Modeling Agent-Based Systems with Graph Transformation*

Ralph Depke, Reiko Heckel, and Jochen Malte Klüster

Universität Paderborn, Fachbereich 17
Warburger Str. 100, D-33098 Paderborn
{depke, reiko, jkuester}@uni-paderborn.de

Technical Report No. 213
Dept. of Comp. Sci., University of Paderborn
August 2000

Abstract

The agent paradigm can be seen as an extension of the notion of (active) objects by concepts like autonomy, cooperation, and goal-oriented behavior. Mainstream object-oriented modeling techniques do not account for these agent-specific aspects. Therefore, dedicated techniques for agent-oriented modeling are required which are based on the concepts and notations of object-oriented modeling and extend these in order to support agent-specific concepts.

In this paper, an agent-oriented modeling technique is introduced which is based on UML notation. Graph transformation is used both on the level of modeling in order to capture agent-specific aspects and as the underlying formal semantics of the approach. Concepts of the concurrency theory of graph transformation systems following the double-pushout (DPO) approach are exploited in order to formalize the relation between global requirements specification by means of message sequence charts, and implementation-oriented design models where graph transformation rules specify the agent's local operations.

1 Introduction

The concepts and technologies of agent-based systems become increasingly attractive to the software industry [25]. In order to add new functionality to applications, agents become a part of more traditional software, or they are integrated into legacy systems. Thus, agent-based software development is about to become one aspect of the "normal" software development process.

Today, most software systems are implemented in an object-oriented programming language like C++ or Java, and the analysis and design of such systems is based on object-oriented modeling languages like the UML [30]. Thus, in order to incorporate agent concepts into mainstream software development, an integrated modeling approach for object- and agent-based systems is required.

As modeling concepts, agents and objects have complementary roles: agents act autonomously, driven by their goals and plans, thereby sensing and reacting

*Research supported by the ESPRIT Working Group APPLIGRAPH.
to their environment and cooperating with other agents. Objects encapsulate data structures and operations and provide services to other objects. In this sense, Jennings, et al. [29] state that “There is a fundamental mismatch between the concepts used by object-oriented developers ... and the agent view.” However, the view of objects as mere service providers has its origins in the paradigms of sequential OO programming, and is no longer adequate when considering concurrent languages like Java where objects may have their own thread of control. As a modeling abstraction for concurrent objects, the concept of active object has been established [30] which has much similarity with the agent paradigm.

What is still missing even in active objects is the idea of goal-driven behavior or proactivity of agents and the related concept of autonomy. Autonomy emphasizes the fact that an agent has control about its operations: they are not called from outside like methods but are only invoked by the agent itself in order to reach a certain goal.

Still, object-oriented modeling languages like the UML provide a good basis also for the modeling of agent-based systems. In fact, a couple of approaches in the literature follow this line of reasoning proposing extensions and adoptions of object-oriented modeling for agent-based systems. Iglesias et al. [23], for example, propose a methodology that covers a wide range of agent-oriented software development, from the conceptualization to the analysis and design of systems. Wooldridge et al. [36] present a methodology for agent-oriented analysis and design which is based on the FUSION method and emphasizes the concept of roles. However, both approaches suffer from the usual problem of diagrammatic modeling languages: the lack of precise semantics for individual diagrams and, as a consequence, the lack of consistency rules for diagrams at different stages of the development. Unfortunately, this applies also to the modeling concepts which are relevant for the particular aspects of agent-based systems. In particular, proactivity of agents is often described just by informal text.

It is the aim of this paper to define a modeling technique for agent-based systems building upon the concepts and notations of object-oriented modeling, in particular the UML [30]. Thereby, we pay special attention to the modeling of cooperation, proactivity, and autonomy of agents. We provide our technique with a formal semantics which captures the relation between different kinds of diagrams on the same level of specification as well as the consistency between different phases of the development like analysis and design.

Compared with other approaches to agent-oriented modeling, the main technical idea (which appears first in [26]) is the use of graph transformation (see, e.g., [31, 10, 12] for a recent collection of surveys and [1] for an introductory text) both as a modeling notation and underlying formal model of our technique. As a modeling notation, graph transformation rules in requirement specification and analysis allow us to capture the cooperation among several agents resulting in a joint activity. In the design phase, local graph transformation rules specify the effect of the agents local operations. Here, the non-deterministic choice of the rule to be applied and the location where to apply it provides a convenient model for autonomous operations. As underlying formal model of our approach, typed graph transformation systems [4] provide a natural integration of structural and dynamic aspects as well as elaborate concepts for relating systems on different levels of abstraction.

Next, in Section 2, we shall describe in more detail the properties of agent-based system we are interested in as well as the main concepts of our modeling approach. We also introduce the running example used in this paper. In the following sections 3 to 5 we describe and formalize the three main phases of software modeling (i.e., requirement specification, analysis, and design) within our approach. Section 6 is concerned with the consistency between these phases and Section 7 concludes the paper.
The paper continues previous work on agent-based systems which is documented in [8, 9, 27, 7].

2 Agent-Oriented Modeling

In this section, we outline our approach to agent-oriented modeling. First, we discuss typical aspects of agent-based systems like reactivity, autonomy, proactivity, and cooperation, and describe how these aspects are captured in our approach. Then, we survey the three main phases of system modeling, i.e., requirement specification, analysis, and design and explain how this general pattern is instantiated in our case.

