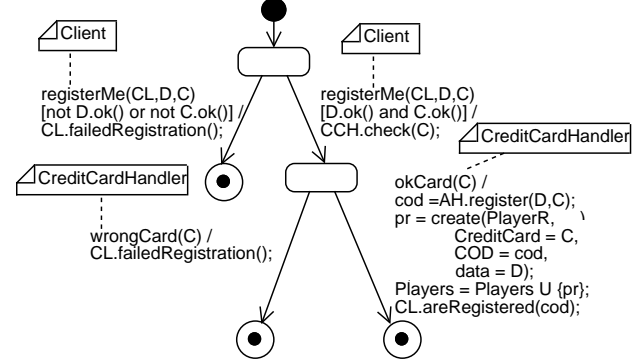
Use Case Description Each use case appearing in the use case diagram will be then described by a Use Case Description that consists of a *textual presentation* of the use case, and of one or more views, which may be of the three kinds: *behaviour* (describing the complete behaviour of the System with respect to such use case), *interaction* (representing the interactions happening in a scenario of the use case) and *causal* (describing all the relevant facts happening during the use case and their causal relationships).

In the following we will describe the Use Case Behaviour View and Use Case Interaction View for the Register use case (see [2] for the others). This use case is textually described by: “A client may register by giving her/his personal data and those of a credit card. If her/his data are correct and those about the credit card are accepted by its handler, then she/he will receive a code, determined by the access handler, and will be considered registered; otherwise she/he will be informed that her/his registration has been refused.”. The Use Case Behaviour View and Use Case Interaction View diagrams are shown in fig. 6.

The Use Case Behaviour View is mandatory for each use case and is defined by a statechart for the <<System>> class such that 1) the conditions on its transitions may test only the System Abstract State given in the Internal View; 2) the actions appearing on its transitions may include only calls of the operations of the context entity classes, as defined in the Context View, and may update the System Abstract State; 3) the events on its transitions may be only call events of the operations of the <<System>> class, as presented in the Context View. To support the definition of the D-CaseLP protocol diagram starting from the behaviour view, we annotate, using the UML note facility (the box with the small ear), the event calls with the “role” making such calls.

A Use Case Interaction View is defined by a sequence (or collaboration) diagram representing the interactions happening in a scenario of the use case among the context entities, the System and the internal abstract constituents of the System, as presented in the Internal View (the abstract executor and the passive components). In the visual representation we split the diagram in two swimlanes, by a dashed line, to show what is happening inside and out-

Use Case Behaviour View:



Use Case Interaction View:

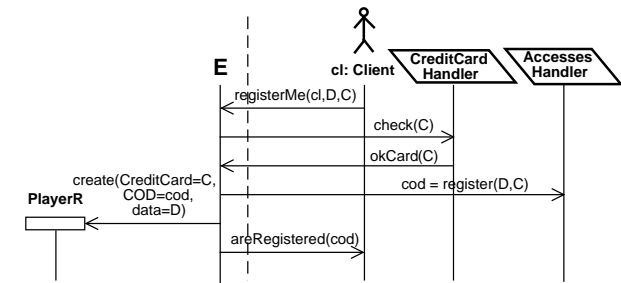


Figure 6. Behaviour and Interaction Views

side the System, and the lifeline corresponding to the abstract executor will be marked by a big **E**. There are no restrictions on the number of the Use Case Interaction View present in this description, but they must be coherent with the behaviour view of the same use case (that is, they must represent particular executions of the complete behaviour described by such view). This coherence can be verified executing the MAS prototype (see below).

The Use Case Causal View is defined by an activity diagram describing all the relevant facts happening during the use case and their causal relationships. For sake of synthesis we do not show this view for the running example.

3 The D-CaseLP AOSE tool

D-CaseLP (Distributed CaseLP, [1]) is a MAS rapid prototyping environment designed and developed by the Logic Programming Group of the CS Dept. of Genova University. It supports the MAS designer from the architectural and detailed design to the development of a MAS prototype.

D-CaseLP agents are characterized by their architecture, roles and ontologies. The agent’s architecture is determined by both the agent internal data structures and the control flow among them, called the engine of the architecture. Roles (for example seller, buyer, client, manager, ...) are characterized by the set of services required and

provided and by the communication protocols for exchanging complementary services. Ontologies are structures that hold information about entities, their properties, and relationships among them that are possible in a specific domain of knowledge. D-CaseLP ontologies are similar in spirit to the Dictionary structure introduced in sect. 2.

