Caste-Centric Modelling of Multi-Agent Systems: The CAMLE Modelling Language and Automated Tools

Hong Zhu[1] and Lijun Shan[2]

[1] Department of Computing, Oxford Brookes University
Wheatley campus, Oxford OX33 1HX, UK
hzhu@brookes.ac.uk
[2] Department of Computer Science, National University of Defence Technology
Changsha, 410073, China
lijunshancn@yahoo.com

Summary. Agent technology is widely perceived to be a viable solution for large-scale industrial and commercial applications in dynamic environments such as the Internet. However, the lack of rigour and language support in the analysis, specification, design and implementation of multi-agent systems has hampered the wide adoption of agent technology. This paper proposes a model-driven approach to the development of multi-agent systems. It combines graphic modelling with formal specification through automated tools . The paper reports an agent-oriented modelling language CAMLE and the automated tools in its modelling environment. Two aspects of particular importance in the model-driven development methodology are addressed in this paper. The first is the definition and implementation of consistency constraints on graphic models. The second is the automated transformation of graphic models into formal specifications.

1 Introduction

Agent technology has long been predicted to be the next mainstream computing paradigm; see, for example, [1, 2, 3]. It is widely perceived to be a viable solution for large-scale industrial and commercial applications in dynamic environments such as the Internet [4]. One of the key factors that contribute to the progress in software engineering over the past two decades is the development of language concepts and facilities that directly support increasingly powerful and natural high-level abstractions with which complex systems are modelled, analyzed and developed. From software engineering point of view, one of the most appealing features of agent technology is its natural way to modularise a complex system in terms of multiple, interacting and autonomous components that have particular objectives to achieve

[5]. Such modularity achievable by multi-agent systems (MAS) is much more powerful and natural than any kind of modularity that can be achieved by existing language facilities such as type, module and class.

However, the research on agent-based systems has been mainly an AI endeavour so far. The majority of extant agent applications are developed in an ad hoc fashion without proper analysis and specification of requirements, and without systematic verification and validation of the properties of the implemented systems. We believe that there are two major factors that hamper the wide adoption of agent technology in software development. First, being autonomous, proactive and adaptive, agent-based systems can be very complicated. They may demonstrate emergent behaviours , which sometimes are neither designed by the developers nor expected by the users. The new features of agent-based systems demand new methods for the specification of agent behaviours and for the verification and validation of their properties to enable software engineers to develop reliable and trustworthy agent-based systems. It has been recognised that the lack of rigour is one of the major factors hampering the wide-scale adoption of agent technology [6]. Second, extant MAS are mostly developed without a proper language facility that directly supports the effective and efficient utilisation of the modularity and abstraction underlying the concept of agents. Due to the lack of language support, the advantages and merits of agent technology are inevitably overwhelmed by the inefficiency of the implementations in incompatible languages, and the high cost and poor productivity due to the unnecessary complexity in design, coding, debugging and testing at a lower level of abstraction, etc.

In this paper, we propose a model driven approach to the development of agent-based systems. It combines graphic modelling of MAS with implementation-independent formal specifications in order to provide the rigour in the analysis, specification and design of MAS. The central concept of the approach is caste, which is a language facility introduced as a natural evolution of the notion of data type in procedural programming and class in object-oriented paradigm. It is the classifier of agents. It serves as the template of agents and the organisational unit of multi-agent systems. This language facility is intended to bridge the gap between the abstract concepts of agent and their concrete representations in computer software so that MAS applications can be developed effectively and efficiently. In this paper, we present an informal introduction to the modelling language CAMLE, which stands for Caste-centric Agent-oriented Modelling Language and Environment [7]. We also report an automated modelling environment that supports the users to construct MAS models at requirements analysis and specification stage in CAMLE graphical notation with multiple views and at different abstraction levels. We will focus on two aspects of particular importance in the tool support to model driven software development. They are the consistency problem of graphic models and the automation problem of model-based development. Diagrammatic models in CAMLE serve as a representation of users' requirements and used as the bases for further design and implementation of MAS.

It is therefore of vital importance to ensure model's consistency [8]. A set of consistency constraints are defined on CAMLE models and an automatic consistency checker is designed and implemented to help the detection of inconsistency according to the constraints. As in many existing modern modelling languages that employ the so-called multiple view principle , the information that specifies one caste of agents is scattered over various diagrams. It is therefore desirable to specify each caste of agents in one 'module' that contains all necessary information for its further design and implementation without unnecessary knowledge of other parts of the system. This is achievable through an automated tool that transforms graphic models into formal specifications in SLABS (a Specifictation Language for Agent-Based Systems)[9, 10], which describes a multi-agent system with a set of specifications of the castes that the sytem contains.

The remainder of the paper is organised as follows. Section 2 presents the meta-model of MAS, which is independent of the implementation platforms and applicable to all types of agent theories and techniques. Section 3 is an informal introduction to the modelling language CAMLE. Section 4 defines the consistency constraints on models in CAMLE. Section 5 describes the algorithms and rules that transform graphic models in CAMLE to formal specifications in SLABS. Section 6 describes the architecture and main functions of the automated modelling environment and reports the case studies with the modelling language and environment. Section 7 concludes the paper with a discussion of the further research.

2 Meta-Model of Multi-Agent Systems

Because the concepts of agents and MAS are controversial, it is worthy spending a few words to clarify what we mean by agent and MAS and how such systems work. Generally speaking, a consistent definition of the basic concepts, structures and mechanisms underlying a specific type of systems forms a conceptual model (sometimes also called meta-model) of these systems. A conceptual model of MAS, therefore, must answer a set of fundamental questions about agents and MAS. For example, what is the structure of agent? How do agents perform their activities? What constitute a MAS? How do agents in a MAS communicate to each other? How are agents in a MAS organised? etc.

Our conceptual model can be characterized by a set of pseudo-equations. Each pseudo-equation answers such a question and thus defines a key feature of MAS. A formal definition of the model can be found in [9, 10].

Pseudo-equation (1) states that agents are defined as real-time active computational entities that encapsulate data, operations and behaviours, and situate in their designated environments.

$$Agent = \langle Data, Operations, Behaviour \rangle_{Environment} \tag{1}$$

Here, data represent an agent's state. Operations are the actions that the agent can take . Behaviour is described by a set of rules that determine how the agent behaves including when and how to take actions and change state in the context of its designated environment. By encapsulation , we mean that an agent's state can only be changed by the agent itself, and the agent can decide 'when to go' and 'whether to say no' according to an explicitly specified set of behaviour rules. Fig.1 illustrates the control structure of agent's behaviour.

```
Begin
    Initialise state;
    Loop
        Perceive the visible actions and states of the agents in its
            environment;
        Take actions and change state according to the situation in the
            environment and its internal state;
    end of loop;
end
```

Fig. 1. The control structure of agent's behaviour

There are two fundamental differences between objects and agents in our conceptual model. First, objects do not contain any explicitly programmed behaviour rule. Second, objects are open to all computation entities to call its public methods without any distinction of them. However, as argued in [9], objects can be considered as agents in a degenerate form. In particular, object is a special case of agent in the sense that it has a fixed rule of behaviour, i.e. 'executes the corresponding method when receives a message'. Consequently, in our conceptual model, a MAS consists of agents and nothing but agents, as stated in pseudo-equation (2).

$$MAS = \{Agent_n\}, n \in Integer \tag{2}$$

Notice that, an agent's state variables and actions are divided into two kinds: visible ones and invisible (or internal) ones. An agent taking a visible action can be understood as generating an event that can be perceived by other agents in the system, while an agent taking an internal action means it generates an event that can only be perceived by its components, which are also agents. Similarly, the value of a visible state variable can be obtained by other agents, while the value of an internal state can only be obtained by its components. Notice that, our use of the term 'visibility' is different from the traditional concept of scope.

The concept of visibility of an agent's actions and state variables forms the basic communication mechanism in our conceptual model. Agents communicate with each other by taking visible actions and changing visible state variables, and by observing other agents' visible actions and visible states, as shown in pseudo-equation (3).

$$Communication(A \rightarrow B) = A.Action\&B.Observation \tag{3}$$

However, an agent's visible action is not necessarily observed by all agents in the system. It is only observed by those interested in its behaviour and considering it as a part of their environments. In other words, the environment of an agent in a MAS at time t is a subset of the agents in the system. As illustrated in pseudo-equation (4), from a given agent's point of view, only those in its environment are visible. In particular, from agent A's point of view, agent B is visible means that agent A can perceive the visible actions taken by agent B and obtain the value of agent B's visible part of state.

$$Environment_t(Agent, MAS) \subseteq MAS - \{Agent\} \tag{4}$$

To enable our model to deal with open and dynamic environments, we introduced the concept of 'designated environment' , i.e. the environment of an agent is specified when the agent is designed, but the specification allows the environment to vary within a certain range. Therefore, the set of agents in the environment of an agent depends on time, hence, the subscription t in pseudo-equation (4). The language facility that enables us to achieve the variation of environment is the concept of caste.

In our conceptual model, the classifier of agents is called caste . Agents are classified into various castes in the way similar to that data are classified into types, and objects are classifies into classes. However, different from the notion of class in object orientation, caste allows dynamic classification. That is, an agent can change its caste membership (called casteship in the sequel) at run time. It also allows multiple classifications, i.e. an agent can belong to more than one caste at the same time. As all classifiers, inheritance relations can also be specified between castes. As a consequence of multiple classifications, a caste can inherit more than one caste. Caste is the basic organizational unit in the design and implementation of MAS. As a modularity language facility, a caste serves as a template that describes the structure and behaviour properties of agents. Pseudo-equation (5) states that a caste at time t is a set of agents that have the same structural and behavioural characteristics.

$$Caste_t = \{agents | structure\ \&\ behaviour\ properties\} \tag{5}$$

The weakness of static object-class relationship in current mainstream object-oriented programming has been widely recognized. Proposals have been advanced, for example, to allow objects' dynamic reclassification [11]. In [12], we suggested that agents' ability to dynamically change its roles is represented by dynamic casteship. In our model, dynamic casteship is an integral part of agents' behaviour capability. Agents can have behaviour rules that allow them to change their castes at run-time autonomously. To change its casteship, an agent takes an action to join a caste or retreat from a caste at run time. Therefore, which agents are in a caste depends on time even if agents can be persistent, hence the subscript of t in pseudo-equation (5). We believe

that this feature allows users to model the real world by MAS naturally and to maximize the flexibility and power of agent technology.

