Chapter XLV
Caste-Centric Development of Agent-Oriented Information Systems

Lijun Shan
National University of Defense Technology, China

Rui Shen
National Laboratory for Parallel and Distributed Processing, China

Ji Wang
National Laboratory for Parallel and Distributed Processing, China

Hong Zhu
Oxford Brookes University, UK

ABSTRACT

Based on the meta-model of information systems presented in Zhu (2006), this chapter presents a caste-centric agent-oriented methodology for evolutionary and collaborative development of information systems. It consists of a process model called growth model, and a set of agent-oriented languages and software tools that support various development activities in the process. At the requirements analysis phase, a modelling language and environment called CAMLE supports the analysis and design of information systems. The semi-formal models in CAMLE can be automatically transformed into formal specifications in SLABS, which is a formal specification language designed for formal engineering of multi-agent systems. At implementation, agent-oriented information systems are implemented directly in an agent-oriented programming language called SLABSp. The features of agent-oriented information systems in general and our methodology in particular are illustrated by an example throughout the chapter.

INTRODUCTION

In Zhu (2006), we presented a vision of future information systems through an agent-oriented meta-model. The promising features of the meta-model were illustrated in the context of software development on the Internet/Web platforms and the utilisation of mobile computing devices. In this chapter, we address the problem of how to develop such agent-oriented information systems (AOISs). Based on the meta-model introduced in Zhu (2006), we propose a methodology for developing an AOIS which consists of a process model that guides the development activities, along with a set of languages and software tools that support various development activities in the process.

The chapter is organised as follows. We begin by describing an information system used as the running example in the chapter. We then propose an evolutionary development process model for AOIS and outline the caste-centric agent-oriented modelling language and environment, CAMLE. The next section reviews the formal specification language SLABS, which stands for a Specification Language for agent-based systems. The focus then turns to implementation issues, and the SLABSp experimental programming language is briefly described. We conclude the chapter with a discussion of related work and further work.

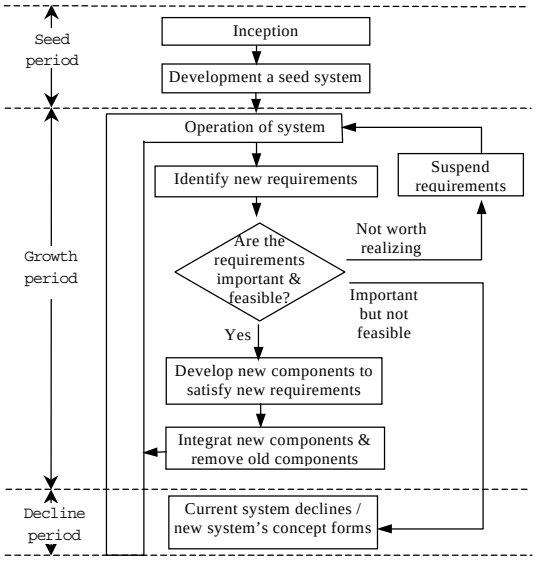
DESCRIPTION OF THE RUNNING EXAMPLE

We will use a simple, but non-trivial, information system to illustrate our methodology as a running example throughout the chapter. The example was proposed and used as a case study by FIPA's AUML Technique Committee (2004) to study agent-oriented modelling methods and notations. It was inspired by the procedure of the United Nations Security Council to pass a resolution. The description of the system follows.

The United Nation Security Council (UNSC) consists of a number of members, some permanent and others elected from UN members. Members become the chair of the Security Council in turn monthly. To pass a UNSC resolution, the following procedure is followed:

1. At least one member of UNSC submits a proposal to the current chair.
2. The chair distributes the proposal to all members of UNSC and sets a date for a vote on the proposal.
3. At a given date that the chair sets, a vote from the members is made.
4. Each member of the Security Council can vote either FOR or AGAINST or SUSTAIN.
5. The proposal becomes a UNSC resolution, if the majority of the members voted FOR and no permanent member voted AGAINST.
6. The members vote one at a time.
7. The chair calls members in a given order to vote, and the chair is always the last one to vote.
8. The vote is open (in other words, when one votes, all the other members know the vote).
9. The proposing member(s) can withdraw the proposal before the vote starts, and in that case no vote on the proposal will take place.
10. All members vote on the same day, one after another, so that the chair does not change within the vote call; but it is possible for the chair to change from one member to another between the time a proposal is submitted until it goes into vote. In this case the earlier chair must forward the proposal to the new one.
11. A vote is always finished in one day and no chair change happens on that day. The date of the vote is set by the chair.

Figure 1. Growth model of software development



In the remainder of the chapter, we will use the above as the initial requirements specification to demonstrate how agent-oriented information systems are analysed, modelled, designed, and implemented in our methodology.

DEVELOPMENT PROCESS

As discussed in Zhu (2006), one of the most attractive potential features of agent-oriented information systems is its strong support of the evolution of information systems and their collaborative developments. To realise this, we proposed a lifecycle model of software systems as shown in Figure 1 (Zhu, Greenwood, Huo, & Zhang, 2000; Zhu, 2002, 2004).