Although it is difficult to find a general (technical) definition of the term agent, some important characteristics of agents can be identified which distinguish them from programs or objects [16]. Reactivity is the capability of an agent to perceive its environment and react to changes. This property can be considered as a prerequisite for purposeful autonomy of agents, and it is already captured within the concept of active objects. In our approach, agents perceive their environment by matching the left-hand sides of their transformation rules against the current state of the system, thus searching for the occurrence of a certain pattern. Then, agents react to an occurrence by the application of the corresponding rule.

Autonomy is a property of agents that manifests in the nondeterminism of its behavior if the system is observed externally. Different to objects agents possess autonomous operations that are not automatically triggered by messages but may be invoked by the agents themselves when a corresponding situation pattern occurs in their environment (see above). If several autonomous operations are applicable in a particular situation, the decision which operation to apply is internal to the agent.

An agent is supposed to be proactive meaning that it tries to reach a certain goal. In our approach, goal-driven behavior is specified by a pay-off function defined on the state space of the system. When choosing between alternatives, an agent aims at maximizing the pay-off. In this way, the intended behavior of complex internal reasoning systems can be abstractly described, and once an implementation is provided, it can be evaluated w.r.t. the pay-off it actually provides. There is a close relationship between the proactivity of an agent and the nondeterminism when observed externally. The decision which and when to apply an autonomous operation is influenced by the agent's drive to optimize the pay-off function.

Cooperation among agents is possible if they have a common goal which is identified at run-time via negotiations. In our approach, global graph transformation rules are used in order to describe the combined effect of negotiations and the resulting joint activities of a group of agents. The communication required is specified by means of UML sequence diagrams.

As a simple but typical example of an agent-based system we describe an online banking application where, in order to enable sophisticated services, customers may be assisted by a personal banking agent (PBA) which offers a range of advanced functionality. In particular, the PBA manages the payment of bills: When a bill is sent to the PBA by the merchant of a shop and the payment of this bill is initiated by the customer, the personal banking agent selects one of the customer's accounts of which the bill is to be payed. This selection takes into account the transaction cost of each account which is considered. (Different costs for transferring money from one account to another may result, for example, from different prices for transactions from one bank institute to another one or from a debit on the account after the money is deducted.) Then, the amount specified in the bill is transferred from the selected account to the destination by account agents responsible for the individual accounts.
The system just described has properties that are characteristic for an agent-based system [16]: The PBA reacts to changes in its environment (like the arrival of a bill) and it modifies this environment through its actions (by paying it). It acts autonomously on behalf of the customer by selecting the account the bill is to be paid from. The agent is goal-oriented in the sense that it aims at selecting the account with the least transaction costs.

We divide the modeling process of agent-based systems in a typical sequence of activities which is already well known from the modeling of object-oriented systems. First, the requirements are specified by informal descriptions of the system’s functionality and by scenarios of important interactions. The analysis of this specification results in a model where the requirements are captured more precisely. Thereafter, in the design model the behavior that has been described globally in the analysis model is expressed by the local behavior of objects and agents.

Within the requirements specification, in Section 3 we follow a use case-driven approach adapted to agent-based systems. Use cases representing the main external functions of the system as well as important internal interactions among agents are refined by typical scenarios which are described by means of global graph transformations and sequence diagrams. The agents’ goals are described informally as goal cases within the use case diagram.

During analysis, in Section 4, the agents and objects as well as their messages, attributes, and links, which are identified in the use cases and scenarios, are specified in an agent class diagram. The scenarios are analyzed in order to derive a more complete specification making explicit the different alternatives in the execution of a use case. The semantics of graph transformations and sequence diagrams thus shifts from optional to mandatory behavior: If the execution reaches a state satisfying a pre-condition (specified by the left-hand side of a graph transformation rule) the further interaction must follow one of the given alternatives. All diagrams of the analysis model are given a formal semantics in terms of typed graphs and graph transformation systems which, in the next step, will be extended to the design model in order to provide a notion of semantic consistency between models.

The design model in Section 5 refines the analysis model in such a way that globally described behavior is mapped to local specifications of the behavior of agents and objects. A refined class diagram introduces additional features, in particular, the signatures of the agents’ autonomous operations. The local execution order of an agent’s operations is determined by a state diagram associated to each agent class. The effect of these operations on the state of the system is described by local graph transformation rules. In the formal presentation of the design model, the states of the state diagram are encoded into the local rules so that the model may be represented as a typed graph transformation system.

The semantic consistency between the global graph transformation rules and sequence diagrams of the requirement specification and analysis model with the state diagrams and local transformation rules of the design model is studied in Section 6. This relation is a subject of ongoing research in the theory of object-oriented modeling. Harel et al. [19], for example, describe a method to synthesize state diagrams from requirements specified by live sequence charts (LSCs) [6]. LSCs provide an extension of message sequence charts (MSCs) [24] with liveness-constraints that allow to distinguish mandatory and optional behavior. Thus, they are able to capture the shift of interpretation between requirement specification and analysis within a single diagram.

Our notion of consistency follows the intuition of Harel [19] as far as the relation between sequence diagrams and state diagrams is concerned. However, we are not dealing with the automatic transformation of MSCs into state diagrams but with the semantic consistency between the two. Moreover, in addition we take into account the specification of agent’s operations by graph transformation rules. Technically,
our approach is based on a comparison of the partial order of send and receive events as specified by a sequence diagram with the causal dependencies within it in a graph transformation sequence. In order to extract these dependencies from a sequence, the concept of graph processes [4] is employed.

3 Requirements Specification

At the beginning of a development, customers and developers have to agree on the requirements a software product has to fulfill. These requirements are collected in a contract which has to be readable by the software developers as well as by customers which are typically not computer scientists. Therefore, at this stage of development a style of specification is appropriate which explains the functional and architectural requirements by means of informal diagrams and examples.