Moreover, dynamic caste membership enables us to describe agents' designated environments in a flexible and effective way. The environment description of an agent (or a caste) defines what kinds of agents are visible. With the concept of caste, we can describe an environment, for example, as the set of agents in a number of particular castes. An environment such described is neither closed, nor fixed, nor totally open. Since agents can change their casteships dynamically, an agent's environment may change dynamically as well. For example, an agent's environment changes when it joins a caste and hence the agents in the caste's environment become visible. The environment also changes when other agents join the caste in the agent's environment.

It is worthy noting that the conceptual model defined above is independent of any implementation platform and applicable to all types of agent theories and techniques.

3 The CAMLE Modelling Language

In this section, we give an informal introduction to the modelling language CAMLE and illustrate its uses by a simple example.

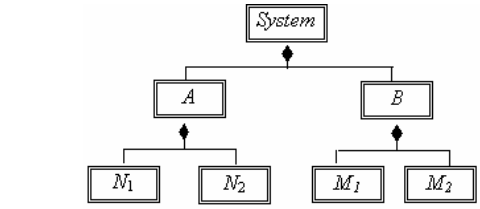
3.1 The overall structure of models

CAMLE employs the multi-view principle to model complicated systems. There are three types of models in CAMLE: caste models, collaboration models and behaviour models. Each model may consist of one or more diagrams. A caste model specifies the castes of agents in the system and the relationships between them, such as the inheritance and whole-part relations. A collaboration model specifies how the agents interact with each other. A behaviour model specifies how an agent decides it actions and state changes.

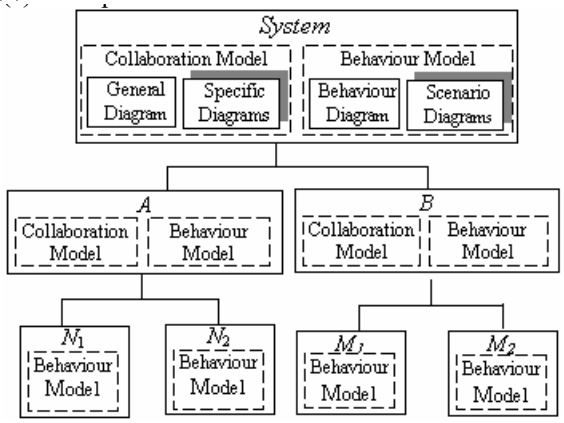
A caste is called a compound caste if its agents are composed from a number of other agents; otherwise, it is called atomic. A MAS can, therefore, be considered as a compound agent. For example, as shown in Fig.2(a), the System is directly composed of castes A and B. Each of them can be further decomposed into smaller components N_1 and N_2, and M_1 and M_2, respectively. To each compound caste, such as the System, A and B in Fig.2, a collaboration model and a behaviour model are associated. Atomic castes only have no collaboration models because they have no components, thus no internal collaboration.

The overall structure of a system's collaboration models and behaviour models can be viewed as a hierarchy, which is isomorphic to the whole-part relations between castes described in the caste model; see e.g. Fig.2(b).

The following subsections describe each model and discuss their uses in agent-oriented software development.



(a) Example of caste model with whole-part relations



(b) Collaboration models and behaviour models

Fig. 2. Overall structure of CAMLE models

3.2 Caste model

We view an information system as an organization that consists of a collection of agents that stand in certain relationships to one another by being a member of certain groups and playing certain roles, i.e. in certain castes. They interact with each other by observing their environments and taking visible actions as responses. The behaviour of an individual agent in a system is determined by the 'roles' it is playing. An individual agent can change its role in the system. However, the set of roles and the assignments of responsibilities and tasks to roles are usually quite stable [13]. Such an organizational structure of information systems is captured in our caste model.

Fig.3 shows the notation and an example of caste diagrams. A caste diagram identifies the castes in a system and indicates the relationships between them. In CAMLE, there are three types of relationships on castes represented in caste models. They are inheritance, aggregation and migration relations.

The inheritance relationship between castes defines sub-groups of the agents that have special responsibilities and hence additional capabilities and behaviours. For example, in Fig.3, the members of a university are classified into three castes: students, faculties and secretaries. Students are further classified into three sub-castes: undergraduates, postgraduates and PhD students.

Migration relations specify how agents change their casteships. There are two kinds of migration relationships: migrate and participate. A migrate relation from caste A to B means that an agent of caste A can retreat from caste A and join caste B. A participate relation from caste A to B means that an agent of caste A can join caste B while retaining its casteship of A. For example, in Fig.3, an undergraduate student may become a postgraduate after graduation. A postgraduate student may become a PhD student after graduation or become a faculty member. Each student becomes a member of the alumni of the university after leaving the university. A faculty member can become a part time PhD student while remaining employed as a faculty member. From this model, we can infer that an individual can be a student and a faculty member at the same time if he/she is a PhD student.

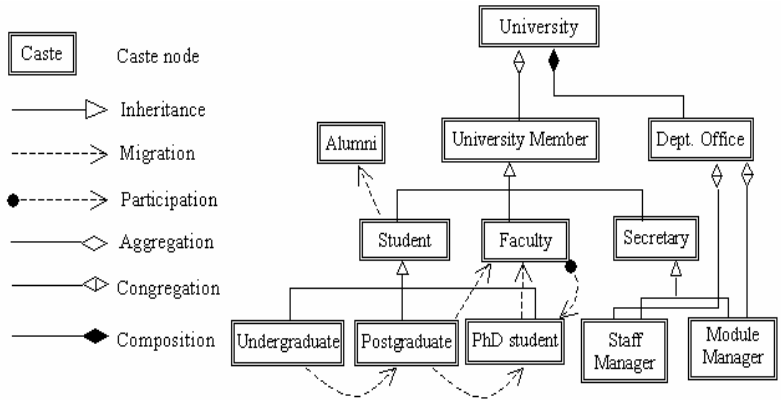


Fig. 3. Caste diagram: notations and example

An aggregate relation specifies a whole-part relationship between agents. An agent may contain a number of components that are also agents. The former is called compound agent of the latter. In such a case, there exists a whole-part relationship between the compound and the component agents, which is represented through an aggregate relation between castes. We identify three types of whole-part relationships between agents according to the ways a component agent is bound to the compound agent and the ways a compound agent controls its components. The strongest binding between a compound agent and its components is composition in which the compound agent is responsible for creation and destruction of its components. If the compound agent no longer exists, the components will not exist. The weakest binding is aggregation, in which the compound and the component are independent, so that the component agent will not be affected for both its existence and casteships when the compound agent is destroyed. The third whole-part relation is called congregation . It means that if the compound

agent is destroyed, the component agents will still exist, but they will lose the casteship of the component caste. For example, as shown in Fig.3, a university consists of a number of individuals as its members. If the university is destroyed, the individuals should still exist. However, they will lose the membership as the university member. Therefore, the whole-part relationship between University and University Member is a congregation relation. This relationship is different from the relationship between a university and its departments. Departments are components of a university. If a university is destroyed, its departments will no longer exist. The whole-part relationship between University and Department is therefore a composition relation. The composition and aggregation relation is similar to the composition and aggregation in UML , respectively. However, congregation is a novel concept in modelling languages. It was introduced by CAMLE. There is no similar counterpart in object oriented modelling languages, such as UML. It has not been recognized in the research on object-oriented modelling of whole-part relations; cf. [14]. We believe that it is important for agent-oriented modelling because of agents' basic features viz. dynamic casteship.

3.3 Collaboration model

While caste model defines the static architecture of MAS, collaboration model defines a dynamic aspect of the MAS organization by capturing the collaboration dependencies and relationships between the agents.

Agents in a MAS collaborate with each other through communication, which is essential to fulfil the system's functionality. Such interactions between agents are captured and represented in a collaboration model. In CAMLE, a collaboration model is associated to each compound caste and describes the interactions between the component agents of the compound caste through a set of collaboration diagrams. Fig.4 gives the notations of collaboration diagrams.

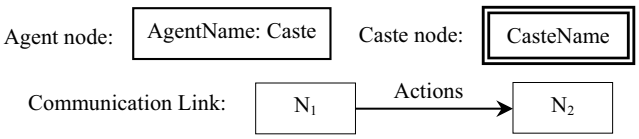


Fig. 4. Notation of collaboration diagram

There are two types of nodes in a collaboration diagram. An agent node represents a specific agent. A caste node represents any agent in a caste. An arrow from node A to node B represents that the visible behaviour of agent A is observed by agent B. Therefore, agent A influences agent B. When agent B is particularly interested in certain activities of agent A, the activities can also be annotated to the arrow from A to B.

Although this model looks similar to the collaboration diagrams in UML, there are significant differences in the semantics. In OO paradigm, what is annotated on the arrow from A to B is a method of B. It represents a method call from object A to object B, and consequently, object B must execute the method. In contrast, in CAMLE the action annotated on an arrow from A to B is a visible action of A. Moreover, agent B does not necessarily respond to agent A's action. The distinction indicates the shift of modelling focus from controls represented by the method calls in OO paradigm to collaborations represented by signalling and observation of visible actions. It fits well with the autonomous nature of agents.

3.3.1 Scenarios of collaboration

One of the complications in the development of collaboration models is to deal with agents' various behaviours in different scenarios. They may take different actions, pass around different sequences of messages even communicate with different agents. Therefore, it is better to describe different scenarios separately . The collaboration model supports the separation of scenarios by including a set of collaboration diagrams. Each diagram represents one scenario. In such a scenario-specific collaboration diagram, actions annotated on arrows can be numbered by their temporal sequence. Fig.5 below gives an example of scenario-specific collaboration diagram. It describes the collaborations of an undergraduate student with his/her personal tutor, the faculty members who give lectures and the PhD students who are practical class tutors.

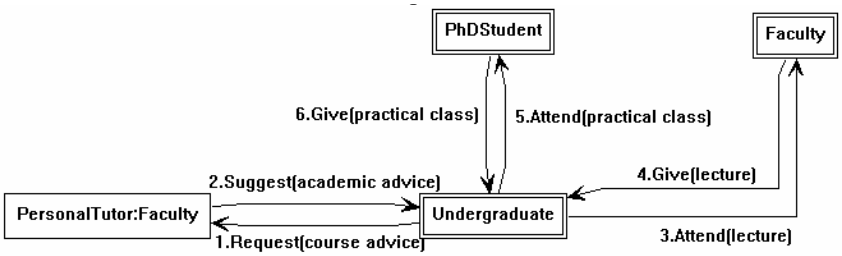


Fig. 5. An example of scenario-specific collaboration diagram

In addition to scenario-specific collaboration diagrams, a general collaboration diagram is also associated to each compound caste to give an overall picture of the communications between all the component agents by describing all visible actions that the component agents may take and all possible observers of the actions. Fig.6 describes the communications within a department between various agents.