The lifecycle model is called *growth model* because it views information systems' lifecycle as a process of growth. From this point of view, a software system's lifecycle can be divided into three periods: the seed period, the growth period, and the decline period. When an information system is initially constructed and put into operation, it is relatively weak and small in terms of the services it provides, the volume of information it contains, and other non-functional attributes such as performance, security, and so forth. During the operation, the system grows in many directions and dimensions. New components may be integrated into the system to provide new services, while current components may be modified to improve the systems' functional or non-functional properties as users' new requirements are identified and implemented. The system gradually goes to the decline period, and dies when it cannot sustain more modification to meet new requirements. It is worth noting that different types of software systems may be suitable to different lifecycle strategies. For example, software systems of Lenman's S-type (1990, 2001) are more suitable to having a strong seed system and little modifications during the rest of its lifecycle, because such systems' requirements are well understood and well specified. The modifications are mostly corrections of errors in the software systems. However, Zhu (2004) argued that most information systems are Lenman's E-type systems whose requirements are changing, and hence they are evolutionary by nature. They are best developed following a growth strategy with the emphasis on the growth period. In comparison with other strategies that guide information system development, the growth strategy has a number of advantages. The first is the lower risk, because only the best understood requirements are implemented and integrated into the system. The investment in each step of the growth is smaller than implementing a huge system in one big bang. Second, it is more likely to have a shorter time delay from the recognition of a well-understood requirement to the delivery of the functionality. Complicated interactions between requirements can also be reduced and abated. Third, the developers can learn from previous develop-

ment experiences and improve their performance in the follow-up development of new components. They can gain confidence during the development process and see their results earlier than other development strategies. Finally, and most importantly, users' feedback can be obtained much earlier than other strategies, as each step of the growth process takes a much shorter period of time. This enables the users to clarify their requirements easily and guide the direction that the system develops. In fact, this strategy differs from the so-called staged development process model in its emphasis on taking users' feedback to guide the direction of software evolution.

To support the growth strategy, we designed and implemented a set of languages and tools for modelling, specification, and programming agent-oriented information systems. These languages and tools support various activities in the development process.

The modelling language and environment CAMLE supports:

- requirements elicitation and analysis by representing the current information system and the required system in agent-oriented models; and
- feasibility study of the requirements by analysing the required modifications to the existing system.

The formal specification language SLABS and its formal reasoning logic Scenario Calculus support:

- formal description of the requirements of the system under development so that new functionalities and services can be implemented as new components in the form of castes/agents; and
- formal reasoning of the design of the system/new components to ensure that the system will meet the requirements and that the new components can be integrated into the systems as expected.

The agent-oriented programming language SLABSp and its runtime support environment are used for:

- the implementation of the system/components according to the semi-formal specification in the CAMLE model and/or the formal specification in SLABS; and
- the testing of new components and the integration into the existing system.

In the following sections, we will describe each of these languages and tools, and illustrate their uses with the running example described earlier in the chapter.

MODELLING AND ANALYSIS

Modelling plays a crucial role in the development of the seed system and its evolution as the main tool of requirements analysis and system/component design. This section presents the modelling process, and the diagrammatic modelling language and environment of CAMLE (Shan & Zhu, 2003, 2004a, 2004b, 2005; Zhu & Shan, 2005).

Process of Modelling

In our methodology, modelling aims at representing the users' requirements with a set of agents at various granularities and organizing the agents into an information system. The key activities in the modelling and analysis phase include:

- Identify the agents and castes of agents in the system as well as the relationships between them, such as the *is-a* relation (inheritance), membership-shift relation (migration or participation), and whole-

part relation (aggregation, congregation, or composition). The artefact produced in this activity is a caste model for the system from the perspective of system architecture.

- Identify the agents' interaction patterns in various scenarios, and produce a set of collaboration models for the system from the perspective of dynamic behaviour. In order to specify the system in sufficient detail, an agent may be decomposed into a number of components, which are also agents. Then the interaction modelling proceeds to capture the interactions between the components. Eventually, the collaboration model is refined into a hierarchy, where collaboration models at various granularities specify the interactions between component agents at various abstraction levels. Along with agent decomposition, the caste model is enriched with further details to present the caste of agents at various granularities and the structural dependencies between them.
- For each caste, elaborate and specify how its agents perform actions and/or change states in typical scenarios so that a set of behaviour rules can be assigned to the caste. The artefact produced in this activity is a set of behaviour models, each associated to a caste in the system.

The result of the modelling is a system model comprising a set of diagrams that represent the system from various views and at different levels of abstraction. For example, in the UNSC system, a caste diagram is constructed to capture the organization structure, which comprises one *chair* and a number of *UNSC members*—either *permanent member* or *elected member*. Collaboration diagrams describe the typical scenarios of the interaction between UNSC members and the chair. Behaviour diagrams respectively for UNSC member and chair define their specific behaviour rules. More details are given in the next subsection.

During the growth phase of an existing agent-oriented information system, new components for providing new functions, services, and features are developed in the context of the existing system, which will be the operating environment of the new components. Therefore, the model of the existing system is the basis for the representation of the new requirements and the analysis of their feasibility. For example, if the organization of the United Nations Security Council is to be reformed to add a new type of member whose power on resolution is between elected member and permanent member, the UNSC information system can be modified accordingly by adding a new caste representing the new type of members.