From a computational point of view, agents modelled in D-CaseLP are characterized by a state, a program and an engine. The state includes data that may change during the execution, the program contains information that does not change during the execution of the agent, and the engine controls the execution of the agent. The architecture of an agent contains components for its state and its program, and an engine operating on them. Agents which share the same roles, architecture and ontologies can be grouped into agent classes: an agent class is determined by the engine and the program. Agent instances belonging to the same class only differ in their state.

In order to develop and execute a working prototype with D-CaseLP, three steps must be followed: architectural design, implementation, and execution/debugging.

Architectural design This stage consists in determining the roles necessary for the application, establishing the complete role model (i.e., the interaction protocols which may take place among roles), grouping the roles into agent classes, assigning the most suitable architecture to each class and finally determining the needed agent instances. The languages provided for facing the architectural design stage include an extension of UML and its XML-based textual counterpart, D-CaseLP-XML. The D-CaseLP role model is specified by means of D-CaseLP *protocol diagrams* based on AUML [13, 14]. With respect to AUML, the D-CaseLP protocol diagrams have some restrictions to avoid ambiguities and to allow a complete automation of the translation from the protocol diagrams into the JESS rule-based language used to implement agents. In D-CaseLP, actors of protocols are roles. The association between a role, the agent classes playing that role and their architecture is defined in an *architecture diagram* while the association between an agent instance and its agent class is defined in an *agent diagram*. Both diagrams are similar to the ones proposed in [5]. Diagrams which only include standard UML constructs can be graphically modelled with existing UML-based modelling tools, otherwise D-CaseLP-XML may be adopted.

At the time of writing only the JESS rule-based architecture has been integrated into D-CaseLP, thus there is only one choice for the architecture to assign to each agent class. This limitation turns out to be an advantage for developers who want to use D-CaseLP simply as a tool for animating UML diagrams and checking their consistence, and not as a MAS prototyping environment. These devel-

opers do not need to face the choice of “the right agent architecture to do the right thing” [11]. They do not even need to know what exactly an agent architecture is: according to the developer’s needs and skills, the agent concepts and technology which characterize D-CaseLP may be kept in the background.

Implementation The implementation of the working prototype exploits the xslt technology and the ability to export UML diagrams into the XML Metadata Interchange Format XML to automatically generate rules which respect the given UML diagrams. The rule-based language we chose for the implementation of D-CaseLP agents is JESS (<http://herzberg.ca.sandia.gov/jess/>). More in detail, if the UML-based modelling tool allows to export UML diagrams into XML, the XML file can be automatically translated into the proprietary intermediate format, D-CaseLP-XML. The diagrams which cannot be modelled using existing tools must be defined directly in D-CaseLP-XML. D-CaseLP-XML specifications can be automatically translated into rule-based JESS code. Some portions of the generated code must be completed by the developer in order to make it executable.

Execution Once completed by the developer, the JESS code can be integrated into the FIPA-compliant JADE platform [4], and the resulting JADE prototype can be executed. Running the prototype allows to follow the interactions that take place between the agents in the MAS thanks to a set of monitoring facilities provided by JADE. If the agents are programmed for interacting in different ways according to the content of their state, the MAS developer can manually modify the files defining the agents’ initial state and run the prototype starting from many different initial situations. This will result in different sequences of messages exchanged by the agents that the MAS developer will observe to get feedbacks on the correctness of the agents definition.

To summarize, the concatenation of modelling, implementation and execution stages results in the process depicted in fig. 7. At the left side of the picture, activities that the developer must face “by hand” are represented. At the right side, activities fully faced by software applications are represented.

4 Combining the approaches: a case study

The SE approaches described in sect. 2 and sect. 3 may be concatenated in order to cover the SE process from the requirement capture and specification to the development of a working prototype and to get feedbacks on the require-