3.3.2 Refinement of collaboration models

The modelling language supports modelling complex systems at various levels of abstraction. Models of coarse granularity at a high level of abstraction can be refined into more detailed fine granularity models. At the top level, a system can be viewed as an agent that interacts with users and/or other systems in its external environment. This system can be decomposed into a number of subsystems interacting with each other. A sub-system can also be viewed as an agent and further decomposed. As analysis deepens, a hierarchical structure of the system emerges. In this way, the compound agent has its functionality decomposed through the decomposition of its structure. Such a refinement can be carried on until the problem is specified adequately in detail. Thus, a collaboration model at system level that specifies the boundaries of the application can be eventually refined into a hierarchy of collaboration models at various abstraction levels. Of course, the hierarchical structure of collaboration diagrams can also be used for bottom-up design and composition of existing components to form a system.

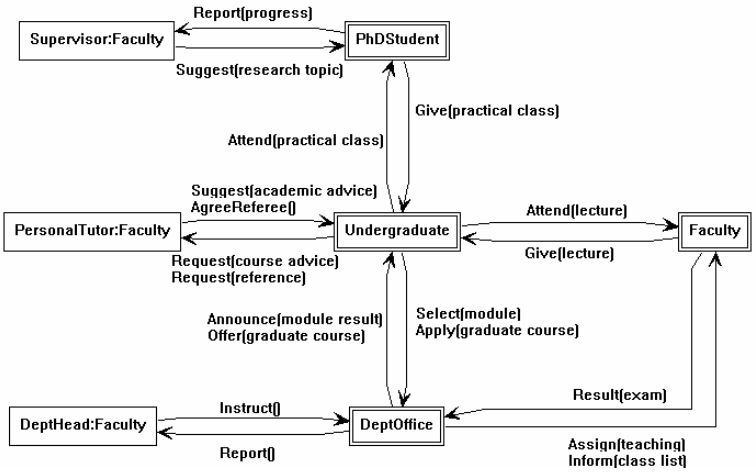


Fig. 6. An example of general collaboration diagram

Fig.7 gives an example of general collaboration diagram that refines the caste Dept Office. In this diagram, the agents in the castes of Student and Faculty as well as a specific agent called Dept Head in the caste of Faculty form the environment of the caste Dept Office. Therefore, they are visible for the component agents of the caste.

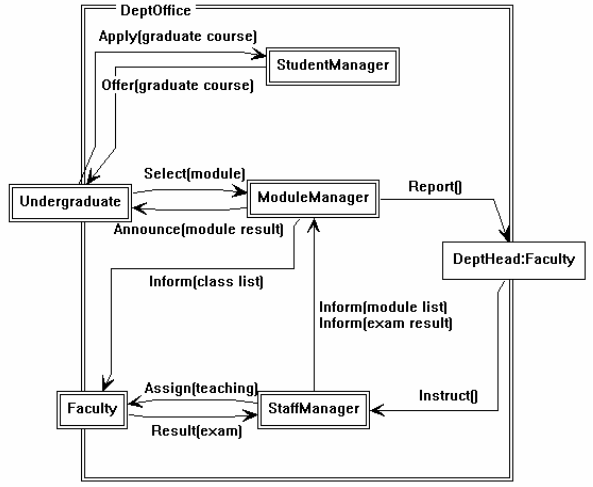


Fig. 7. An example of general collaboration diagram that refines a caste

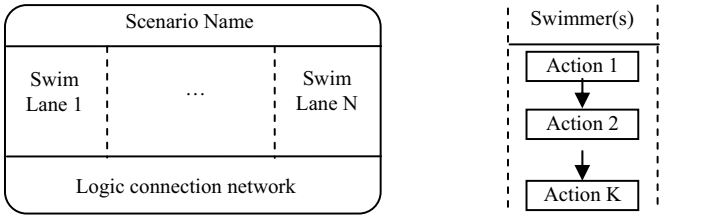
3.4 Behaviour model

While caste and collaboration models describe MAS at the macro-level from the perspective of an external observer, behaviour model adopts the internal or first-person view of each agent. It describes an agent's dynamic behaviour in terms of how it acts in certain scenarios of the environment at the micro-level. A behaviour model consists of two kinds of diagrams: scenario diagrams and behaviour diagrams.

3.4.1 Scenario diagrams

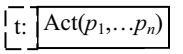
We believe that each agent's perception of its environment should be explicitly specified when modelling its behaviour. From an agent's point of view, the situation of its environment is characterized by what is observable by the agent. In other words, a scenario is defined by the sequences of visible actions taken by the agents in its environment. Scenario diagrams identify and describe the typical situations that the agent must respond to. In Fig.8 below, part (a) shows the layout of scenario diagrams and part (b) shows the layout of swim lanes.

The swimmer(s) of a swim lane can be in one of the following forms. (a) $\forall x \in C$, where C is a caste and x is a bounded variable. It means all agents of caste C take the same sequence of actions specified in the swim lane. (b)$\exists x \in C$, where C is a caste and x is a bounded variable. It means there is at least one agent in caste C that takes the sequence of actions specified in the swim lane. (c) $\alpha \in C$, where α is an agent in caste C. It means that agent α takes the sequence of actions specified in the swim lane.

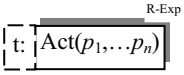


(a) The layout of scenario diagrams (b) The layout of swim lanes

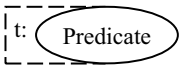
Fig. 8. Format of scenario diagram



Single action node: the agent takes action *Act* with parameters p_1,…p_n at time *t* .

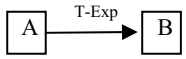


Repetitive action node: the agent takes action *Act* repetitively starting at time *t*, where *R-Exp* defines the number of repetitions of the action.

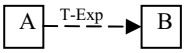


State assertion node: the agent's state satisfies the predicate at time *t*.

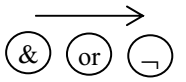
Continuous state assertion node: the agent's state satisfies the predicate for a continuous period of time starting from *t*, where the period satisfies the expression *C-Exp*.



Temporal order between events: event *B* is immediately after event *A*, where *T-Exp* is the constraint on the time gap between event *A* and *B*.



Temporal order between events: event *B* is after event *A*, while there may be other events between them, where *T-Exp* is the constraints on the time gap between the events.



Logic links connect logic connective nodes

Logic connective nodes

Fig. 9. Notations of scenario diagram

Fig.9 depicts the notations to specify visible events by nodes and temporal ordering by arrows in scenario diagrams, as well as logic connective nodes and links for the combination of situations.

For example, Fig.10 describes a scenario where Greenspan announces that the interest rate will decrease by 0.25 points and all stock market analysts recommend buy Microsoft's share.

3.4.2 Behaviour diagrams

A behaviour diagram is associated to a caste to define a set of behaviour rules for the agents of the caste. The notation of behaviour diagrams includes the notation of scenario diagrams plus those in Fig.11.

A behaviour diagram contains event nodes linked together by the temporal ordering arrows as in scenario diagrams to specify the agent's previous

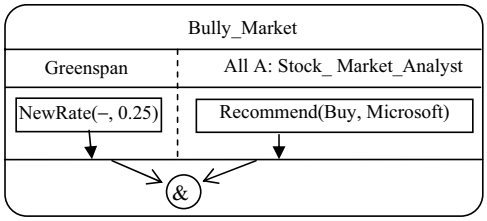


Fig. 10. Example of scenario diagram

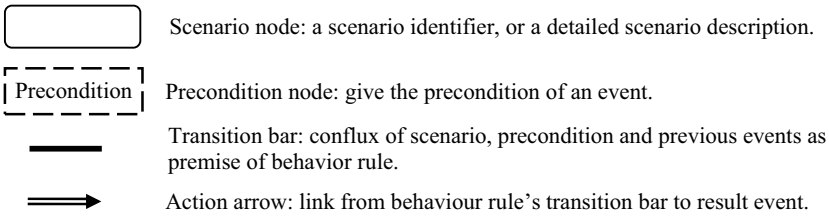


Fig. 11. Notation for behaviour diagrams

behaviour pattern. A transition bar with a conflux of scenario, precondition and previous pattern and followed by an event node indicates that when the agent's behaviour matches the previous pattern and the system is in the scenario and the precondition is true, the event specified by the event node under the transition bar will be taken by the agent. In a behaviour diagram, a reference to a scenario indicated by a scenario node can be replaced by a scenario diagram if it improves the readability.

For example, the behaviour diagram in Fig.12 defines the behaviour of an undergraduate student. It states that if the student is in the final year and the average grade is 'A', he/she may request a reference from the personal tutor for the application of a graduate course. If the personal tutor agrees to be a referee, the student may apply for a graduate course. If the department office offers a position in a graduate course, the student will join the Graduates caste and retreat from the Undergraduates caste.

4 Consistency Constraints on the Models

Consistency constraints are the conditions on the uses of diagrammatic notations, variables and names, types and symbols so that a set of well-formed diagrams can be regarded as a meaningful model. These conditions are usually related to the semantics of the diagrams. However, in order to enable the automated checking of a model effectively and efficiently, consistency constraints often have to be simplified and represented as syntactic rules.