The Modelling Language

CAMLE employs the multiple views principle to model complicated systems. There are three

Figure 2. Caste diagram: Notation and the UNSC example

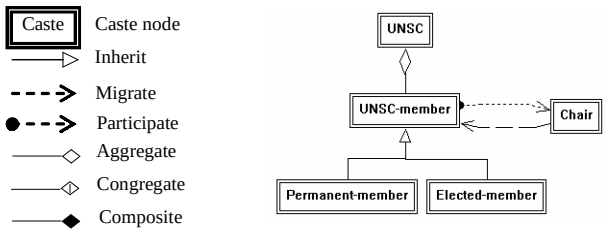
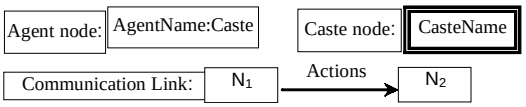


Figure 3. Notation of collaboration diagra



types of models in CAMLE: caste models, collaboration models, and behaviour models. Each model may consist of one or more diagrams.

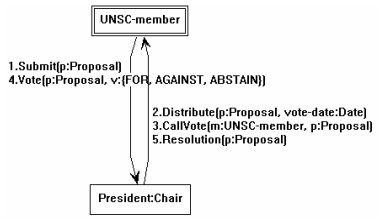
A caste model usually consists of one caste diagram. Figure 2 shows the notation of caste diagrams and an example caste diagram of UNSC system. A caste diagram comprises a set of caste nodes representing various types of agents in the system, and a set of links representing various relationships between agents of the castes.

In the UNSC caste diagram, caste UNSC represents the organization, which is composed of a number of members represented by caste UNSC-member. The aggregate link between the UNSC-member and the UNSC denotes the part-whole relationship between the members and the organization. The rule that members take the role of Chair in rota is described by participate and migrate relation between caste UNSC-member and caste Chair. Two types of members are represented by two sub-castes of UNSC-member, Permanent-member and Elected-member, respectively.

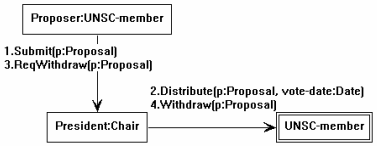
The collaboration models capture agents' interaction patterns that represent dynamic behaviours of the system. The notation of collaboration diagrams is shown in Figure 3. A collaboration model may consist of a set of scenario-specific collaboration diagrams that represent the interactions between agents in specific scenarios, and a general collaboration diagram that summarises the communications between agents.

For example, Figure 4 depicts the collaboration model of UNSC. Figures 4a and 4b describe the interactions between agents in the scenarios of voting on a proposal and withdrawal of a proposal, respectively. The general collaboration diagram, such as Figure 4c, describes all possible communications between all agents that may occur during the system's execution.

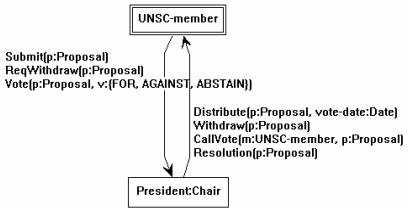
Figure 4. Collaboration model of UNSC information system



(a)

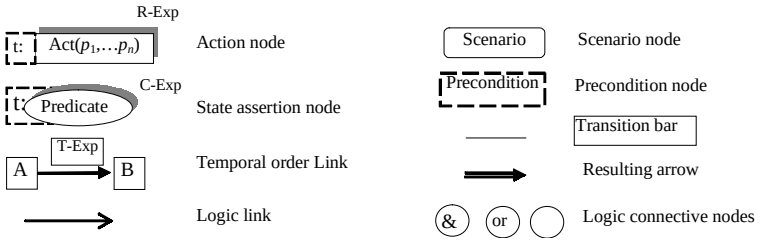


(b)



(c)

Figure 5. Notation of behaviour diagram



a. Scenario-specific diagram: Voting
b. Scenario-specific diagram: Withdraw
c. General collaboration diagram

Note that when an agent is decomposed into components, the interactions between the component agents also need to be specified. This results in a hierarchy of collaboration models defining the dynamic behaviours of agents at various granularities. Readers are referred to Shan and Zhu (2004b) for details about the process of collaboration modelling, the hierarchical structure of collaboration models, as well as examples.

While caste and collaboration models describe multi-agent systems at the macro-level from the perspective of an external observer, behaviour modelling adopts the internal or first-person view of each agent. It describes an agent's behaviour in terms of how it acts in certain scenarios of the environment at the micro-level. The notation of behaviour diagrams is shown in Figure 5. Readers are referred to Shan and Zhu (2003) for detailed explanation of the notation.

Each caste is associated with a behaviour diagram that describes the behaviour rules of its agents. In the UNSC example, there are two behaviour diagrams: one for caste Chair and the other for caste UNSC-member. Figure 6 depicts the behaviour diagram for caste Chair. The behaviour of a Chair agent is defined by four behaviour rules describing its actions under various circumstances, namely to distribute a proposal when some member submits the proposal, to withdraw the proposal when requested by the proposing member(s), to call all the members to vote on a proposal, and to quit from Chair when its turn finishes.