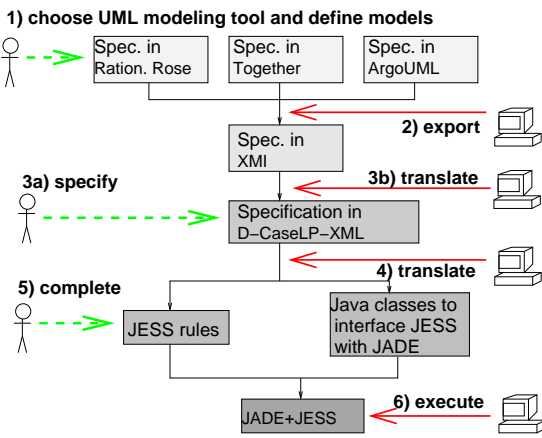


Figure 7. Development steps in D-CaseLP

ment specification correctness thanks to the prototype execution. As clearly stated in [2], the requirement specification discussed in sect. 2 is preliminary to design and does not commit to any system model. This makes it possible to use the output of the requirement capture for specifying an application modelled as a MAS, and to use D-CaseLP for developing a working prototype of the given specification.

In this section we use the simple running example taken from [2], whose requirement specification has been given in sect. 2, to practically show how the two approaches can be combined. The application we will model is intentionally simple and not all the entities involved in it show the features of intelligent agents as defined in [8]; for this application D-CaseLP is mainly exploited as a tool for UML protocol diagram animation rather than as an agent-based environment.

4.1 MAS modelling with D-CaseLP

From the artifacts composing the requirement specification, see sect. 2, we have most of the information we need to model the MAS: we just need to complete the given specification by providing the D-CaseLP protocol, architecture and agent diagrams.

As far as the protocol diagrams are concerned, their definition can be easily derived from the use cases defined during the requirement specification stage. We need to slightly modify the entities involved in the use case diagrams for use in D-CaseLP because, since all the entities in the system are now treated as agents, they are no more passive service providers and they must always use asynchronous message exchange instead of method calls. The main sources of information for developing the protocol diagrams are the textual description and the behaviour view.

Starting from the Register use case discussed in the previous section we developed the D-CaseLP protocol dia-

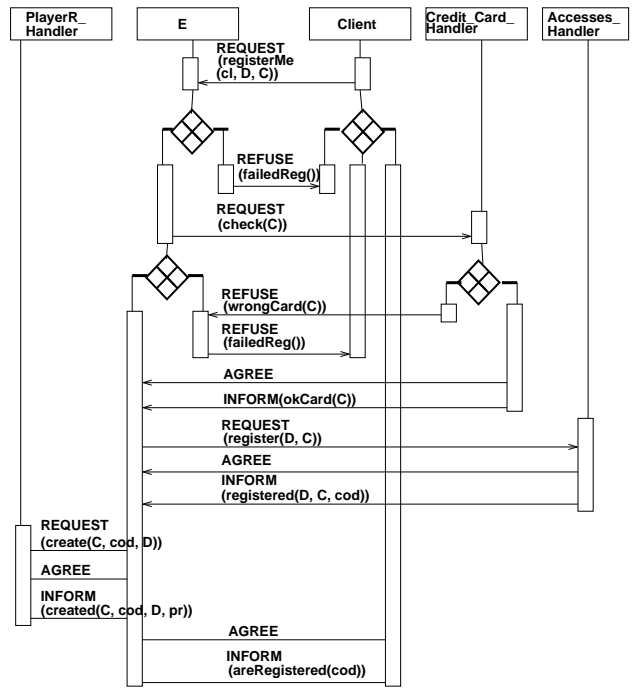


Figure 8. D-CaseLP protocol diagram for the Register use case

gram shown in fig. 8. Similar protocol diagrams are defined for each use case appearing in the requirement specification (precisely in the use case view). The way these diagrams can be composed to obtain a unique D-CaseLP running prototype is not yet fully defined; for the moment, we limit ourselves to run the parts of the prototype related to each use case separately. We are working to extend D-CaseLP method and support tools to be able to combine all the protocol diagrams and to run the resulting complete prototype. In the D-CaseLP protocol diagram the method calls appearing in the Behaviour View are transformed into FIPA messages. As an example, the registerMe(cl, D, C) request becomes a REQUEST(registerMe(cl, D, C)) message (first message exchanged starting from the top of the protocol diagram). If the not C.ok() or not D.ok() condition stated in the Behaviour View is satisfied, the abstract executor sends a REFUSE message to the client, specifying that the registration failed. Otherwise it asks to the Credit Card Handler to check the given card code, and, according to the answer provided (wrongCard(C) or, respectively, okCard(C)), it sends a REFUSE message to the client or, respectively, goes on with the register protocol by contacting the Accesses Handler and the PlayerR Handler. Note that the conditions for sending messages stated in the Behaviour View are not explicitly present in the D-CaseLP protocol diagram, thus the JESS code generated from the