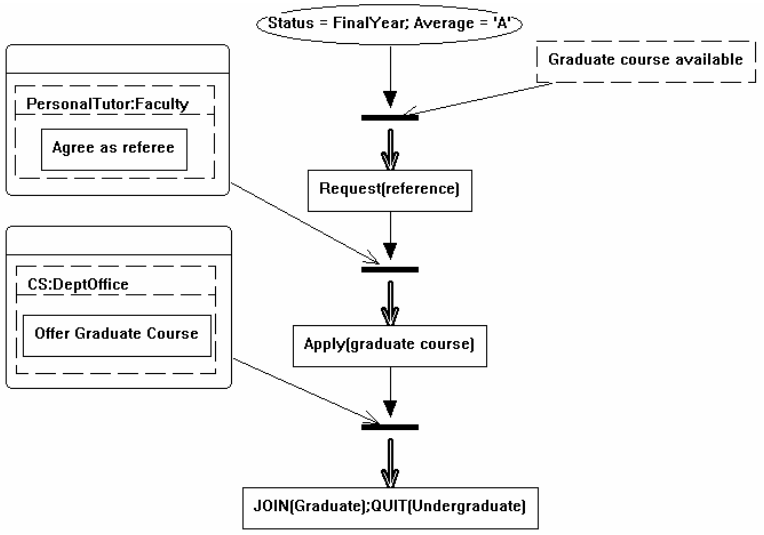


Fig. 12. An example of behaviour diagram

A typical example of such a consistency constraint is that the same identifier that occurs at different places must refer to the same entity and an entity should be referred to by the same identifier even if it occurs at different places in the model. This rule cannot be mechanically checked directly. Instead of checking the consistency against this rule directly, we check a set of syntactical rules that represents the consistency in some more concrete forms. For example, we can use one model as the declaration of all entities and check all other occurrences of identifiers against the declaration. An alternative approach is to check if the entities referred to by the same identifier have the same features. Such rules are necessary conditions of the consistency rather than sufficient ones. However, well-defined consistency conditions can significantly improve the quality of models just like type compatibility checking in programming languages can detect programming errors.

In this section we define the consistency constraints for the CAMLE language. These constraints are classified into two types. Intra-model consistency constraints are those conditions that only involve the diagrams of the same type. Inter-model constraints involve more than one type of diagrams.

4.1 Intra-model consistency

4.1.1 Constraints on caste models

As discussed in section 3.2, a caste diagram defines the castes in the system and three kinds of relationships between them: inheritance, aggregation and migration. A well-formed caste diagram must satisfy the following conditions.

Constraint (1a) *A caste diagram defines a naming space. In this naming space each node defines a caste with a unique name.*
Constraint (1b) *Each link defines a binary relation on castes by linking two nodes in the diagram.*
Constraint (1c) *An inheritance relation and a migration relation must be associated to two different caste nodes.*
Constraint (1d) *Inheritance relations must not form any loops in a caste diagram.*

Aggregation and migration relations are allowed to form loops. It is not required for an aggregation relation to be associated to different caste nodes.

4.1.2 Constraints on collaboration models

A collaboration model may contain a number of collaboration diagrams including a general collaboration diagram (GCD) and a set of scenario-specific collaboration diagrams (SCD). A GCD serves as a declaration of what castes and their instance agents are involved in the collaborations, while SCDs define the details of the collaboration protocols in various scenarios. Each SCD specifies a linear sequence of actions taken by the agents in a specific scenario of collaboration. To be well-formed a collaboration diagram must satisfy the following conditions.
Constraint (2a) *A caste or agent node in a collaboration diagram must have a unique name.*
Constraint (2b) *The number assigned to an action must be unique, if any.*

Let G be a GCD, $\boldsymbol{S}$ be the set of SCD and $D \in \boldsymbol{S}$ be any given SCD. Let $ANode(X)$, $CNode(X)$ and $Node(X)$ denote the set of agent nodes, the set of caste nodes and the set of all nodes in the collaboration diagram X, respectively. Let $CName(x)$ denote the caste name of a node x. The nodes and arrows in G and those in $\boldsymbol{S}$ must satisfy the following consistency conditions.
Constraint (2c) *Every agent node in the GCD G must appear in at least one SCD. Formally,*

$$\forall n \in ANode(G).\exists D \in \boldsymbol{S}.(n \in ANode(D)) \tag{6}$$

Constraint (2d) *A caste node in the GCD must appear at least once in a SCD as either a caste node or an agent node representing a specific agent of the caste. Formally,*

$$\forall n \in CNode(G).\exists D \in \boldsymbol{S}.$$
$$(n \in CNode(D) \vee \exists n' \in ANode(D).(CName(n') = CName(n))) \tag{7}$$

Constraint (2e) *Every caste node in a SCD must also appear in the GCD. Formally,*

$$\forall D \in \boldsymbol{S}.\forall n \in CNode(D).(n \in CNode(G)) \tag{8}$$

Constraint (2f) *For every agent node in any SCD, there must be either a node of the same agent or the caste of the agent in the GCD. Formally,*

$$\begin{gathered}\forall D \in \boldsymbol{S}.\forall n \in ANode(D).\\ (n \in ANode(G) \vee \exists n' \in CNode(G).(CName(n') = CName(n)))\end{gathered} \tag{9}$$

Assume that $a = Act(p_1, p_2, ..., p_n)$ is an action associated to an arrow from node b to c. We call $\langle a, b, c\rangle$ an interaction from b to c with action a and define $Action(\langle a, b, c\rangle) = a$, $Begin(\langle a, b, c\rangle) = b$, and $End(\langle a, b, c\rangle) = c$. Let $Interaction(X)$ be the set of all interactions in a collaboration diagram X.
Constraint (2g) *Every interaction in the GCD must appear in at least one SCD, where a caste node in GCD can be replaced by an agent node of the same caste in the SCD. Formally,*

$$\begin{gathered}\forall \alpha Interaction(G).\exists D \in \boldsymbol{S}.\exists \beta \in Interaction(D).\\ (CName(Begin(\alpha)) = CName(Begin(\beta))\\ \wedge CName(End(\alpha)) = CName(End(\beta))\\ \wedge Action(\alpha) = Action(\beta)\\ \wedge Begin(\alpha) \in ANode(G) \Rightarrow Begin(\beta) \in ANode(D)\\ \wedge End(\alpha) \in ANode(G) \Rightarrow End(\beta) \in ANode(D))\end{gathered} \tag{10}$$

Constraint (2h) *Every interaction in any SCD must also be defined in the GCD. Formally,*

$$\begin{gathered}\forall D \in \boldsymbol{S}.\forall \alpha \in Interaction(D).\exists \beta \in Interaction(G).\\ (CName(Begin(\alpha)) = CName(Begin(\beta))\\ \wedge CName(End(\alpha)) = CName(End(\beta))\\ \wedge Action(\alpha) = Action(\beta)\\ \wedge Begin(\alpha) \in CNode(G) \Rightarrow Begin(\beta) \in CNode(D)\\ \wedge End(\alpha) \in CNode(G) \Rightarrow End(\beta) \in CNode(D))\end{gathered} \tag{11}$$

As discussed in section 3.3.2, CAMLE supports the decomposition of an agent into a number of component agents in the same way as the analysis of the whole system. The collaboration among the component agents can also be defined by a set of collaboration diagrams. Thus, the consistency between diagrams at different levels in the hierarchy of collaboration models of a system must be ensured. Let X be a collaboration diagram for a caste. We use $Env(X)$ to denote the environment of X, i.e. the set of agent and caste nodes

on the boarder of X.

Constraint (2i) *The environment of a SCD must be identical to the environment of the GCD. Formally,*

$$\forall D \in \boldsymbol{S}.(Env(D) = Env(G)) \tag{12}$$

For the sake of simplicity, we assume that a collaboration model M satisfies the consistency constraints within one model discussed above. Therefore, we can overload the notation $Env(X)$ defined on diagrams to be the environment of the model, i.e. for a model M and any diagram D in M, define $Env(M) = Env(D)$, provided that M satisfies condition (2i).

Let C be a compound caste in a collaboration model M, and M_C be the collaboration model for C. That is, M_C specifies the collaborations between C's components. The environment of C defined in M should be consistent with the environment description in M_C. The following two constraints are imposed on the models at different levels.

Constraint (2j) *The set of agents and castes in C's environment described in M must be equal to the set of agents and castes in M_C's environment description. Formally,*

$$\begin{gathered}\forall n.(n \in Env(M_C) \Leftrightarrow \\ \exists \alpha \in Interaction(G).(n = Begin(\alpha) \wedge C = End(\alpha)));\end{gathered} \tag{13}$$

where G is the GCD in M.

Constraint (2k) *The interactions that C participates as an observer described in M must be realized as interactions between environment elements and C's components in M_C. Formally,*

$$\begin{gathered}\forall \alpha \in Interaction(G).\exists \beta \in Interaction(G_C). \\ (End(\alpha) = C \Rightarrow Begin(\alpha) = Begin(\beta) \\ \wedge Action(\alpha) = Action(\beta) \\ \wedge Begin(\beta) \in Env(G_C) \\ \wedge End(\beta) \in Component(G_C));\end{gathered} \tag{14}$$

where G_C is the GCD in M_C and $Component(G_C)$ is the set of C's components depicted in G_C.

4.1.3 Constraints on behaviour models

A behaviour model associated to a caste may contain two kinds of diagrams: scenario diagrams (SD) and behaviour diagrams (BD). The following well-formedness conditions are imposed on BDs and SDs.

Constraint (3a) *The temporal order between events must be linear, i.e. the in-degree and out-degree of an event node must be less than or equal to 1.*
Constraint (3b) *The logic connective nodes 'AND' and 'OR' are binary operators, and 'NOT' is unitary operator.*
Constraint (3c) *A transition bar has at most three nodes directly connected to it: at most one scenario (may be a logical combination of several scenario nodes), at most one pre-condition node, and at most one event node.*

Each scenario reference node in a BD refers to a scenario defined in a SD. Therefore, a consistency condition on the relationship between a BD and the SDs in one behaviour model is defined as follows.
Constraint (3d) *The set of scenarios referred to in a BD by using scenario reference nodes is a subset of the scenarios defined by SDs. Formally, let C be a caste, D_C be the behaviour diagram of caste C, and $\boldsymbol{S}_C$ be the set of scenario diagrams of C.*

$$\forall n \in ScenarioNode(D_C).\exists S \in \boldsymbol{S}_C.(Name(n) = Name(S)). \quad (15)$$

4.2 Inter-model Consistency

This subsection discusses the consistency between different types of models, viz. the inter-model constraints. In the sequel, models are assumed to be consistent with regard to the intra-model constraints defined above.

4.2.1 Consistency between collaboration models and caste models

Let CD be the set of collaboration diagrams in a collaboration model, and C the caste model for the system in question.

Constraint (4a) *The set of the castes in collaboration model must be a subset of the castes in caste model. Formally,*

$$\forall D \in CD.\forall n \in Node(D).\exists n' \in Node(C).(CName(n) = Name(n')). \quad (16)$$

It is possible that a caste in the caste model does not appear in any collaboration diagram. For example, a caste can be an abstract caste, which has no direct instance agent and any instance of the caste is always an instance of its sub-caste. The behaviours of the agents of the abstract caste can be defined by its sub-castes. Consequently, the abstract caste may not occur in any collaboration diagram.

