Object-oriented specification of coordinated collaboration

G. Engels\textsuperscript{a}, L.P.J. Groenewegen\textsuperscript{a}, and G. Kappel\textsuperscript{b}

\textsuperscript{a}Dept. of Computer Science, Leiden University, P.O. Box 9512, NL-2300 RA Leiden, The Netherlands, \{engels|luuk\}@wi.leidenuniv.nl

\textsuperscript{b}Dept. of Computer Science, University of Linz, Altenbergerstr. 69, A-4040 Linz, Austria, gerti@ifs.uni-linz.ac.at

Abstract
Object-oriented specification mechanisms have become famous for modelling structure and behaviour together in terms of autonomous objects communicating via message passing. However, whereas most object-oriented specification methods are strong in modelling the local behaviour of single objects, they fall short on modelling the coordinated behaviour and collaboration of several objects together. The aim of this paper is to contribute to fill this gap. The paper reports on concepts, language constructs, and experiences with three collaboration formalisms in the area of object-oriented specifications.

Keywords
object-oriented specification, collaboration, synchronization, cooperation contract

1 INTRODUCTION
Due to the high availability of interconnected computers, software systems are no longer restricted to support the work of individuals. Now it becomes more and more common that software systems support the work of groups of somehow interrelated persons, too. Concrete examples of such collaborative environments can be found in all areas of industrial, administrative, and social life.
After some years of experience with prototypical systems, it has become obvious that computer-supported collaborative systems — like any other complex software system — require a clear specification of structure and behaviour prior to any realization. Only this can ensure qualities like correctness, reliability, and extensibility.

During the last decade, object-oriented specification mechanisms have become famous for modelling structure and behaviour together in terms of autonomous objects communicating via message passing. Thus, they seem to provide appropriate concepts to specify in particular the above mentioned computer-supported collaborative systems. However, whereas most object-oriented specification methods are strong in modelling the local behaviour of single objects, they fall short on modelling the coordinated behaviour and collaboration of several objects together. The aim of this paper is to clarify this and to contribute to fill this gap. The paper reports on concepts, language constructs, and experiences with three collaboration formalisms in the area of object-oriented specifications. The collaboration formalisms have been successfully applied to specify the collaboration within different types of engineering and business process support systems.

The paper is structured as follows. In the next section we briefly review object-oriented specification mechanisms and identify the requirements for collaboration formalisms in the realm of object-oriented specifications. In Section 3 three collaboration formalisms are presented, namely enforced usage collaboration in OBD (Object/Behavior Diagrams), collaboration constraints in SOCCA (Specification of Coordinated and Cooperative Activities), and cooperation contracts supporting multi-polymorphic behaviour specifications. All three formalisms are introduced on the basis of the well-known example problem on modelling the behaviour of an Automated Teller Machine (ATM). The paper concludes with a discussion to which extent the requirements have been met by the presented collaboration formalisms and reports on future work.

2 DIMENSIONS OF COLLABORATION SPECIFICATIONS

2.1 Object-Oriented Specifications Revisited

It is commonly accepted that there are three fundamental views on a system, which are also supported by most object-oriented specification methods, such as OMT* (Rumbaugh 1991), Booch’s method (Booch 1994), and OOSA (Embley 1992). These are the data view (to whom does something happen?), the behavioural view (when does something happen?), and the process view or functional view (what happens?).

The data view is specified in terms of the object model defining the static structure, i.e., the attributes of objects, the relationships to other objects, and the operations of the involved objects. Objects having a similar static structure are defined by an object class, which is depicted by a class diagram. Figure 1 represents the class diagram of the object class ATM showing the class name, its attributes, and its operations (for the sake of conciseness the relationships have been omitted from the class diagram).

The behavioural view is specified in terms of the dynamic model defining the allowed
sequences of operation invocations on the objects of an object class. The dynamic model is depicted by a state transition diagram (STD). Figure 2 represents the state transition diagram of the object class ATM. The names annotating the edges, i.e., the state transitions, refer to the operation names of the class diagram (cf. figure 1). A state transition happens if the corresponding operation is explicitly invoked on some object being in the source state of the transition. A state transition without corresponding operation happens if the respective object reaches the source state.

The functional view is specified in terms of the functional model defining the implementation of an operation, i.e., which other operations are invoked when the corresponding operation gets executed. There are various kinds of notations used for depicting the functional model, such as pseudo code, and data flow diagrams (DFDs) in OMT. However, DFDs have been developed in the realm of structured development supporting functional decomposition, and, thus, are difficult to map to object-oriented concepts in the static model and dynamic model. Hence, we suggest to use a STD for the implementation specification of an operation as well. Figure 3 represents the functional model STDs of the operation $\text{fetchMoney}$ of the object class $\text{USER}$ and of the operation $\text{enterPIN}$ of the object class $\text{ATM}$. The names annotating the edges, i.e., the state transitions, refer to the operations being invoked on an instance of the denoted object class.