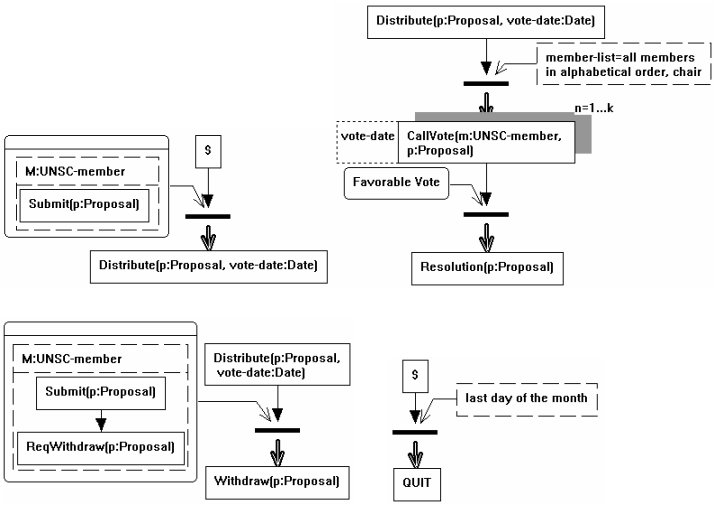
The castes Permanent-member and Elected-member inherit the behaviour rules of UNSC-member. They have no additional behaviour rules, thus require no different behaviour diagram than that of the UNSC-member.

The Modelling Environment

A software environment to support the process of analysis and modelling in CAMLE has been designed and implemented. It integrates the following set of tools:

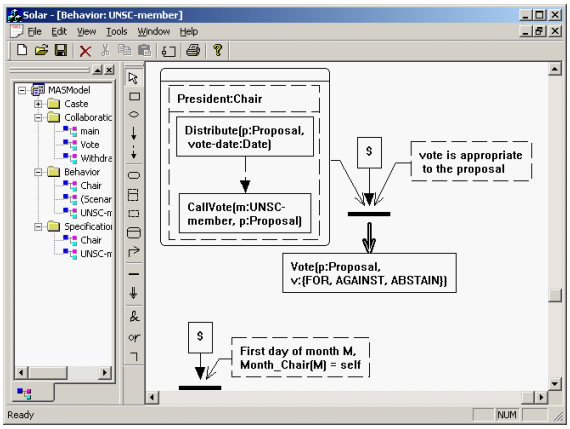
- **Model Construction and Management Tools:** A set of interactive diagram editors with graphic user interface are provided to enable the creation, editing, and modification of various diagrams in CAMLE models. These diagrams are organized and managed into development projects. Reuse of models from other projects is enabled. Figure 7 shows a screen snapshot of the CAMLE environment's interface.

Figure 6. Behaviour diagram of the chair in UNSC information system



- **Consistency Checkers:** A set of consistency constraints is defined on the CAMLE language to ensure that a set of diagrams form a meaningful model of an information system. The consistency of a model is checked by a set of tools to identify any violence of the constraints. Details of the consistency constraints and the implementation of the checkers can be found in Shan and Zhu (2004a).

Figure 7. CAMLE's graphical user interface for model construction



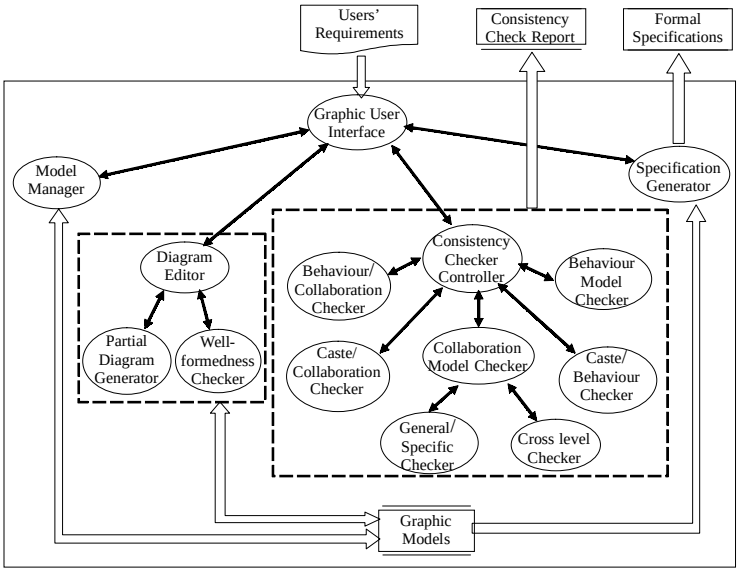
- **Specification Generator:** It transforms a well-defined model into a formal specification in SLABS. Details of the transformation algorithms can be found in Zhu and Shan (2005).

Figure 8 shows the architecture of the modelling environment. Readers are referred to Zhu and Shan (2005) for detailed description of the architecture and functionality of the CAMLE environment.