Let CM be the collection of collaboration models of the system. Let x be a caste in the system, and M_x be the collaboration model for x. For models

M_A and M_B in CM, we say that M_B is an immediate refinement of model M_A and write $M_B \lhd M_A$, if B is the component caste of caste A. Let $Aggr(C)$ be the set of aggregation relations in the caste model C.

Constraint (4b) *The hierarchical structure of the collaboration models must be consistent with the whole-part relations between castes defined in caste diagram. Formally,*

$$\forall M_A, M_B \in CM.(M_B \lhd M_A \Rightarrow \exists R \in Aggr(C).(R(B, A)) \tag{17}$$

4.2.2 Consistency between behaviour models and caste models

Due to the existence of inheritance relations, some castes may have no explicit behaviour definition. Therefore, we have the following consistency conditions on the relationship between a caste model and the set of behaviour models.

Let BM be the set of behaviour models of a system, and C the caste model. The caste with a behaviour model X defining its behaviour is denoted by $Caste(X)$.

Constraint (4c) *Each behaviour model B in BM defines the behaviour of a caste and the caste must be in the caste model. Formally,*

$$\forall B \in BM.\exists n \in Node(C).(Caste(B) = n). \tag{18}$$

In a behaviour model, say, of caste B, the description of scenarios may refer to the agents in the environment of B. Let $Agents(B)$ be the set of agents referred to in a behaviour model B, $CasteOf(x)$ the caste of such an agent.

Constraint (4d) *Every agent in a scenario in a behaviour model must have its caste defined in the caste model. Formally,*

$$\forall B \in BM.\exists a \in Agents(B).\exists n \in Node(C).(CasteOf(a) = Name(n)). \tag{19}$$

In a caste model, an agent's change of casteship is described through a migration relation between the castes. In a behaviour model, an agent's change of casteship is defined through actions $JOIN(caste)$, $MOVETO(caste)$ and $QUIT$. Such information in the behaviour model must be consistent with the caste model.

Constraint (4e) *Let BC be the behaviour model for caste C.*

* *BC contains an action $JOIN(C')$, where C' is a caste name, if and only if there is a participate migration relation from C to C' in the caste model.*
* *If BC contains an action $MOVETO(C')$, where C' is a caste name, there must be a migrate relation from C to C' in the caste model.*
* *If BC contains an action $QUIT$, there must be a migrate relation from C to some caste in the caste model.*
* *If there is a migrate from C to some caste (say C') in the caste model, there must be either a $MOVETO(C')$ or $QUIT$ action in the behaviour model of C.*

By 'an action in a behaviour model', we mean a result action of a behaviour rule, depicted as an action node immediately after a transition bar in a BD.

4.2.3 Consistency between collaboration models and behaviour models

Both collaboration models and behaviour models define the behaviour of agents. However, collaboration models define the behaviours of agents from an inter-agent interaction point of view, while behaviour models are from the view of agents' internal activities. Due to the overlap in the information provided by these two types of models, consistency between them is of particular importance.

Let $Components(C)$ be the set of C's component castes.

Let $VisibleActions(C)$ be the set of visible actions of caste C defined in the collaboration model. Let B_X be the behaviour model for caste X, $Rules(B)$ be the set of rules in the behaviour model B, and $Action(r)$ be the result action of the rule r.

Constraint (4f) *Every visible action of caste C defined in the collaboration models must occur in the behaviour model of C or at least one of C's components as a result action. Formally,*

$$\begin{gathered}\forall a \in VisibleActions(C).\\ ((\exists r \in Rules(B_C).(a = Action(r))\\ \vee(\exists M \in Components(C).\exists r \in Rules(B_M)).(a = Action(r)))\end{gathered} \tag{20}$$

Let G be a caste or agent that has a communication link to caste C in the collaboration model. We call G a collaborator of caste C and write $Collaborators(C)$ to denote the set of C's collaborators. Let $Scenarios(B)$ be the set of scenarios used in a behaviour model B, and $Ref(Sc)$ denote the set of castes or agents that a scenario Sc refers to.

Constraint (4g) *For each scenario used in the definition of caste C's behaviour, the agents and/or castes that the scenario refers to must occur in the collaboration model as C's collaborators. Formally,*

$$\forall Sc \in Scenarios(BC).\forall G \in Ref(Sc).(G \in Collaborators(C)). \quad (21)$$

Notice that, an actor in a scenario may be specified with qualifier, e.g. '$\forall A : CasteX$', and '$\exists Y : CasteX$'. In such cases, the caste $CasteX$ must be a collaborator of caste C. If the actor of a scenario refers to a specific agent, i.e. in the form of '$AgentM : CasteX$', the agent $AgentM$ of caste $CasteX$ must be a collaborator.

Constraint (4h) *The agents and castes referred to in a scenario must be elements in the environment of the caste described by the collaboration model. Formally, let C to be the caste described by a behaviour model B.*

$$\forall Sc \in Scenarios(B).\forall G \in Ref(Sc).(G \in Env(C)), \quad (22)$$

where $Env(X)$ is the set of caste and agents in X's environment description.

The collaboration between an agent A of caste C and other agents may be realized through the collaboration of A's component agents. Therefore, we do not require all collaborators of caste C to be referred to in the definition of caste C's behaviour.

Let $p_1, p_2, ..., p_n$ be the sequence of actions of a caste C (or an agent of caste C) described in a scenario Sc. Each p_i, $i = 1, 2, ..., n$, is called a referred action of caste C in scenario Sc. We write $ReferredActions(C, Sc)$ to denote the set of all such actions.

Constraint (4i) *Every referred action in a scenario used in a behaviour diagram must be a visible action of the caste described by the scenario. Formally,*

$$\forall Sc \in Scenarios(BC).\forall a \in ReferredActions(C, Sc). \\ (a \in VisibleActions(C)). \quad (23)$$

It is not required that all visible actions of a collaborator should be referred to in the definition of a caste's behaviour, because the collaboration may be realized through component agents.

4.3 Discussion

Consistency conditions can play at least two important roles in model-driven development. First, consistency conditions serves as check points for quality assurance in modelling process. Violation of consistency conditions indicates the existence of contradictions in the model. Therefore, automatic consistency check can help engineers to detect errors at modelling stage, hence prevent errors from being propagated to later stages. Inconsistency may also be caused by conflict in requirements. Consistency checks on requirement models help to identify and thereafter to resolve and manage such conflict. Second, in

model-driven development of software systems, it is desirable to automatically transform one model to another model (maybe partial model), and to generate code (or code framework) from models. Design and implementation of such tools must ensure that the transformation rules preserve the models' meanings. Therefore, consistency between the original and the resultant must be guaranteed. Consistency conditions provide a means to formally specify the correctness of the transformation rules.

The consistency constraints defined above have been used for both of the above purposes in the implementation of CAMLE environment [4]. The consistency constraints defined in this paper are computable and have been directly implemented in the environment as consistency check tools. Diagnostic information as the result of the check is recorded to help users to locate and correct errors. The partial diagram generator in the environment generates partial models (incomplete diagrams) from existing diagrams to help model construction. The rules to generate partial models are based on the consistency constraints so that the generated partial diagrams are consistent with existing ones. Preliminary case studies show that both consistency check and partial model generation are very helpful to improve the quality of models and software engineers' productivity. Besides model construction and consistency check, another main function of CAMLE environment is to automatically transform graphic models into the formal specifications in SLABS. Consistency check also simplifies the implementation of the automatic transformation because less error processing is required.

Well-defined visual notations for modelling software systems' structures and behaviours have the advantages of readability and preciseness due to their semi-formal nature. A common feature of such visual notations is that multiple views are utilized to model a system's different aspects and/or at different levels of abstraction. Since different views emphasise on different aspects of a system or at different levels of abstraction, consistency between the views has become a serious problem in the development of models. It is a crucial quality attribute of software models. It is widely recognised as very desirable to automatically check the consistency of software models [15, 16]. However, due to the semi-formal nature of modelling languages, the definition of effective and computable consistency constraints is a difficult and nontrivial problem [17]. Most existing modelling languages, for example UML , have no explicitly defined consistency constraints.

The past few years has seen a rapid increase in the research on defining consistency conditions and implementing consistency check tools for modelling languages, especially for UML [18, 19, 20, 21]. Among the related works on consistency check, Xlinkit is a flexible tool for checking the consistency of distributed heterogeneous documents [22]. It comprises a language for expressing constraints between such documents, a document management mechanism and an engine that checks the documents against the constraints. In comparison with Xlinkit, our approach is language specific. The direct implementation of consistency constraints as a part of modelling environment is highly efficient

and effective in detecting errors. In addition, the explicitly defined constraints form a base for automatic transformations between models. Formal methods, such as model checking, have also been used for checking the consistency between multiple views of software specifications, e.g. in [23, 24]. It requires translating models into a formal notation as the input to a model checker, while assumes that syntactic errors have been removed before the translation. Therefore, to check consistency before translation is still necessary.

5 Automatic Generation of Formal Specifications

As shown in the previous sections, graphic models in the CAMLE notation are suitable for the representation of users' requirements. To further develop MAS in a modular way in which castes are used as the templates of agents and the basic organisational units of software systems, it is desirable to specify MAS with modularity. That is, all information required to design and implement a caste should be specified in one module, but nothing more. However, in the CAMLE language, the information about a caste is scattered over various diagrams. This section presents the rules and algorithms that transform models in CAMLE to formal specifications in SLABS, which provide modular specifications of MAS.

5.1 The specification language SLABS

SLABS is a model-based specification language with the conceptual model described in section 2 as its meta-model [9, 10].

A formal specification in SLABS consists of a set of descriptions of castes. Fig.13 shows the structure of the description of a caste in SLABS. The clause '$C \Leftarrow C_1, C_2, ..., C_n$' specifies that caste C inherits the structure, behaviour and environment descriptions of existing castes $C_1, C_2, ..., C_n$. The environment description explicitly specifies a subset of the agents in the system that may affect the agent's behaviour. The state space of an agent is described by a set of variables with keyword VAR. The set of actions is described by a set of identifiers with keyword ACTION.

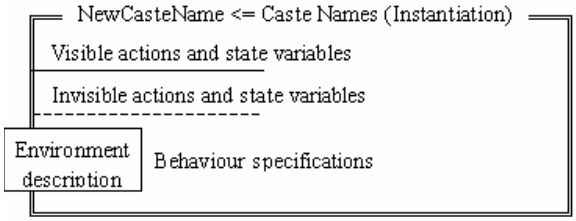


Fig. 13. Caste descriptions in SLABS

A behaviour rule has the following structure.