Use case diagrams are designed exactly for this purpose. They provide an abstract view of the system by identifying the main actors using it and the main functions that the system provides to them. In the context of agent-based systems, UML use case diagrams are extended by a special kind of actor (with square heads) representing agents. Goal cases (shown as clouds) are used in order to specify the goals of agents (cf. [23]). The use case diagram of Fig. 1, for example, identifies, besides two kinds of users, the agents PBA and AccountAgent. In this way, additional architectural requirements about the distribution of the system’s functionality over different agents can be expressed. The use cases select account and pay bill that these agents participate in are internal to the system. They would not be shown in a typical UML use case diagram.

The abstract prosaic description given by use cases is illustrated by typical examples, called scenarios, of how the system behaves when a use case is performed. In the methodology of this paper, scenarios are specified in two complementary ways. The overall effect of a use case like select account is described by a pair of instance diagrams as shown in Fig. 2 modeling a before-after scenario of the use case. In the following section, this pair of diagrams shall be formally interpreted.
as an individual graph transformation representing the state change of objects and agents in the system.

In order to specify the communication between actors participating in a use case, UML sequence diagrams are used. In general, several such diagrams are used to capture the behavior for a use case. They may be overlapping and need not be complete. The interaction that is necessary to select an account offering minimal transaction cost would typically be realized by the contract net protocol [15, 35] which describes the negotiation between a manager and a set of potential contractors about the delegation of a task. In terms of our example, a simplified version of this protocol may be informally described as follows.

The Personal Banking Agent solicits proposals from the Account Agent by issuing a call for proposals, which specifies the interest in an account's transaction costs. Account Agent receiving the call for proposals are viewed as potential contractors, and are able to generate proposals to perform the task. Alternatively, account agents may refuse to propose. Once the Personal Banking Agent receives back replies from the Account Agent, it evaluates the proposals and makes its choice of which Account Agent will perform the task. The agent of the selected proposal will be sent an acceptance message, the others will receive a notice of rejection.

A typical scenario for two AccountAgents is depicted in Fig. 3. Other scenarios for our example would include the possibility that no Account Agent makes a proposal or that no proposal is accepted.
4 Analysis

The rather informal and incomplete requirements specified in the first stage of the development process have to be analysed and refined in order to serve as a basis for future design decisions. The analysis is an activity that aims at making the requirements more precise but avoids making implementation decisions. Similar to object-oriented analysis, the refined model constructed from the requirements specification is structured into (sub)models [33], a structural model, a dynamic model and a functional model. The structural model consists of an agent class diagram presenting attributes, operations and messages understood by agents. The dynamic model describes, by means of sequence diagrams the interaction among agents. The functional model specifies the overall effect of these interactions on the state of the system. In addition, our methodology provides a goal model which allows to quantify the achievements of agents by means of pay-off functions.

Structural model An agent class diagram specifies the types of objects and agents, their attributes, associations, and messages. 1 Notationally, we build on class diagrams in UML [30] where agent classes are represented as active classes (with bold borders) that have an extra compartment for messages.

In the agent class diagram in Fig. 4 we have agent classes PBA and AccountAgent and object classes Bill, Account and Proposal. Associations connect the PBAs to the Bills they have to pay and AccountAgents to the Accounts they manage. Another association specifies the AccountAgents used by a PBA. A Bill specifies an amount to be paid and the Account it is to be paid to. A proposal carries a reference to the proposer and another one to the bill it is concerned with, and it specifies the cost for the proposed transaction. The messages are like in the sequence diagram in Fig. 3. They are modeled in the special message compartment of the agent class.

The distinction between diagrams on the type and on the instance level is one

---

1In the case of agents, messages do not automatically result in the execution of methods since agents decide autonomously how and when to react to an incoming message. The autonomous operations of agents are specified in the design model in Section 5
of the most fundamental concepts in object-oriented modeling. In the context of
graph transformation, this distinction is formalized by the notion of typed graphs
[4]. Moreover, in order to represent data-valued attributes of objects and agents,
typed attributed graphs are required.

Attributed graphs [34, 28, 3, 2] are graphs whose vertices or edges are coloured
with elements of abstract data types (like strings or natural numbers), mathemati-
cally represented as algebras over suitable signatures. For the purpose of this paper,
it is enough to consider graphs with attributed vertices. Following [28], we regard
attributes as edges from vertices to attribute values. Fig. 5 on the left shows an
attributed type graph representing a fragment of the class diagram in Fig. 4. Classes
like Bill or AccountAgent are represented as vertex types while associations like pays
are modeled as edge types. Messages like initPayment or propose are shown as vertex
types, too, with edges pointing to recipient and parameters. Data types like int are
modeled as a vertex types of oval shape, and edge types pointing from a class to
a data type represent attributes. E.g., edge type amount from Bill to int represents
the attribute amount: int of class Bill.

An instance graph over this type graph is shown in the same figure on the right. It
represents the instance diagram given by the pre-state of the transformation in Fig. 2
extended by a Bill object with amount = 5000. Formally, an instance graph over a
type graph TG is a graph G equipped with a typing homomorphism g : G → TG,
i.e., a structure preserving function mapping vertices, edges, attributes, and data
values x ∈ G to their types g(x) = t in TG. In this case, we also write x : t ∈ G.