SPECIFICATION

One of the most appealing features of agent technology is its natural way to modularise complex systems in terms of multiple interacting autonomous components. This feature is supported by the language facility caste in SLABS for modular and composable specification of multi-agent systems. It bridges the gap between graphic modelling and implementation in the AOIS development process. The output of the modelling phase—a system model in CAMLE—is further analysed at the specifica-

Figure 8. The architecture of CAMLE environment



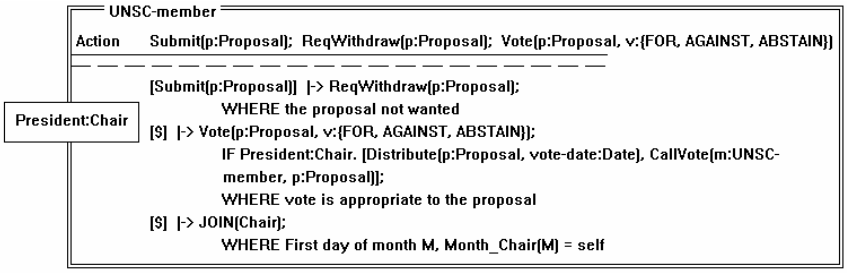
tion phase, which involves the following two main activities:

- **Generation of Formal Specifications:** As for all software developments, it is necessary to analyse the design of an agent-based system before the developers are committed to costly implementation. It is particularly true during the evolution of a system when new components are to be integrated into the existing system. Formal analysis of the new components in the context of the system is therefore highly desirable. However, the manual production of formal specifications of multi-agent systems is labour intensive, costly, time consuming, and error prone. With the help of the CAMLE modelling environment, formal specifications in SLABS can be automatically generated from graphic models in CAMLE.
- **Formal Analysis of the System:** Formal analysis can be applied on formal specifications in SLABS to prove the properties of the specified system. We have been devising a formal system Scenario Calculus to reason about the behaviours of multi-agent systems, especially their most complicated behaviours such as emergent behaviours (Zhu, 2005). If the formal reasoning about the system/new components based on the formal specification reveals that the system model is unsatisfactory on certain properties, the flow of the process goes back to the modelling phase to rectify the design. Thus, the process iterates the modelling and specification stages until a satisfactory model/specification is achieved.

The formal definition of the SLABS language and its meta-model can be found in Zhu (2001, 2003). A formal logic for reasoning about MASs' behaviours based on SLABS can be found in Zhu (2005).

Figure 9 shows an example of caste specification. It is the UNSC-member caste generated by the CAMLE environment's specifica-

Figure 9. Specification of UNSC-member caste in SLABS



tion generator from the CAMLE model of the UNSC system.

IMPLEMENTATION

A distinctive feature of our agent-oriented development methodology of information systems is that we aim at the direct implementation of information systems with a novel agent-oriented programming language that is based on the meta-model of the agent-oriented information system described in Zhu (2006). Such a programming language can significantly narrow the gap between specification and implementation. This section presents our research on the design and implementation of the agent-oriented programming language SLABSp and illustrates the style of programming through the running example.

SLABSp is designed to support the caste-centric approach to agent-oriented software development methodology by extending the object-oriented programming language Java (Shen, Wang, & Zhu, 2004; Wang, Shen, & Zhu, 2005a, 2005b, 2005c). As shown in Figure 10, it extends Java with three key concepts and language facilities: caste, scenario, and environment descriptions.

These language facilities become the dominant language facilities in the implementations of an AOIS and significantly change the styles of programming. In particular, caste becomes the basic program unit from which a complicated software system is built. Although class in object-orientation can still be used in the programming, it is now mainly used to define encapsulated data types that agents manipulate and use to represent agent states. Other Java constructs, such as Import statements, Expressions, Statements, and so on, are still legal language facilities, but they are extended to include identifiers to refer to agent states and actions, which are represented by preceding '#' and '~', respectively. There are also the additional join and quit statements to enable agents to dynamically join into and quit from castes.

Another significant change of programming style is the result of the introduction of the scenario description language facility. The syntax of scenario description is given in Figure 10, where an expression in the form of (a) describes the situation that a specific agent behaves in a certain pattern, where the agent is referred to by its name or keyword self. Expressions in the form of (b) describe the situation that the number of agents of a caste that behave in a certain pattern is within a specified interval, where the interval's boundaries are optional. The default value of the left boundary (i.e., the lower number) is 'zero'. When the right boundary is absent, it means 'all'—that is, the size of all the caste. The *count-condi-*

Figure 10. Syntax of SLABSp in EBNF

```
Caste ::= { Java-Import }
      'caste' Name [ ':' { Name / ',' }+ '{'
            { Environment }
            { State | Action | Rule }
      '}'
Agent ::= { Java-Import }
      'agent' Name [ ':' { Name / ',' }+ ] '{'
            { Environment }
            { State | Action | Rule }
      '}'
Environment ::= Name Id ';'

State ::= [ 'internal' ] Type '#'Id '(' Formal-Parameters ')' '{'
            { Java-Definition }
            'get' ':' Statement
            [ 'set' ':' Statement ]
      '}'
Action ::= [ 'internal' ] '~'Id '(' Formal-Parameters ')' '{'
            { Statement }
      '}'
Rule ::= 'rule' Id '(' Formal-Parameters ')'
      [ 'when' '(' Scenario ')' ]
      [ 'where' '(' Conditional-Expression ')' ]
      'do' '{' { Statement } '}'

Scenario ::= ( Name | 'self' ) Pattern                    (a)
      |   '<' [Number] ':' [Number] '>' Name Pattern      (b)
      |   Count-Conditional-Expression                    (c)
      |   Scenario ',' Scenario                           (d)
      |   Scenario '|' Scenario                           (e)
      |   '!' Scenario                                    (f)
      |   '(' Scenario ')'                                (g)
Pattern ::=
      '\' { ( Action-Pattern | State-Assertion )  / ',' } '\'
Action-Pattern ::=
      ('~' | '*' | '~'Id '(' Parameters ')' ) ['^' Number]
State-Assertion ::= Conditional-Expression
```