$BehaviourRule ::=$

$[\langle RuleName \rangle] Pattern[Prob] \rightarrow Event$,[**if** $Scenario$][**where** $PreCond$];

A pattern describes the behaviour of an agent by a sequence of observable state changes and actions. In addition to the pattern of individual agents' behaviour, SLABS also provides the facility of scenario to describe global situation of the whole system. Informally, a scenario is a set of typical combination of the behaviours of related agents in the system. The syntax of scenarios is given below.

$Scenario ::= AgentName : Pattern \mid AtomicPredicate$
$\mid \exists_{[ArithmeticExp]} AgentVar \in CasteName : Pattern$
$\mid \forall AgentVar \in CasteName : Pattern$
$\mid Scenario \& Scenario | Scenario \vee Scenario | \ Scenario$
$Pattern ::= [\{Event[\parallel Constraint]/, \}]$
$Event ::= [TimeStamp :][Action][!StateAssertion]$
$Action ::= AtomicPattern[\hat{}ArithmeticExp]$
$AtomicPattern ::= \$ \mid\sim\mid ActionVariable$
$\mid ActionIdentifier[(\{ArithmeticExp\})]$
$TimeStamp ::= ArithmeticExp$

An informal definition of the semantics of various forms of scenarios and patterns is given in Table 1. The following are some examples of scenarios.

$$\exists p \in Parties : t2004 : [Nominate(Bush)] \parallel t2004 = (March/2004). \quad (24)$$

It describes the situation that at least one agent in the caste called Parties took the action $Nominate(Bush)$ at the time of March 2004.

$$(\mu x \in Voter : [vote(Bush)] > \mu x \in Voter : [vote(Kerry)]). \quad (25)$$

It describes the situation that there are more agents in the caste Voter who took the action of $vote(Bush)$ than those in the caste who took the action of $vote(Kerry)$.

An important feature of the formal specification language SLABS is that it provides a modular specification of MAS in which each caste is specified by one caste description. Each caste description contains all necessary information about one caste but nothing more. The analysis, design and implementation of a caste can be based on the caste description without referring to other units. The modular specifications in SLABS are composable and reusable [25]. Therefore, it is more suitable to be used for further development of MAS than graphic models where the specification of a caste is spread over a number of diagrams due to the multiple view principle.

5.2 The overall transformation algorithm

The following algorithm translates each caste in a CAMLE model into a caste description in SLABS. Various parts of caste description are generated according to the information spread in various models. In the sequel, we assume

Table 1. Semantics of scenario descriptions

Pattern/Scenario	Meaning
$	The wild card, it matches with all actions
$\sim$	The silence event
Action variable	It matches an action
$P\hat{}k$	A sequence of k events that match pattern P
$Action(a_1, \cdots, a_k)$	An action takes place with parameters that match $(a_1, \cdots, a_k)$
$!Predicate$	The state of the agent satisfies the $Predicate$
$[p_1, \cdots, p_n]$	The previous sequence of events match the patterns $p_1, \cdots, p_n$
$A : P$	Agent A's behaviour matches pattern P
$\forall X \in C : P$	The behaviours of all agents in caste C match pattern P
$\exists_{[m]} X \in C : P$	There are at least m agents in caste C whose behaviour matches pattern P. The default value of m is 1
$\mu X \in C : P$	The number of agents in caste C whose behaviour matches P
$S_1 \wedge S_2$	Both scenario S_1 and scenario S_2 are true
$S_1 \vee S_2$	Either scenario S_1 or scenario S_2 or both are true
$\sim S$	Scenario S is not true

that graphic models are consistent with regards to the consistency constraints defined in section 4.

ALGORITHM 1. {Overall}

INPUT: $\langle CM, CLM, BM \rangle$, /* CM is a caste model,
/* CLM is a collaboration model, and
/* BM is a behaviour model

OUTPUT: $\{C_i\}_{i \in I}$, /* C_i is a caste description, $i \in I$.

BEGIN
FOR each node N in caste model CM DO
BEGIN /* Generate a caste description with caste name N
/* Step 1: Generate inheritance clause
IF there is an inheritance arrow from node N to node A in CM,
THEN $A \in Ancestors(N)$;
/* Step 2: Generate environment description
IF there is an arrow from node X to node N in a CD in CLM
THEN
CASE X OF
X is an agent node with label '$A : CasteName$':
'$A : CasteName' \in Environment(N)$
X is a caste node with label '$CasteName$':
'$All : CasteName' \in Environment(N)$
END_CASE;
/* Step 3: Generate visible actions and variables
FOR each collaboration model CD in CLM that contains N
DO IF there is an arrow from N to X with 'Action' annotated
on the arrow

THEN '$Action' \in VisibleAction(N)$;
END_FOR;
/* Step 4: Generate invisible actions and variables
FOR each collaboration diagram of caste N
DO IF there is an arrow from caste N to a component node X
with '*Action*' annotated on the arrow
THEN '$Action' \in InvisibleAction(N)$;
END_FOR;
/* Step 5: Generate behaviour rules
$GenerateBehaviourRule(BM_N)$,
/* BM_N is the behaviour model of caste N.
END_FOR
END_ALGORITHM

The generation of castes' behaviour rules is more complex compared with other parts of caste's structure. It is discussed in the next sub-section.

5.3 Generation of behaviour descriptions

Generation of a caste's behaviour description from a behaviour diagram consists of two main steps. The first is to recognize the rules in a network of interconnected nodes in the diagram. The second is to generate a behaviour rule in SLABS syntax from each rule recognized in the first step. The algorithm is as follows.

ALGORITHM 2. {Generate Behaviour Rules};
INPUT: BM_N /* a behaviour model for caste N
OUTPUT: $R = \{r_i\}_{i \in I}$,
/* a set of behaviour rules in SLABS syntax for caste N
VARIABLE: $P = \{p_i\}_{i \in I}$, /* a set of rules recognized from BM_N
BEGIN
$P := RecogniseRules(BM_N)$;
FOR each p_i in P DO $r_i := TranslateRule(p_i)$ END_FOR
END_ALGORITHM

5.3.1 Recognition of behaviour rules

In a behaviour diagram, several behaviour rules may be depicted independently or interconnected. The recognition of behaviour rules is achieved through an analysis of the diagram's structure. It converts a diagram into a set of graphically unconnected rules. Fig.14 shows the structure of rules.

When a behaviour diagram contains several interconnected behaviour rules such as in Fig.15, the number of the transition bars in the diagram determines the number of the rules contained in the diagram. For example, three rules can be recognized from the behaviour diagram given in Fig.12. The recognition algorithm uses the transition bars in the diagram as boundaries between various rules. For instance, in Fig.15 the sequence of event nodes between the first

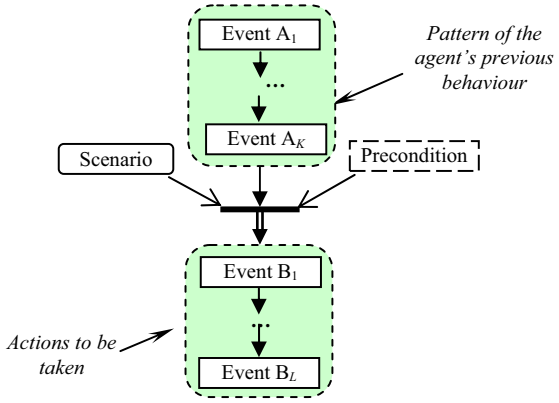


Fig. 14. Structure of behaviour rule in behaviour diagrams

and the second transition bar are the first rule's result-events. They are also the second rule's pre-events. Generally, the event nodes on the path from transition bar T_1 to transition bar T_2 are result events of the rule corresponding to T_1. They are also the pattern of the rule corresponding to T_2.

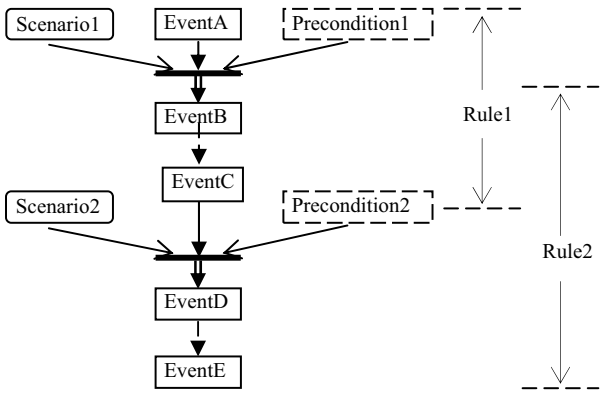


Fig. 15. Recognition of rules in behaviour diagrams

5.3.2 Translation of rules into SLABS format

As shown in Fig.16, a behaviour rule defines the cause and effect of an agent's behaviour through five parts: (a) a scenario that describes the situation in the environment, (b) a pattern that describes the agent's own previous behaviour, (c) a pre-condition on the agent's internal state, (d) a sequence of

resulting events that specifies the actions to be taken, and (e) a transition bar that links these parts together. The first three parts, which are connected to the transition bar through logical and temporal links, constitute the premise of a rule to define 'when to go'. The transition bar is connected to one or a sequence of event nodes, which indicates 'what to do'. Fig.16 gives the rule for transforming a behaviour rule in diagrammatic notation to SLABS syntax. Fig.17 shows a typical behaviour rule, which governs UN-SC member's behaviour in a voting process, and its equivalent form in SLABS syntax.

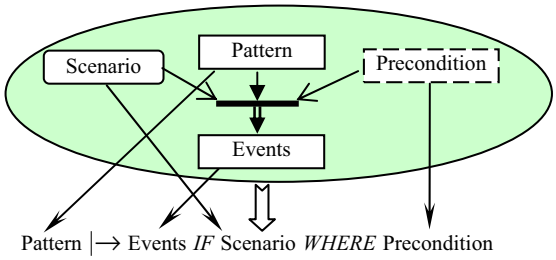


Fig. 16. Top level transformation rule

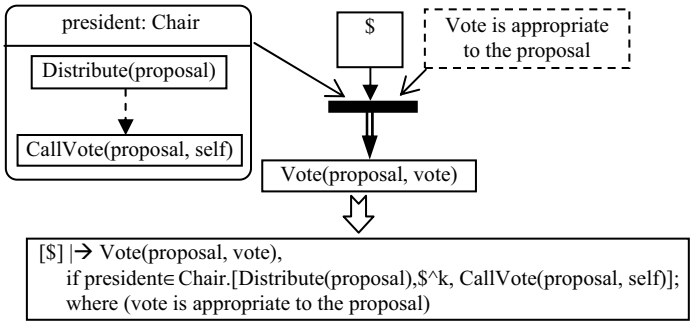


Fig. 17. Example of the transformation of behaviour rules

The translation of the precondition of a behaviour rule from the precondition node in the behaviour diagram is fairly straightforward and the details are omitted for the sake of space. The translation of scenarios and patterns deserve a few words.