Functional model  In Section 3 we have seen how the overall effect of a use case
can be illustrated by a graph transformation, that is, a pair of graphs modeling a
before-after scenario of the use case. Formally, this scenario can be seen as an
individual test case which has to be demonstrated by the implementation of the
system. However, in order to have a complete view of the use case’s overall effect,
many such graph transformation pairs would be needed. Thus, a mechanism is
required to specify (rather than to enumerate) pairs of graphs.

The theory of graph transformation suggests a rule-based approach to this prob-
lem. A graph transformation rule \( r = L \rightarrow R \) consists of a pair of TG-typed graphs \( L, R \) sharing the same data algebra such that the union \( L \cup R \) is defined. (This ensures that, e.g., edges which appear in both \( L \) and \( R \) are connected to the same vertices in both graphs.) The left-hand side \( L \) represents the preconditions of the rule while the right-hand side \( R \) describes the postconditions. A graph transition [14, 21] from a pre-state \( G \) to a post-state \( H \), denoted by \( r(o) : G \rightharpoonup H \), is given by a subgraph isomorphism \( o : L \cup R \rightarrow G \cup H \), called occurrence, such that

- \( o(L) \subseteq G \) and \( o(R) \subseteq H \), i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state
- \( o(L \setminus R) \subseteq G \setminus H \) and \( o(R \setminus L) \subseteq H \setminus G \), i.e., at least that part of \( G \) is deleted which is matched by elements of \( L \) not belonging to \( R \) and, symmetrically, at least that part of \( H \) is added which is matched by elements new in \( R \).

Notice that, during analysis, rules are considered as incomplete specifications of the transformations to be performed, i.e., additional (unspecified) changes are permitted. This (quite liberal) notion of graph transition shall be strengthened in the design model by the notion of graph transformation \( r(o) : G \Rightarrow H \) which assumes a complete specification of the changes during a step.

Fig. 6 shows three rules specifying the possible effects of the use case select account. Each rule is only concerned with the interaction of one PBA with one of its Account Agents during the execution of the contract net protocol. They specify the three possible results of each binary interaction.

**Dynamic model** The dynamic model complements the functional model by focussing on the communication required to execute a certain protocol. Like in the requirements specification, we use sequence diagrams to model the message flow between agents in the system. However, during analysis, we strengthen the semantics of these diagrams from an existential to a universal interpretation. This is analogous to the shift from individual transformations to universal transformation rules in the functional model. Thus, a sequence diagram associated with a graph transformation rule provides a complete specification of the interactions to be performed when the precondition is met.

In Fig. 7, the sequence diagrams for the banking example are presented. The first diagram models the case that the proposal of the Account Agent is accepted by
the **Personal Banking Agent** and the second one the rejection of the proposal. The third diagram depicts the case that the **Account Agent** does not answer upon a call for proposal. They correspond to the three rules in Fig. 6. A sequence diagram is activated when the precondition of the corresponding rule is met. For the rules in Fig. 6 associated with the sequence diagrams in Fig. 7, the precondition requires that the **Account Agent** is connected with the **Personal Banking Agent** by a **uses** link, and that the latter is activated by a **initPayment** message. Since the precondition is the same in all three cases, if the condition is met, the interaction between the two agents may conform to one of the three sequence diagrams.

Sequence diagrams are part of the UML, but they do not have a formal semantics in the standard. Thus, we will use sequence diagrams with the semantics of message sequence charts (MSCs) [24]. This is straightforward as sequence diagrams have originally evolved from MSCs (and there are even attempts to unify them again [32]). Thus, a sequence diagram is seen as a specification of a partial order over the set of events representing the sending and receiving of messages. These events are visible in the diagram as the sources and targets of message arrows at the vertical instance lines. As defined in the standard [24], all events along an instance line are ordered, as are any pair of events signaling the sending and receiving of the same message.

Consider as an example the left sequence diagram in Fig. 7 and the associated first transformation rule in Fig. 6, jointly modeling a successful negotiation between a **PBA** and an **Account Agent**. Fig. 8 shows the core graph of this interaction, i.e., a graph representing all objects, links, and messages occurring in the rule and the sequence diagram. (We use the representation of messages as vertices introduced in the structural model.) The set of events of this interaction is now given as

\[ \{rec(m_1), \ldots, rec(m_4), snd(m_2), \ldots, snd(m_4)\} \]

with \( snd(m_i) \) and \( rec(m_i) \) representing, respectively, the event of sending and receiving message \( m_i \). The partial order of events is generated by the orderings induced by the two vertical bars

\[ rec(m_1) \leq snd(m_2) \leq rec(m_3) \leq snd(m_4) \text{ and } rec(m_2) \leq snd(m_3) \leq rec(m_4) \]

and the causal dependency between the submission and reception of the same message \( \text{snd}(m_i) \leq \text{rec}(m_i) \) for \( i \in \{2,3,4\} \). The interaction is thus given by the triple \( I = (G, H, E, \leq) \) where \( (E, \leq) \) is the partial order of events over \( C = G \cup H \).
Goal model In order to capture an agent's pro-activity in a more precise way, the textual description of the agent's goals is reformulated with the help of a so-called pay-off function. This function measures the level of "satisfaction" of an agent at the end of an interaction. It can be considered as an abstract control mechanism enforcing, in the case of non-determinism, the choice of the most valuable alternative.