tional-expression in the form of (c) is an extension of Java *conditional-expression* with count-expression. The result of evaluating a count-expression is the number of agents in a caste that behave in the pattern. Expressions in the form of (d), (e), and (f) are the logic 'and', 'or', and 'not' combination of scenarios in the above forms, respectively. Expressions in the form of (g) are used to change the preference of the logic combinations. The uses of scenario descriptions in conjunction with agents' visible actions and environment descriptions enable communication and collaboration among agents to be described at a high level of abstraction and in the same style of conditional expressions in structured programming.

For example, the UNSC system can be implemented in SLABSp as shown in Figure 13. Based on the specification of the UNSC system, caste Member has three behaviour rules. Rule Withdraw will enable the agent to request the proposal to be withdrawn when the agent regards the proposal as inappropriate. Rule Vote will guide the agent to vote on the proposal with a specific attitude when the chair calls the

Figure 11. Fragments of UNSC system in SLABSp

```
import java.util.Calendar;

caste Member {

      ~Submit(Proposal proposal){
            // ...
      }

      ~ReqWithdraw(Proposal proposal){
            // ...
      }

      ~Vote(Proposal proposal, Attitude attitude){
            // ...
      }

      rule Withdraw(Proposal proposal)
      when ( self\~Submit(?proposal)\ )
      where ( proposal.isInappropriate() )
      do {
            ~ReqWithdraw(proposal);
      }

      rule Vote(Proposal proposal)
      when ( Chair\~CallVote(self, ?proposal)\  )
      do {
            // think about the proposal
            Attitude attitude = Attitude.FOR ; // or AGAINST, or ABSTAIN
            ~Vote(proposal, attitude);
      }

      rule AlternateJoin()
      when ( self\~\ )
      where (// the first day of a month
            ChairOfMonth(self, Calendar.getInstance().get(Calendar.MONTH))
        && (Calendar.getInstance().get(Calendar.DAY_OF_MONTH) ==
       Calendar.getInstance().getActualMinimum(Calendar.DAY_OF_MONTH)) )
      do {
            join Chair;
      }
}


// the caste for permanent members
caste PermanentMember : Member {
      // ...
}
```

```
import java.util.Date;
import java.util.Calendar;

caste Chair : Member {
      ~Distribute(Proposal proposal, Date date){ // ...
      }

      ~Withdraw(Proposal proposal){ // ...
      }

      ~Callvote(Member member, Proposal proposal){ // ...
      }

      ~Resolution(Proposal proposal){ // ...
      }

      rule Distribute(Proposal proposal)
      when ( self\~\, <1:1>Member\~Submit(?proposal)\ )
      do { // schedual voting a week later
            Date date = Calendar.getInstance()
                  .add(Calendar.DAY_OF_MONTH, 7).getTime();
            ~Distribute(proposal, date);
      }

      rule CallVote(Proposal proposal, Date date)
      when ( self\~Distribute(?proposal, ?date)\ )
      where ( Calendar.getInstance().getTime().equals(date) )
      do { // call all members (except the Chair) to vote, the call the Chair
            Collection members = Member#agents();
            members.remove(self);
            for(Member member: members)
                  ~CallVote(member, proposal);
            ~CallVote(self, proposal);
      }

      rule AlternateQuit()
      when ( self\~\ )
      where (// the last day of a month
            Calendar.getInstance().get(Calendar.DAY_OF_MONTH) ==
       Calendar.getInstance().getActualMaximum(Calendar.DAY_OF_MONTH) )
      do { quit Chair; }

      rule Resolution(Proposal proposal)
      when (  <:>Member\~Vote(?proposal, *)\, // all members have voted
            *Member\~Vote(proposal, FOR)\ > Member#population() / 2,
            !<1:>PermenantMember\~Vote(proposal, AGAINST)\ )
      do { ~Resolution(proposal); }
}
```

agent to vote. Rule AlternateJoin will trigger the agent to join caste Chair when it is its turn. Caste Chair extends caste Member with four additional rules. Rule Distribute will guide the Chair agent to distribute the proposal and schedule a voting date for each submitted proposal. Rule CallVote will direct the Chair agent to call the members to vote on the proposal on the voting date. Rule AlternateQuit will ask the current chair to quit from caste Chair when its turn finishes. Rule Resolution will define how a decision should be made based on the members' votes.

A runtime environment for the execution of multi-agent systems has been implemented as an extension of Java runtime environment. In particular, an automaton called the pattern process machine is designed and implemented to process patterns and scenarios. A compiler has been developed to translate SLABSp programs into Java and to execute in the runtime environment. More details can be found Shen et al. (2004) and Wang et al. (2005a, 2005b, 2005c).