5.3.3 Transformation rules for behaviour patterns

In a behaviour diagram, a pattern as a list of events in a behaviour rule is depicted as a set of action nodes or state nodes connected by temporal

links. Therefore, the formal specification of a pattern can be derived as a combination of the specifications of the events. Fig.18 illustrates some of the transformation rules for various kinds of nodes and links.

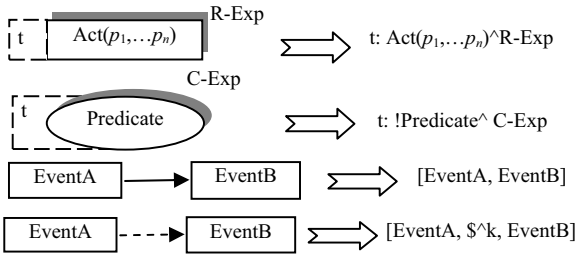


Fig. 18. Transformation rules for nodes and links

5.3.4 Transformation rules for scenarios

A scenario description node consists of three parts: the scenario name, a set of swimming lanes and a logical connective network comprising logical connective nodes and links which connect the set of swimming lanes. Fig.19 shows the transformation rule for swimming lanes, where Qu is a qualifier $\forall$ or $\exists$.

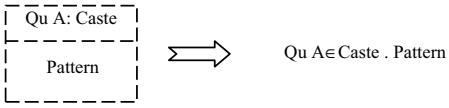


Fig. 19. Transformation rule for pattern nodes

Fig.20 shows the formal specification generated by the CAMLE tools from the behaviour diagram given in Fig.12.

5.4 Discussion

Our approach to the development of MAS follows model-driven development (MDD) point of view. Models are not just supportive documents for facilitating implementation, but they are also treated as indispensable part of software artefacts. Model transformation , therefore, is widely recognised as the heart of MDD. It can serve for various development purposes, such as model refactoring [26], PIM-to-PIM and PIM-to-PSM [27], code generation [28], etc.; see e.g. [29] for a classification of the kinds of transformations that can be performed during MDD activities. Different to the above works, our purpose of

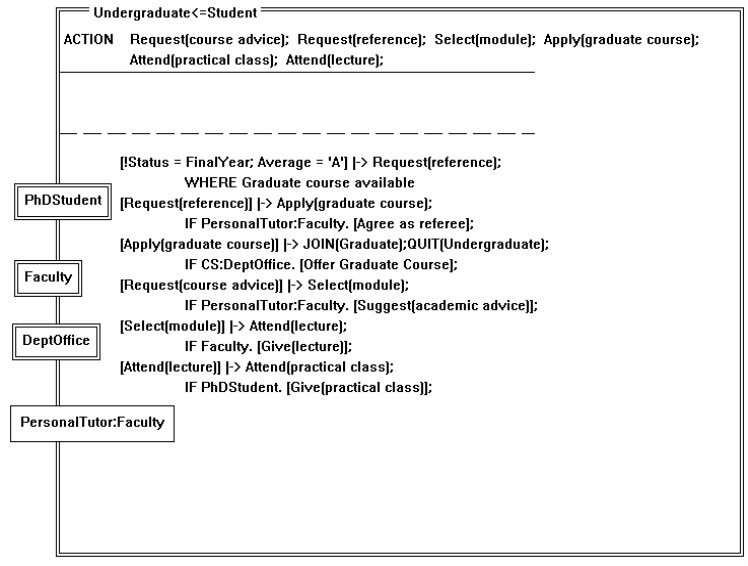


Fig. 20. An example of automatically generated formal specification

transformation from CAMLE models to SLABS formal specification is for combining the advantages of informal and formal methods.

In software engineering literature, a number of proposals have been advanced to combine graphic notation and formal methods, such as the employment of dual languages and method integration [30]. In our previous work, an automated tool was developed to translate structured models of software requirements definitions into Z [31]. A flexible framework to define mappings from graphic models to formal specifications was proposed in [32]. A prototype program to convert an adapted form of UML class diagrams into specifications in the B language was reported in [33]. The work in [34] presented some schemes of the derivation of B specifications from UML behavioural diagrams. An alternative approach is to project formal specifications back to diagrammatic models. For example, techniques were presented in [30] to transform the integrated formalism to UML diagrams. Another approach is to combine diagrammatic notation with formal notation in one language. [35] discussed how UML can be augmented with Z in the Unified Process. The work closely related to this paper is perhaps that reported in [32], which employed two languages and defined mappings from front-end notations to formal models. The customisable framework works with different front-end notations and formal models. It supports mappings of analysis results obtained on the formal model back to the front-end notation chosen by the practitioners. In comparison, our approach is language-specific, but more efficient.

6 The CAMLE Modelling Environment

Modelling environments containing automated tools can play a significant role in MDD as discussed in [36] of this book and demonstrated in [37, 38, 26, 39, 28] for the tools that supports various MDD activities. This section gives a brief description of our automated modelling environment.

6.1 The overall architecture

A software environment to support the process of system analysis and modelling in CAMLE has been designed and implemented. The main functionalities of the environment are:

(1) *Model construction*. It consists of a set of graphical editors to support the construction of models and tools for version control and configuration management.
(2) *Model consistency check*. It checks if a model satisfies the consistency constraints defined in section 4.
(3) *Automated generation of formal specifications*. It provides the function of transforming graphic models into the corresponding formal specifications in SLABS.

Fig.21 shows the architecture of the environment.

In addition to the consistency checker and formal specification generator that have been discussed in detail in section 4 and 5, respectively, the diagram editor supports the manual editing of models through a graphic user interface. The well-formedness checker ensures that the user entered models are well-formed. The diagram generator can generate partial models (incomplete diagrams) from existing diagrams to help users in model construction. The rules to generate partial models are based on the consistency constraints so that the generated partial diagrams are consistent with existing ones according to the consistency conditions.

6.2 Case studies

A number of systems have been modelled in CAMLE and their formal specifications in SLABS generated as the case studies of the modelling language and its modelling environment. The following are these systems.

(1) *United Nations' Security Council*. The organisational structure and the work procedure to pass resolutions were modelled and a formal specification of the system in SLABS was generated. Details of the case study as well as modelling in other agent-oriented modelling notations can be found on AUML's website at the URL:http://www.auml.org/.

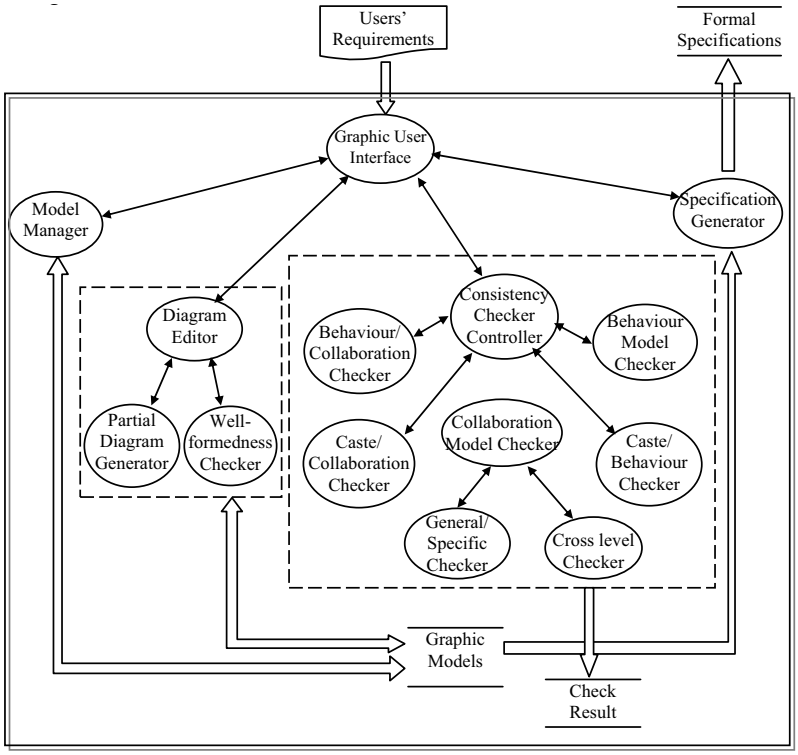


Fig. 21. The Architecture of CAMLE Environment

(2) *Amalthaea*. Amalthaea is an evolutionary multi-agent system developed at MIT's Media Lab to help the users to retrieve information from the Internet [40]. A formal specification of the system was generated from a model in CAMLE.

(3) *University*. This is a partial model of the university organisation. The objective of the case study was not to provide a complete model; instead, it aims at providing illustrative examples to demonstrate the style of modelling in CAMLE. Examples given in this paper were taken from this case study.

(4) *Web Services*. The case study modelled the architecture of web services and an application of web services on online auctions. A formal specification in SLABS of the architecture and application was generated successfully. See [41] for more details.

Before the development of modelling language and the environment, a number of agent-based systems were formally specified in SLABS manually. These systems include Maes' personal assistant Maxims [42, 9], the speech-act theory [43, 44, 9], a simplified communication protocol [25], a distributed

resource allocation algorithm, an ant colony [9], etc. In comparison with the systems in the case studies, these systems are less complicated and hence manageable to write the formal specifications without tool support. The formal specification of Amalthaea system in SLABS was first developed manually, which met much difficulty due to the complexity of the system. It was only completed with the help of informal diagrams to organise the ideas [45]. This diagrammatic notation was later developed into the modelling language CAMLE. Using the modelling language and the automated tool, the system was modelled without too much difficulty and the formal specification was generated successfully. We found that the use of the modelling language was very helpful. It is much more efficient to develop formal specifications through modelling with the help of automated tools than manual approach; especially, the automated consistency checking facility helped to remove syntax errors in the models. In the case studies, we found that the CAMLE language was highly expressive to model information systems' organisational structures, dynamic information processing procedures, individual decision making processes, and so on. The models in CAMLE were easy to understand because they naturally represent the real world systems.