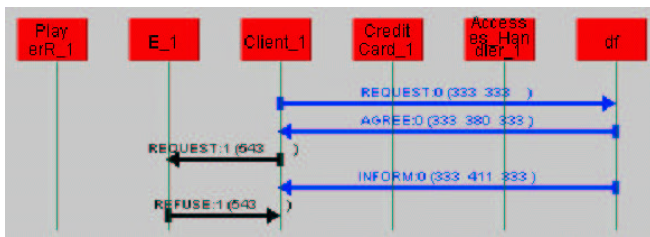


Figure 9. Personal and CC data are not ok

D-CaseLP protocol diagram *will not include* these conditions. It is up to the prototype developer to manually complete the JESS code in a coherent way with the rest of the specification. This task is not difficult to face because the Use Case Behaviour View clearly defines these conditions.

As far as class and agent diagrams are concerned, we assume that there is a one-to-one relation between roles and classes, identified by role name plus “_Class”, and that there is only one agent instance for each class: `PlayerR_1` of class `PlayerR_Handler_Class`, `E_1` of class `E_Class`, and so on.

4.2 Development and execution of the prototype

From the D-CaseLP protocol, architecture and agent diagrams we can automatically generate the JESS code for the given agent classes. As already observed, the code for the agent classes program must be completed by adding the conditions under which a message can be sent. These conditions are explicitly stated in the Use Case Behaviour View (fig. 6), thus the developer can easily add them to the JESS code. Once the code is completed and the initial state of the agent instances has been defined, the Java classes for interfacing JESS and JADE can be automatically created and the resulting JADE prototype can be executed.

The agent’s initial state determines the protocol diagram branch that will be followed in a simulation run. As an example, let us suppose that the client agent `Client_1` sends a `registerMe` request with credit card data 23300 and personal data `viviana_mascardi` to `E_1`. If `E_1` initial state does not include the information that both data are ok, the client request cannot be accepted, leading to the situation shown in fig. 9. This figure shows the output of the JADE Sniffer agent. Besides the agents from the algebraic lottery application, there is a directory facilitator agent (`df`, the last one on the right) which is automatically provided by JADE to offer a yellow pages service to the user-defined agents.

If `E_1` state includes the information (ok 23300) and (ok `viviana_mascardi`) (i.e., both data sent by `Client_1` are ok), `E_1` will send a request to `CrediCard_1` to know if the given credit card can be accepted or not. If `CrediCard_1`

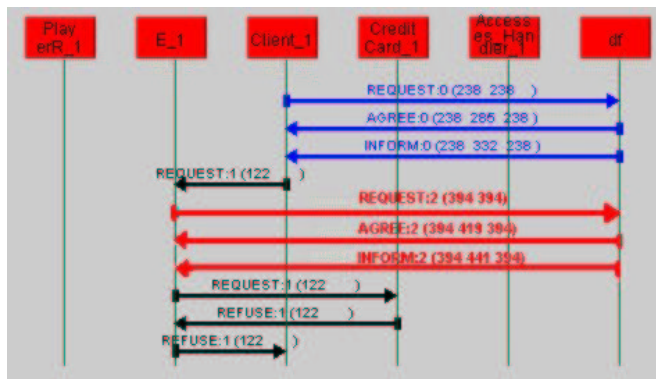


Figure 10. Credit card number is wrong

initial state contains an atom stating the 23300 is a wrong credit card number, the situation illustrated in fig. 10 will take place, otherwise the registration will end up with a success (not shown).

Thanks to the prototype execution, it is possible to check if the interaction views for the use case are coherent with the other views. By running the prototype a sufficient number of times starting from as many different agents’ initial states as possible, all the possible situations should be observed. If the software engineer who captured the requirements of the system forgot to describe some interaction views or described them incorrectly, the prototype execution may help her/him in completing (resp. correcting) the missing (resp. incorrect) interaction views, ensuring their coherence with the other views. For example, let us suppose that the only interaction view described by the requirement analyst was the one described by the Use Case Interaction View of fig. 6: the execution run leading to the message exchange shown in fig. 9 and fig. 10 helps the analyst in understanding that there are more scenarios she/he should take into account.