The design and implementation of SLABSp demonstrate that caste and scenario are feasible as programming language facilities. Our experiences and experiments with the language clearly show that they provide power abstractions for AO programming. In particular, the caste facility enables the modularity in the concept of agents to be realized directly and in full strength. An obvious advantage of using scenarios to define agents' behaviours is that it can significantly reduce the unnecessary, explicit, message-based communications among agents. This also enables AO programming at a very high level of abstraction.

CONCLUSION

We now conclude the chapter with a summary of our main ideas and research results, and a comparison of our work with related works.

Summary

Our caste-centric methodology of agent-oriented information systems is based on a well-defined meta-model presented in Zhu (2006). It consists of a process model called the *growth model*, a set of languages including a *modelling language CAMLE* for the requirements analysis and design, a *formal specification language SLABS* and a *programming language SLABSp*, and a set of support tools including *CAMLE's modelling environment*, a formal reasoning system *Scenario Calculus*, and a *runtime support environment* of agent-oriented programs. A number of case studies on modelling, formal specification and verification, and programming have been conducted to develop the heuristics of using the languages and tools. Our methodology has the following features.

The methodology aims at modern information systems, especially those running on the Internet and the Web platforms. As argued in Zhu (2004), such systems belong to Lenman's E-type and are by nature evolutionary. The agent-oriented approach is very suitable for the development of such systems as we have shown in Zhu (2006). Moreover, the growth process model explicitly reflects the evolutionary characteristics of such systems and encourages the growth strategy, that is, the sustainable long-term evolution strategy of their lifecycle. This strategy is also strongly supported by the languages and tools.

The set of languages designed for use at different phases in the development and evolution of AOISs are based on a well-defined meta-model. The gaps between requirements specification, system and component design, and implementation are much smaller than their counterparts in other existing paradigms and approaches. In particular, the key concepts of agents and castes can be directly implemented in the agent-oriented programming language.

Our methodology is an extension of the current mainstream paradigm (the object-orientation) of information system development. In our model, object is a special degenerate form of agent. Agent-orientation provides a better metaphor for modelling the information systems in the real world than object-orientation. It can directly represent active and autonomous elements in information systems such as humans, independent information processing components such as Web services, and so on. It enables the design and implementation of computerised information systems in a structure that is closer to the structure of the system in the real world than object-orientation.

Finally, our approach to agent-orientation is caste-centric. In other words, caste plays the central role in our methodology. It is not just an abstract concept, but also a language facility that can be directly implemented in a programming language. It is the basic form of program unit from which complicated systems are constructed. It realises the kind of modularity inherent in the concept of agents. Our case studies show that caste can be used in a nice and straightforward way to model and implement various useful notions developed in agent technology, such as roles, agent society, collaboration protocols, normative behaviours, and so forth.

Related Work

Since Jennings (1999) advocated the notion of agent-oriented software engineering as a paradigm for building complex systems, a number of methodologies for agent-oriented software development have been proposed, such as MaSE (Wood & DeLoach, 2000), Gaia (Wooldridge, Jennings, & Kinny, 2000; Zambonelli, Jennings, & Wooldridge, 2003), Tropos (Bresciani, Perini, Giorgini, Giunchiglia, & Mylopoulos, 2004), and PASSI (Burrafato & Cossentino, 2002). A survey and analysis of the current state of the art in the research on agent-oriented software engineering can be found in Zambonilli and Omicini (2004).

As in early work on MAS engineering methodology, MaSE provides a development process covering the phases from capturing goals down to assembling agent classes and system design. Notations for representing system specifications in various stages and an environment supporting MAS development are developed (Wood & DeLoach, 2000). Gaia provides guides for analysis and design of agent-based systems with the view that a multi-agent system is a computational organization consisting of various interacting roles (Wooldridge et al., 2000). Role is adopted as the key concept, which is associated with responsibilities, permissions, activities, and protocols. The new version Gaia methodology advocates computational organization abstractions as the key abstraction of agent-based computing (Zambonelli et al., 2003). Tropos methodology emphasizes the use of the notion of agent and all the related mentalistic notions in all phases of software development, and purports to cover the very early phases of requirements analysis (Bresciani et al., 2004).

Despite the subtle differences in the research aims and focuses, the above works hold the same beliefs that the concept of agent is on a higher level of abstraction than object, thus agent-orientation will bring more efficiency to software engineering than object-orientation. Most of the existing methodologies attempt to exploit agents' advantages, such as autonomy and sociality using the mentalistic notions including goal, plan, role, and so on. Our work distinguishes from them in that the notions of caste and scenario, instead of the mentalistic notions, are the basic concepts for embodying agents' power.

Further Work

There is still a long way to go before agent-orientation become a mature development para-

digm of information systems. There are many issues remaining for future work. On the top of our research agenda is further investigation of the languages and tools in industrial context. We will connect the languages and tools with the ongoing development of Web technologies such as Web services, grid computing, and peer-to-peer computing. Another aspect of development methods that has not been discussed in depth in this chapter is testing, verification, and validation. We will further develop the formal reasoning system scenario calculus for analysing SLABS specifications and reasoning about the properties of emergent behaviours. We are also investigating software tools to support the formal reasoning. Automatic transformation from SLABS specifications to executable system is also in our agenda.