In the interactions associated to the use case select account we have distinguished three different cases represented by the three rules in Fig. 6 and the three sequence diagrams in Fig. 7. The transfer of a message between two agents leads to a constant effort msgcost. For each binary interaction the cost is given as

\[
\begin{align*}
\text{cost}_{\text{Accept}}(a, \text{acc}) &= a.\text{selected}.\text{cost} + 3 \ \text{msgcost} \\
\text{cost}_{\text{Reject}}(a, \text{acc}) &= 3 \ \text{msgcost} \\
\text{cost}_{\text{Ignore}}(a, \text{acc}) &= \text{msgcost}
\end{align*}
\]

if the interaction is finished according to the first, the second, or the third alternative scenario, respectively. The pay-off for a given interaction \( I = (G, H, E, \leq) \) with pre-state \( G \), post state \( H \), and partial order of events \( E, \leq \) is given by the (negative) sum of the local costs

\[
\text{pay-off}(I, a) = -\sum_{a \in \text{Accepted}(I, a)} \text{cost}_{\text{Accept}}(a, \text{acc}) - \sum_{a \in \text{Rejected}(I, a)} \text{cost}_{\text{Reject}}(a, \text{acc}) - \sum_{a \in \text{Ignored}(I, a)} \text{cost}_{\text{Ignore}}(a, \text{acc})
\]

where \( \text{Accepted}(I, a) \), \( \text{Rejected}(I, a) \), and \( \text{Ignored}(I, a) \) denote the sets of account agents \( \text{acc} \) participating in a binary interaction with the given PBA \( a \) according to the respective scenario.

The pay-off can be optimized by the PBA by implementing some strategy, e.g., in order to avoid sending useless messages. At this level of modeling, however, we abstract from the "internal intelligence" of agents.

5 Design

The analysis phase is concerned with developing a model of what the system is supposed to do. The design model elaborates the analysis model concentrating on
the question how the system will function. As a consequence, the focus of models is shifted from a global view on the system during analysis to a local view, thus providing the basis for an implementation.

Similar to the analysis model, our design model consists of a structural model, a dynamic model and a functional model. The structural model consists of a refined agent class diagram presenting attributes, operations and messages understood by agents. The dynamic model describes, by means of state diagrams for each class, the order in which operations of classes can be performed. The functional model specifies the effect of operations using graph transformation rules.

Models constructed during design are refinements of models of the requirements specification and analysis phase. As a consequence, the design model must be syntactically and semantically consistent with the earlier model. Using the notions of graph processes, in this and the next section, these consistency rules are formally expressed.

Structural Model The class diagram of the design phase refines the class diagram of the analysis adding, in particular, the signatures of the agent’s autonomous operations for which an extra compartment is provided. Notice the difference with methods as specified in the method compartment of objects; agent’s operations are autonomous, that is, they are never called by another object or agent but only executed under control of the agent itself (cf. Section 2). As a consequence, we distinguish agent’s messages and operations while in the case of objects, both notions are integrated in the notion of a method.

Formally, the signatures of both operations and methods are represented by a family of sets of operation symbols \( OP = (OP_w)_{w \in TG^+} \), indexed by non-empty lists of types of the type graph \( TG \) representing the class diagram. For \( w = t_0 \ldots t_n \), the
first argument \( v_0 \) is the type representing the class where the method is defined. A pair \((TG, OP)\) of a type graph and a corresponding operation signature is also called a transformation signature.

Consider, for example, the class diagram in Fig. 9. The AccountAgent has two autonomous operations, answerCFP and doTransaction both belonging to the set \( OP_{\text{AccountAgent}} \). These two operations reflect the agent’s autonomy. The agent decides by itself whether and when to answer a call for proposal and to perform a transaction on the account it manages. The concept of an autonomous operation can be seen as an abstraction mechanism which allows to concentrate on the behavior of the agent to the outer world without focusing on the internals.

Besides operations, the design level class diagram may also add other model elements like classes, associations, attributes, and messages. In the banking example, several new associations are introduced, e.g., a sent association between Bill and AccountAgent for expressing that the request for proposals concerning this bill has already been sent to this agent. Syntactic consistency between the class diagrams at the analysis level and at the design level can be formally expressed by a signature morphism. A morphism \( h : (TG, OP) \to (TG', OP') \) between two transformation signatures consists of a graph homomorphism \( h_{TG} \) between the type graphs and a \( TG^+ \)-indexed family of mappings of operation symbols \( (h_w : OP_w \to OP_w')_{w \in TG^+} \) such that \( op : t_1 \ldots t_n \) implies that \( h_w(m) : h_{TG}(t_1) \ldots h_{TG}(t_n) \). Thus, signature morphisms allow the extension, renaming and homomorphic transformation of class diagrams, preserving the structure of classes and associations, and the typing of methods.

The benefit of making explicit the relation between the class diagrams is that rules and diagrams can automatically be translated along these relationships. This is essential for defining the semantic consistency between analysis and design in Section 6. In particular, a homomorphism \( h_{TG} : TG \to TG' \) between type graphs induces a retyping of \( TG \)-typed instance graphs to \( TG' \)-typed ones. For a graph \( TG \)-typed instance graph \( G \) with typing \( g : G \to TG \), the retyping yields the same instance graph typed by \( h_{TG} \circ g : G \to TG' \). Thus the retyped graph has the same elements and connections like the original one, but the types of these elements may be renamed, formerly different types may be identified with each other, and new types may be introduced. The retyping operation extends easily to occurrences and transformations. On this background, in Section 6 we may assume that all diagrams, rules, graphs, transformations, etc., which are dependent on a class diagram or type graph, are retyped appropriately to the class diagram of the design model.

**Dynamic Model** By a state diagram for each agent class, the dynamic model specifies the ordering of operations an agent of this class may perform.

As our agents are autonomous, they do not react to events of their environment immediately with a specific action but rather decide themselves when and how to react. As a consequence, transitions are not labeled with an event and an action but only with the name of the operation. In particular, our usage of statecharts is semantically different from traditional approaches [18]. The notion of a protocol state machine [30] comes closest to our understanding.