7 Conclusion

In this paper, we proposed a model-driven approach to the development of MAS. It combines graphical models with formal specifications through the employment of automated tools. It is based on a common meta-model of MAS, which is independent of implementation platforms and applicable to all types of agent theories and techniques. A modelling language CAMLE was introduced and an automated modelling environment was reported. We addressed two important issues in model-driven software development of MAS. The first was the consistency problem of the models with multiple view representations. We formally defined consistency constraints as a set of computable rules and implemented them as automated consistency checkers. The second was the automation problem in model-based development. An automated specification generator was designed and implemented to transform graphic models into formal specifications. While graphic models containing a number of diagrams in various views and at different levels of abstraction are more suitable to the representation and understanding of users' requirements involving various stakeholders, modular formal specifications are more suitable to be used by software engineers as the bases for further design and implementation of the specified system. The automated specification generator bridges the gap between them. Case studies show that the approach is effective and efficient for the development of multi-agent systems, especially at requirements analysis and specification and system design stages.

We are further investigating language facilities that directly support efficient implementations of multi-agent systems and the techniques that enable

graphic models and formal specifications to be automatically transformed into executable code.

8 Acknowledgement

The work reported in this paper is partly supported by China High-Technology Programme (863) under the Grant 2002AA116070. The authors are grateful to the colleagues at FIPA's Modelling Technical Committee for the invaluable discussions on various issues related to modelling multi-agent systems via emails and at meetings. The authors would also like to thank the colleagues at the Department of Computing of Oxford Brookes University, especially Prof. David Duce, Mr. David Lightfoot and other members of the Applied Formal Method Research Group and the Computer, Agent and People Research Group.

References

1. P. C. Janca, "Pragmatic application of information agents." BIS Strategic Decisions, Norwell, United States., 1995.
2. P. Sargent, "Back to school for a brand new abc," *The Guardian*, vol. March, p. 28, March 1992.
3. Ovum, "Intelligent agents: The new revolution in software." Ovum Report, London: Ovum Publications., 1994.
4. N. Jennings and M. Wooldridge, *Agent Technology: Foundations, Applications And Markets*. Springer, 1998.
5. N. Jennings, "On agent-based software engineering," *Artificial Intelligence*, vol. 117, pp. 277–296, 2000.
6. F. Brazier, B. Dunin-Keplicz, N. Jennings, and J. Treur, "Desire: Modelling multi-agent systems in a compositional formal framework," *Int. J. of Cooperative Information Systems*, no. 1, pp. 67–94, 1997.
7. L. Shan and H. Zhu, "Camle: A caste-centric agent-oriented modelling language and environment," in *Proc. of SELMAS'04 at ICSE'94*, (Edinburgh, UK), pp. 66–73, IEEE CS, IEE/IEEE, 2004.
8. L. Shan and H. Zhu, "Consistency check in modelling multi-agent systems," in *Proc. of COMPSAC'04*, (Hong Kong), pp. 114–119, IEEE CS, IEEE CS, 2004.
9. H. Zhu, "Slabs: A formal specification language for agent-based systems," *J. of SEKE*, vol. 11, no. 5, pp. 529–558, 2001.
10. H. Zhu, "A formal specification language for agent-oriented software engineering," in *Proc. of AAMAS'2003*, (Melbourne, Australia), pp. 1174–1175, ACM, ACM, 2003.
11. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini, "More dynamic object reclassification: Fickleii," *ACM TOPLAS*, vol. 24, no. 2, pp. 153–191, 2002.
12. H. Zhu and D. Lightfoot, "Caste: A step beyond object orientation," in *Modular Programming Languagesr, Proc. of JMLC'2003* (L. Boszormenyi and P. Schojer, eds.), vol. 2789 of *LNCS*, pp. 59–62, Springer, 2003.

13. J. Odell, H. Parunak, and M. Fleischer, "The role of roles," *Journal of Object Technology*, vol. 2, no. 1, pp. 39–51, 2002.
14. F. Barbier, B. Henderson-Sellers, A. L. Parc, and J.-M. Bruel, "Formalization of the whole-part relationship in the unified modeling language," *IEEE TSE*, vol. 29, no. 5, pp. 459–470, 2003.
15. J. Xu, L. Jin, and H. Zhu, "Tool support of orderly transition from informal to formal descriptions in requirements engineering," in *Proc. of IFIP'96: Advanced IT Tools* (N. Terashima and E. Altman, eds.), pp. 199–206, Chapman and Hall, 1996.
16. L. Kuzniarz, G. Reggio, J. Sourrouille, and Z. Huzar, "Consistency problems in uml-based software development. workshop materials at uml'2002." Research Report. Blekinge Institute of Technology., 2002.
17. C. Nentwich, W. Emmerich, and A. Finkelstein, "Static consistency check for distributed specifications," in *Proc. of 16th Int. Conf. on Automated Software Engineering*, (Coronado Island, CA.), pp. 115–124, 2001.
18. Z. Pap, I. Majzikl, A. Pataricza, and A. Szegi, "Completeness and consistency analysis of uml statechart specifications," in *Proc. of IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pp. 83–90, 2001.
19. P. Andre, A. Romanczuk, and J.-C. Royer, "Check the consistency of uml class diagrams using larch prover," in *Proc. of 3rd Rigorous Object-Oriented Methods Workshop* (T. Clark, ed.), BCS, UK, 2000.
20. R. Paige, J. Ostroff, and P. Brooke, "Check the consistency of collaboration and class diagrams using pvs," in *Proc. of 4th Workshop on Rigorous Object-Oriented Methods*, (London), British Computer Society, 2002.
21. E. Astesiano and G. Reggio, "An attempt at analysing the consistency problems in the uml from a classical algebraic viewpoint," in *Recent Trends in Algebraic Development Techniques, Selected Papers of the 16th Int. Workshop WADT'02*, vol. 2755 of *LNCS*, pp. 56–81, Springer Verlag, 2003.
22. C. Nentwich, W. Emmerich, and A. Finkelstein, "Flexible consistency check," *ACM TOSEM*, vol. 12, no. 1, pp. 28–63, 2003.
23. P. Inverardi, H. Muccini, and P. Pelliccione, "Automated check of architectural models consistency using spin," in *Proc. of 16th IEEE Int. Conf. on Automated Software Engineering*, (San Diego, California), pp. 346–349, 2001.
24. T. Schafer, A. Knapp, and S. Merz, "Model checking uml state machines and collaborations," in *Workshop on Software Model Checking* (S. Stoller and W. Visser, eds.), vol. 55 of *ENTCS*, Elsevier, 2001.
25. H. Zhu, "The role of caste in formal specification of mas," in *Intelligent Agents: Specification, Modeling and Application* (S.-T. Yuan and M. Yokoo, eds.), vol. 2132 of *LNCS*, pp. 1–15, Springer, 2001.
26. J. Gray, J. Zhang, and Y. Lin, "Generic and domain-specific model refactoring using a model transformation engine," in *This book*, Springer, 2005.
27. L. Grunske, L. Geiger, A. Zndorf, N. V. Eetvelde, P. V. Gorp, and D. Varro, "Using graph transformation for practical model driven software engineering," in *This book*, Springer, 2005.
28. R. Silaghi and A. Strohmeier, "Parallax - an aspect-enabled framework for plugin-based mda refinements towards middleware," in *This book*, Springer, 2005.
29. A. Metzger, "A systematic look at model transformations," in *This book*, Springer, 2005.

30. J. Dong, "State, event, time and diagram in system modelling," in *Proc. of ICSE'01*, (Toronto, Canada.), pp. 733–734, IEEE Press, 2001.
31. L. Jin and H. Zhu, "Automatic generation of formal specification from requirements definition," in *Proc. of ICFEM'97*, (Hiroshima, Japan), pp. 243–251, IEEE Computer Society, 1997.
32. A. Orso, L. Baresi, and M. Pezze, "Introducing formal specification methods in industrial practice," in *Proc. of ICSE'97*, (Boston, USA), pp. 56–66, 1997.
33. C. Snook and M. Butler, "Using a graphical design tool for formal specification," in *The 13th Workshop of the Psychology of Programming Interest Group* (G. Kadoda, ed.), (Bournemouth, UK), pp. 311–321, 2001.
34. H. Ledang and J. Souquires, "Integrating uml and b specification techniques," in *Proceedings of Informatik2001*, (Vienna University, Autria), pp. 641–648, 2001.
35. E. T. Hvannberg, "Combining uml and z in a software process," in *Formal Approaches to Agent-Based Systems* (J. Rash, ed.), vol. 1871 of *LNCS*, pp. 47–52, Springer, 2001.
36. I. Hammouda, "A tool infrastructure for model-driven development using aspectual patterns," in *This book*, Springer, 2005.
37. J. Elmqvist and S. Nadjm-Therani, "Intents, upgrades and assurance in model-based development," in *This book*, Springer, 2005.
38. J. Jrjens, "Tool-support for model-driven development of security-critical systems with uml. in: This book.," in *This book*, Springer, 2005.
39. A. Gokhale, G. Trombetti, A. Gokhale, and D. Schmidt, "A model-driven development environment for composing and validating distributed real-time and embedded systems: A case study," in *This book*, Springer, 2005.
40. A. Moukas, "Amalthaea: Information discovery and filtering using a multi-agent evolving ecosystem," *J. of Applied AI*, vol. 11, no. 5, pp. 437–457, 1997.
41. H. Zhu, B. Zhou, X. Mao, L. Shan, and D. Duce, "Agent-oriented formal specification of web services," in *GCC Workshops 2004*, pp. 633–641, 2004.
42. P. Maes, "Agents that reduce work and information overload," *C. ACM*, vol. 37, no. 7, pp. 31–40, 1994.
43. M. Singh, "A semantics for speech acts," *Annals of Mathematics and Artificial Intelligence*, vol. 8, no. II, pp. 47–71, 1993.
44. M. P. Singh, "Agent communication languages: Rethinking the principles," *IEEE Computer*, vol. 31, pp. 40–47, Dec. 1998.
45. H. Zhu, "Formal specification of evolutionary software agents," in *Formal Methods and Software Engineering* (C. George and H. Miao, eds.), vol. 2495 of *LNCS*, pp. 249–261, Springer, 2002.

Index