ACKNOWLEDGMENT

The work reported in this chapter is partly supported by the National Key Foundation Research and Development Program (973) of China under Grant No. 2005CB321802, the National High Technology R&D (863) Programme of China under grants No. 2002AA116070 and No. 2005AA113130, and the Program for New Century Excellent Talents in University.

REFERENCES

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.

Burrafato, P., & Cossentino, M. (2002). Designing a multi-agent solution for a bookstore with the PASSI methodology. *Proceedings of AOIS'02 at CAiSE'02.*

FIPA Agent UML Technique Committee. (2005). *Case studies of agent modelling: The Security Council of United Nations.* Retrieved May 30, 2005, from http://www.auml.org/auml/documents/

Jennings, N. R. (1999, June/July). Agent-oriented software engineering. In F. J. Garijo & M. Boman (Eds.), *Multi-Agent System Engineering, Proceedings of 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World* (pp. 1-7), Valencia, Spain. Berlin: Springer-Verlag (LNAI 1647).

Lehman, M. M., & Ramil, J. F. (2001). Rules and tools for software evolution planning and management. *Annals of Software Engineering, Special Issue on Software Management, 11*(1), 15-44.

Lehman, M. M. (1990). Uncertainty in computer application. *Communications of the ACM, 33*(5), 584-586.

Shan, L., & Zhu, H. (2003, October). Modelling and specification of scenarios and agent behaviour. *Proceedings of the IEEE/WIC Conference on Intelligent Agent Technology* (IAT'03) (pp. 32-38), Halifax, Canada.

Shan, L., & Zhu, H. (2004a, September). Consistency check in modelling multi-agent systems. *Proceedings of COMPSAC'04* (pp. 114-121), Hong Kong.

Shan, L., & Zhu, H. (2004b). Modelling cooperative multi-agent systems. *Proceedings of the 2nd International Workshop on Grid and Cooperative Computing* (pp. 994-1001), Shanghai, China. Berlin: Springer-Verlag (LNCS 3033).

Shan, L., & Zhu, H. (2005). CAMLE: A caste-centric agent-oriented modelling language and

environment. In R. Choren, A. Garcia, C. Lucena, & A. Romanovsky (Eds.), *Software engineering for multi-agent systems III: Research issues and practical applications* (pp. 144-161). Berlin: Springer-Verlag (LNCS 3390).

Spivey, J. M. (1992). *The Z notation: A reference manual* (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall.

Shen, R., Wang, J., & Zhu, H. (2004). Scenario mechanism in agent-oriented programming. *Proceedings of APSEC'04* (pp. 464-471), Busan, Korea.

Wang, J., Shen, R., & Zhu, H. (2005a, July 25-28). Agent-oriented programming based on SLABS. *Proceedings of COMPSAC'05* (pp. 127-132), Edinburgh, UK.

Wang, J., Shen, R., & Zhu, H. (2005b, September). Caste-centric agent-oriented programming. *Proceedings of the 1st International Workshop on Integration of Software Engineering and Agent Technology at QSIC'05,* Melbourne, Australia.

Wang, J., Shen, R., & Zhu, H. (2005c, July 27-29). Towards an agent-oriented programming language with caste and scenario mechanisms. *Proceedings of AAMAS'05,* Utrecht, The Netherlands.

Wood, M. F., & DeLoach, S. A. (2000). An overview of the multiagent systems engineering methodology. *Proceedings of AOSE 2000* (pp. 207-222).

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

Zambonelli, F., Jennings, N., & Wooldridge, M. (2003). Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology, 12*(3), 317-370.

Zambonilli, F., & Omicini, A. (2004). Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems, 9,* 253-283.

Zhu, H., & Shan, L. (2005). Caste-centric modelling of multi-agent systems: The CAMLE modelling language and automated tools. In S. Beydeda & V. Gruhn (Eds.), *Model-driven software development, research and practice in software engineering II* (pp. 57-89). Berlin: Springer-Verlag.

Zhu, H. (2001). SLABS: A formal specification language for agent-based systems. *International Journal of Software Engineering and Knowledge Engineering, 11*(5), 529-558.

Zhu, H. (2002). A growth process model and its supporting tools for developing Web-based software. *Acta Electronica Sinica, 30*(12A), 2090-2093.

Zhu, H. (2004, September). Cooperative agent approach to quality assurance and testing Web software. *Proceedings of COMPSAC'04 (Workshop Papers and Fast Abstracts), the Workshop on Quality Assurance and Testing of Web-Based Applications* (QATWBA'04) (pp. 110-113), Hong Kong.

Zhu, H. (2005, July). Towards formal reasoning about emergent behaviours of MAS. *Proceedings of SEKE'05* (pp. 280-285), Taipei.

Zhu, H. (2006). Towards an agent-oriented paradigm of information systems. In J.-P. Rennard (Ed.), *Handbook of research on nature inspired-computing for economics and management.* Hershey, PA: Idea Group Reference.

Zhu, H., Greenwood, S., Huo, Q., & Zhang, Y. (2000, July 30). Towards agent-oriented quality

management of information systems. *Proceedings of the 2nd International Bi-Conference Workshop on Agent-Oriented Information Systems at AAAI'2000* (pp. 57-64), Austin, TX.