The characteristics of agents also show in the non-determinism inherent to this usage of statecharts. Consider, as an example, the statechart for the AccountAgent. From the first state, this agent may either proceed to the proposed state by answering a call for proposal or it may decide not to propose and proceed to the final state. At this level of abstraction, we assume that the agent decides by itself what to do.
Figure 10: Statecharts for agents PBA and AccountAgent

Functional Model  In the functional model, the operations declared in the structural model are specified by typed graph transformation rules. Whereas the dynamic model is concerned with the order of operations the functional model shows how operations change the state of the system. Each operation has a precondition depicted on the left hand side of the graph transformation rule and the result of the operation depicted on the right hand side. An agent’s operations only affect that part of the system state which the agent can access locally. Therefore, objects can only be modified if in the state they are targeted by a path of links originating from the agent. Other agents are influenced by messages sent to them.

In Fig. 11 the operations getBill and sendCFP of the PBA are specified. The first operation triggers the agent to issue requests for proposals for a given bill. If a PBA has not yet sent a call to a particular AccountAgent (expressed by the negative context condition for the sent link) the PBA may use the second rule for issuing the call to this agent.

On reception of a call for proposal message, an AccountAgent may decide to propose to the PBA by sending a Proposal with the costs for the required transaction for paying the bill as specified by the rule answerCFP in Fig. 12. The alternative rule for ignoreProp is not shown. It has the same pre-condition and the only effect of removing the cfp message.

Receiving a proposal, the PBA may either reject it if it has bigger cost than the best proposal received so far or it may record this proposal as its current favorite. The first proposal is recorded when the agent stops sending calls. This is specified in the rules in Fig. 13, 15 and 14. Rejection results in the sending of a reject message. If the new proposal is better than the one selected so far, it is selected as new favorite and the sender of the old favorite proposal is sent a reject message.

Whenever the PBA has received enough proposals it decides to accept the cur-
Figure 11: Graph transformation rules getBill and sendCFP

Figure 12: Graph transformation rule answerCFP

Figure 13: Graph transformation rule stopCFP
a. recordProp(p)

b : Bill
  pays
  for
a : PBA
  propose(acc, p)
  selected
  for
p' : Proposal
  cost = c'

acc : AccountAgent

p : Proposal
  cost = c

Figure 14: Graph transformation rule recordProp

a. rejectProp(p)

b : Bill
  pays
  for
a : PBA
  propose(acc, p)
  selected
  for
p' : Proposal
  cost = c'

acc : AccountAgent

p : Proposal
  cost = c

Figure 15: Graph transformation rule rejectProp
rent best by sending an accept message. Upon reception of this message, the AccountAgent records its proposal as accepted. When rejected, the agent deletes its proposal, as specified in Fig. 17.

With respect to the autonomy of agents, we explicitly allow that two graph transformation rules may have the same left hand side. It is the decision of the agent itself which operation to perform, on the basis to maximizing the pay-off function and possibly on its internal representation. However, in the design model we abstract from these aspects.

The integration of the functional and the dynamic model is achieved by encoding the states of the state diagram into the left- and right-hand sides of the rules. For this purpose we introduce for each state a new vertex type together with an edge type from the state type to the agent type the state diagram is associated with. Thus, in our example, new vertex types waiting, sending and receiving are introduced along with edge types to the vertex type PBA. Moreover, for a rule specifying an operation op labeling a transition from state A to state B in a state diagram, we add an A-vertex in the left-hand side and a B-vertex in the right-hand side, both connected to the self agent of the operation. In this way, a rule can only be applied if the agent is in the state displayed in the precondition, and after application the agent is in the state displayed in the result. In Fig. 18, the corresponding formal presentation of the operation stopCFP is shown.

Due to the encoding of state diagrams into the rules, the design model can be formally represented as a graph transformation system. A graph transformation system $G = (TG, OP, R)$ consists of a type graph $TG$, an operation signature $OP$, and a set of graph transformation rules $R$ equipped with the names and formal parameters of the operations which are specified (cf. the examples in Fig. 11 to Fig. 17. In the following section, this formal presentation shall be used in order to define the semantic consistency of the design model with the analysis model and the requirement specification.

6 Semantic consistency

The syntactic consistency of the class diagrams of the analysis and design model has been expressed by a structure-preserving mapping between the corresponding type graphs and operation signatures. Based on the corresponding translation of diagrams of earlier models to diagrams over the class diagram of the design model, we shall now study the semantic consistency between the models.

In the analysis model, the pre and post conditions as well as the communication associated with a use case have been described by global graph transformation rules and sequence diagrams, respectively. Our design model must conform to these
Figure 17: Graph transformation rules getAccepted and getRejected

Figure 18: Formal presentation of rule stopCFP with encoding of states
specifications: Whenever the pre-condition of a global graph transformation rule is satisfied in a state conforming to the design class diagram, a corresponding sequence of transformations using the local rules of the design model must exist which implement (at least) the same overall effect. Moreover, the sequence must realize the message flow specified in this case by the corresponding sequence diagram. Next, we give an operational semantics to the design model which allows us to make more precise these requirements.