5 Conclusions

The compositional approach to the development of SE methodologies in general, and AOSE ones in particular, is a very recent approach to engineer MAS applications. It is based on the idea of an AOSE feature, namely an entity which encapsulates software engineering techniques, models, CASE tools and development knowledge. Every SE/AOSE method provides a set of features which can be isolated and dealt with like atomic bricks to pick up and compose in a new modular method.

We used the Use Case Behaviour View provided by the method discussed in sect. 2 to produce the D-CaseLP protocol diagram and create a “trait-d’union” between the methods (fig. 11). The method resulting from this concatenation process is definitely more powerful than the

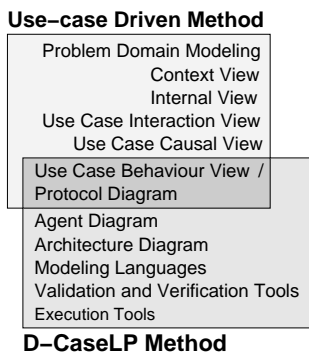


Figure 11. AOSE features

two methods it originates from. It supports a very well structured requirement capture stage which abstracts from any specific architecture in modelling the System and the roles involved in the application. It provides a set of languages and guidelines for “agentifying” the application by grouping roles into agent classes, establishing the interaction patterns among roles, assigning an agent architecture to each agent class and defining the agent instances of each agent class. It allows to execute the resulting agentified specification; the burden on the prototype developed is very light because, thanks to the detailed requirement specification, all the conditions that she/he has to add to the generated JESS code can be easily found in the use case descriptions. Finally, it allows to check that the artifacts produced during the requirement specification stage are all the ones necessary for describing the system requirements, and moreover that they are correct. This “coherence check” is done by comparing the results of the execution runs with the interaction views. Since the prototype is based on protocol diagrams built from the behaviour views, the coherence between the prototype results and the interaction views implies a coherence between the behaviour view and the interaction view.

Acknowledgements

The authors thank Stefano Miglia for his collaboration in developing the D-CaseLP “Algebraic Lottery” prototype.

References

- [1] R. Albertoni, M. Martelli, V. Mascardi, and S. Miglia. Specifica, implementazione ed esecuzione di un prototipo di sistema multi-agente in D-CaseLP. In F. De Paoli, S. Manzoni, and A. Poggi, editors, *Proc. of WOA*, Milano, Italy, 2002.
- [2] E. Astesiano and G. Reggio. Knowledge Structuring and Representation in Requirement Specification. In *Proc. of SEKE 2002*. ACM Press, 2002.
- [3] E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications: Complete Version. Technical Report DISI-TR-03-06, DISI, Università di Genova, Italy, 2003. Available at <ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio03c.pdf>.
- [4] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with JADE. In *Intelligent Agents VII*. Springer Verlag, 2001. LNAI 1986.
- [5] F. Bergenti and A. Poggi. Exploiting UML in the design of multi-agent systems. In *Engineering Societies in the Agents World*. Springer Verlag, 2000. LNCS 1972.
- [6] P. Ciancarini and M. Wooldridge. Agent-oriented software engineering: The state of the art. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering - First International Workshop, AOSE 2000*, Limerick, Ireland, 2000. Springer Verlag.
- [7] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.
- [8] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.
- [9] T. Juan, M. Martelli, V. Mascardi, and L. Sterling. Customizing AOSE methodologies by reusing AOSE features. To appear in: *Proc. of the 2nd International Conference on AAMAS*, 2003.
- [10] T. Juan and L. Sterling M. Winikoff. Assembling agent-oriented software engineering methodologies from features. In *Proc. of the 3rd International Workshop on AOSE, at AAMAS’02*, Bologna, Italy, 2002.
- [11] J. P. Müller. The right agent (architecture) to do the right thing. In *Intelligent Agents V*. Springer-Verlag, 1999.
- [12] Object Modeling Group. *Unified Modelling Language Specification, version 1.3*, 2000.
- [13] J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In *Proc. of the Agent-Oriented Information System Workshop at the 17th National Conference on Artificial Intelligence*, 2000.
- [14] J. Odell, H. V. D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering - First International Workshop, AOSE 2000*, Limerick, Ireland, 2000. Springer Verlag.
- [15] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000*, Berlin, 2000. Springer Verlag. LNCS 1783.
- [16] G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Approach. In H. Hussmann, editor, *Proc. of FASE 2001*, Berlin, 2001. Springer Verlag. LNCS 2029.
- [17] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1989.