First, a notion of rule application is defined which reflects the intuition of the design model that the transformations on instance graphs are completely specified by the rules. A graph transformation from a pre-state $G$ to a post-state $H$, denoted by $r(o) : G \Rightarrow H$, is given by a subgraph isomorphism $o : L \cup R \rightarrow G \cup H$, called occurrence, such that

- $\alpha(L) \subseteq G$ and $\alpha(R) \subseteq H$, i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state
- $\alpha(L \setminus R) = G \setminus H$ and $\alpha(R \setminus L) = H \setminus G$, i.e., exactly that part of $G$ is deleted which is matched by elements of $L$ not belonging to $R$ and, symmetrically, exactly that part of $H$ is added which is matched by elements new in $R$.

Therefore, a graph transformation $r(o) : G \Rightarrow H$ is a graph transition $r(o) : G \sim H$ where, in addition, $\alpha(L \setminus R) \supseteq G \setminus H$ and $\alpha(R \setminus L) \supseteq H \setminus G$ (cf. Section 4).

A sequential model of the computations in a graph transformation system $G$ is defined by the set of all sequences of transformation steps in $G$. We assume to be given a set $S$ of $TG$-typed graphs as states, such that their union $\mathcal{S} = \cup_{G \in S} G$ is well-defined. A trace $\tau = \tau_1 \ldots \tau_n : G \Rightarrow H$ in $G$ with $G, H \in S$ is a sequence of transformations $\tau_i = \alpha_i \beta_i : G_i \Rightarrow H_i$ with $G_0 = G$, $H_0 = H$ and $G_i, H_i \in \mathcal{S}$ for all $1 \leq i \leq n$, and such that $H_i = G_{i+1}$ for all $1 \leq i \leq n$.

We denote a trace by a sequence of expressions $op(P_1, \ldots, P_n)$ consisting of an operation $op$ identifying the rule with actual parameters $P_1, \ldots, P_n$, specifying the occurrence. The following trace realizes the before-after scenario of Fig. 2

A.getBill(B); A.sendCFP(B, Acc1); A.sendCFP(B, Acc2); Acc2.answerCFP(B); Acc1.answerCFP(B); A.stopCFP(P2); A.recordProp(P1); A.acceptProp(P1); A.rejectProp(P2); Acc2.getRejected(P2); Acc1.getAccepted(P1).

More formally, we say that a trace $\tau : G \Rightarrow H$ realizes a global transformation rule $r = L \rightarrow R$ (of the requirement specification or the analysis model) if there exists an occurrence $o : L \cup R \rightarrow G \cup H$ such that $r(o) : G \sim H$ forms a graph transition. That means, the trace implements at least the same overall effect as specified by the rule $r$.

If a trace is to be seen as the implementation of an interaction (like the negotiation between two agents) we have to ensure that, besides pre- and post-conditions specified by the global rule, it respects also the message order described by the corresponding sequence diagram. In order to be able to compare traces and sequence diagrams we need to understand the different nature of the two representations. First, the sequence diagram is only concerned with a local view of the system, i.e., the agents participating in the specified interaction. A trace, on the other hand, is a global sequence where operations of different interactions are interleaved. Second, the order of operations (events) specified in a sequence diagram is in general partial while a sequence represents a total (linear) order of operations.

In order to resolve this mismatch, we abstract from the concrete ordering of steps in a sequence. By the structure obtained in this way, called graph process [4],

\footnote{This is a set-theoretic presentation of graph transformation according to the DFO approach [13] (see [5] for a recent survey), restricted to injective matching. Like in the categorical definition [13] the resulting graph $H$ is only determined up to isomorphism by the rule $m(o) : L \rightarrow R$ and the occurrence $o|_L : L \rightarrow G$.}
only the data dependencies between the steps are recorded using the occurrences of the rules, while the information about the scheduling of independent steps is lost. Formally, a graph process $\rho$ in a graph transformation system $\mathcal{G}$ is a set of transformation steps in $\mathcal{G}$ such that there exists a linear order on $\rho$ turning the set into a trace.

The causal dependencies between the steps in a process are represented by a partial order which can be derived from a process $\rho$ as follows. Let $C \subseteq \mathcal{S}$ be the union of all pre- and post-graphs of the transformations in $\rho$. Let $r(o) : G \Rightarrow H$ be one arbitrary transformation in $\rho$, $r = L \rightarrow R$ be the corresponding rule, and $e$ be any arc, node, or attribute in $C$. We say that

- $r(o)$ consumes $e$ if $e \in o(L \setminus R)$
- $r(o)$ creates $e$ if $e \in o(R \setminus L)$
- $r(o)$ preserves $e$ if $e \in o(R \cap L)$

The partial order $\leq$ defined on $\rho \cup C$, called causal relation [4], is defined as the transitive and reflexive closure of the relation $<$ where

- $e < r_1(o_1)$ if $r_1(o_1)$ consumes $e$
- $r_1(o_1) < e$ if $r_1(o_1)$ creates $e$
- $r_1(o_1) < r_2(o_2)$ if $r_1(o_1)$ creates $e$ and $r_2(o_2)$ preserves $e$, or $r_1(o_1)$ preserves $e$ and $r_2(o_2)$ consumes $e$

The causal order can be used in order to recover from a process $\rho$ the initial and the final graphs of a trace, formed by the sets of minimal and maximal elements of $C$, respectively. This justifies to denote a process as $\rho/C : G \Rightarrow H$ where graphs $C, G$ and $H$ are determined by the set of transformations $\rho$.

Based on the partial order induced by a process, we can now formalize the idea of realization of a sequence diagram. Consider a trace $\tau : G \Rightarrow H$ which realizes a global transformation rule $r = L \rightarrow R$ via an occurrence $o : L \cup R \rightarrow G \cup H$. We say that $\tau$ realizes the sequence diagram associated with $r$ if there exists a mapping $rl : E \rightarrow \rho$ from the set of events of the sequence diagram to the set of steps forming the process such that $rl$ preserves the partial order of events, that is, $rl(e_1) \leq rl(e_2)$ for all $e_1 \leq e_2 \in E$. We also require that the mapping $rl$ is compatible with the occurrence $o$ in the sense that $o$ extends from $L \cup R$ to the core graph $C$ of the diagram (that is, to the messages) such that $rl(rec(m))$ consumes $o(m)$ and $rl(snd(m))$ creates $o(m)$ for all messages $m$.

The following mapping $rl : E \rightarrow \rho$ shows that the trace above realizes the first sequence diagram of Fig. 7. It associates events $snd(m)$ and $rec(m)$ to steps in the trace.

$$
rec(m_1) \mapsto A.getBill(B) \\
snd(m_2) \mapsto A.sndCFP(B, Acc1) \\
rec(m_2), snd(m_3) \mapsto Acc1.answerProp(B) \\
rec(m_3) \mapsto A.recordProp(P1) \\
snd(m_4) \mapsto A.acceptProp(P1) \\
rec(m_4) \mapsto Acc1.getAccepted(P1)
$$

This mapping is indeed a realization because the partial order of events specified in the sequence diagram is preserved. For example, the dependency $snd(m_2) \leq rec(m_2)$ is preserved because $rl(snd(m_2)) = A.sndCFP(B, Acc1)$ creates the cfp message and $rl(rec(m_2)) = Acc1.answerProp(B)$ consumes this message. Thus, according to the definition of the partial order above, $A.sndCFP(B, Acc1) \leq Acc1.answerProp(B)$. The dependency $rec(m_1) \leq snd(m_2)$ is realized in the trace by means of the encoding of the sending state into the right-hand side of
rule $A.\text{getBill}(B)$ and the left-hand side of rule $A.\text{sndCFP}(B, \text{Accl})$ (cf. rule $\text{stopCFP}$ in its visual and formal presentation in Fig. 13 and 18, respectively).

According to the different interpretation of rules and sequence diagrams in requirement specification and analysis, we define two notions of consistency. In the first case, we assume to be given typical scenarios corresponding for each use case. Thus we require that each of these scenarios has at least one realization in the design model. In the second case, a set of alternative rules and sequence diagrams delivers a universal specification of the transformation and communication associated with a use case. Therefore, we ask that from each state graph in the design model satisfying the pre-condition of the use case, there exists a realization of at least one of the alternatives.

- For each rule $r$ (and corresponding sequence diagram) in the requirement specification, the design model is consistent with $r$ if there exists a trace $\tau : G \Rightarrow H$ which realizes $r$.

- For each set $R$ of alternative rules $r_i = L \Rightarrow R_i$ (and corresponding sequence diagrams) in the analysis model and for every occurrence $o_L : L \Rightarrow G$ in a graph $G$ conforming to the design class diagram, there exists a rule $r_i$ in $R$ and a trace $\tau : G \Rightarrow H$ which realizes $r_i$ such that the occurrence $o : L \cup R \Rightarrow G \cup H$ of the corresponding transition is an extension of $o_L$ (that is, $o_L = o|_L$).

The partial-order based approach to relate sequence diagrams and global system traces is not the only possible solution to this problem. Other approaches rely on the construction of a global system automaton [19] which is obtained by computing the product of the state diagrams of the design model. Traces of this automaton can then be compared with interleavings of the partial order obtained from the sequence diagram or MSC.

In Fig. 19, the product automaton of the state diagrams of agent classes PBA and AccountAgent from Fig. 10 is constructed. The example shows that the automata-theoretic approach does not take into account data dependencies defined by the functional model because the effect of operations on the state of the system is not specified in the state diagram. For example, the $\text{acc.\_answerCFP}$ rule of an AccountAgent can only be applied if there has already been issued a cfp message from the PBA. As a consequence, many transitions in the global system automaton are never possible if the preconditions of the operations are taken into account. In Fig. 19, only solid arcs describe possible transitions of the product automaton. Dotted arcs are transitions that result from the product of the state diagrams but they are impossible because data is missing in the source state of the transition in order to apply a graph transformation rule. For example, the operation (rule) $\text{acc.\_answerCFP}$ can not be applied in the state (1.5) because no cfp message has been issued before.

Another advantage of the partial order-based approach is the fact that a single partial order represents a whole equivalence class of global traces, i.e., all its interleavings. Thus, the representation of system traces by means of partial orders is also a matter of efficiency. This fact is exploited, e.g., in the model checker SPIN where partial order reduction is used in order to minimize the set of relevant intermediate states for the verification [22]. A disadvantage of the global system automaton relies on the fact that all possible traces are represented. Thus, all interleavings are explicitly visible as alternative paths within the automaton. In Fig. 19 for example, two possible interleavings of the operations $\text{acc.\_answerCFP}$ and $\text{a\_stopCFP}$ are given by the paths between the states (2,5) and (3,6) via states (2,6) and (3,5).
7 Conclusion

In this paper, we have presented an approach to agent-oriented modeling based on UML notation and concepts of typed graph transformation systems [4]. Extending the notion of active object from object-oriented modeling, specific support is provided for characteristic aspects of agent-based systems like autonomy, goal-driven behavior, and cooperation of agents.

The theory of graph transformation also provides the mathematical background for the formalization of the approach. In particular, the theory of graph processes [4] and concepts of refinement of graph transformation systems [14, 20, 17] are used in order to formalize the consistency between requirement specification, analysis and design in agent-oriented modeling.

References


REFERENCES


REFERENCES