In addition to these three views, some object-oriented methods support the specification of an interaction model in terms of an interaction diagram depicting the uses relationships between object classes, i.e., which object class uses which operations from which other object class. Due to space limitations we refer to the literature for details (Booch 1994. Embley 1992).
2.2 Requirements for Collaboration Specifications

The previous subsection gave a short overview of main concepts and used diagram techniques of existing object-oriented modelling approaches. A critical review of the field reveals that there is a great agreement on used concepts for modelling the static structure of objects and their possible interrelations. On the other side, concepts and techniques for an appropriate modelling of the allowed behaviour of single objects and, in particular, of the collaborating behaviour of complete object systems is still under discussion. Furthermore, there seems to be no (commercially used) approach up to now, which offers sophisticated means to model all structural and behavioural aspects of a system with collaborating objects and which has, in addition, a well-defined underlying semantics. The main reason for this mismatch between structural and behavioural modelling concepts is the still restricted experience with modelling of collaborative behavioural aspects. Up to now, this has prevented a natural agreement on modelling primitives for behaviour modelling. Nevertheless, a variety of collaboration formalisms has been proposed by others as well as by the authors. As a first step towards structuring the field of proposed behaviour modelling primitives, this paper summarizes and compares three different proposals for particularly modelling collaborative behaviour. All three proposals were originally developed by the authors. The comparison will be done along four dimensions, namely:

1. usage constraints referring to the restricted or even enforced usage of operations of an object class.
2. number of threads of control referring to sequential versus parallel operation execution.
3. degree of synchronization referring to synchronous versus asynchronous collaboration.
4. number of collaborators referring to the number of objects acting together to fulfill a certain task.
After introducing the three formalisms, which are known as enforced usage collaboration, collaboration constraints, and cooperation contracts, they will be compared along these four dimensions in turn.

3 DIFFERENT APPROACHES

3.1 Enforced usage collaboration

Enforced usage collaboration has been originally introduced in the object-oriented specification method Object/Behavior Diagrams (OBD) (Kappel 1991). An enforced usage collaboration expresses the fact that a certain operation of an object class must be and may only be invoked by a certain operation of another object class. Thus, the concept of enforced usage collaboration refines the above mentioned interaction diagrams. Interaction diagrams define, which operations are imported from other classes and, therefore, are visible and usable within the importing class. The notion of usability can be further refined into the notions of statically and dynamically used. Statically used means that the imported operation occurs somewhere in the implementation specification of an operation of the importing class. During an actual execution, a statically used operation may also be dynamically used, i.e., invoked. The notion of enforced usage, as it was introduced in the OBD method, goes (at least) one step further. It allows to restrict the visibility of exported operations to a specific importing class, and enforces that the exported operation is invoked during the execution of a specific operation of this importing class.

This concept of restricted and enforced usage of operations is needed to model real world events in which several objects are involved and change their state. The reason is that the object-oriented paradigm requires that every operation is executed on exactly one object and, furthermore, that the implementation of the operation does not directly change the state of another object. To stimulate such a state change, an appropriate message must be sent to that object, which corresponds to invoking an operation in the functional model STD. The concept of enforced usage allows to enforce this invocation of an operation of another object during the execution of an operation on the invoking object.

An enforced usage collaboration is represented in a dynamic model STD by annotating the restricted usable operation, i.e., state transition, with the invoking object class and the invoking operation. As an example of an enforced usage collaboration, consider the functional model STD of the operation enterPIN of ATM in figure 3 and selected parts of the dynamic model STD of the object class BANK in figure 4. The operation checkPIN of BANK is annotated with [ATM.enterPIN]. The constraint expresses that a PIN code may only be checked if the request comes from an ATM object and the corresponding PIN has been entered. It also expresses that if a PIN is entered into some ATM, the PIN has to be checked. (Note, that in the original article (Kappel 1991) Petri nets are used to depict the dynamic model of objects, and thus, a slightly different notation is used for enforced usage collaboration.)
3.2 Collaboration Constraints

The diagram types introduced and discussed so far, provide appropriate means to answer the following four questions within an object-oriented model:

- who’, i.e., which objects are involved and how is the structure of these objects?
- when’, i.e., what are allowed sequences of operation calls to a certain object?
- what’, i.e., what is the functionality of an operation?
- who with whom?, i.e., which operations are imported from which other class, thus enabling a collaboration with objects of the other class.

At a first glance, answering these questions seems to be sufficient and appropriate to model a real-world situation. But, a more detailed look at systems where objects are collaborating in different ways shows that the following questions cannot be answered and appropriately modelled on the basis of answers to the above questions:

- is an object sending a message to another object (operation invocation) obliged to wait until the receiving object has really received this message, or is the sending object allowed to proceed immediately with other activities?, i.e., is the collaboration synchronous or asynchronous?
- is it allowed that objects are concurrently active, or is an object obliged to postpone its activities until another currently active object has finished its execution?, i.e., is the objects’ execution parallel or sequential?

The former question refers to the degree of synchronization known from concurrent languages, i.e., synchronous versus asynchronous collaboration. The latter question refers to the number of threads of control, i.e., how many concurrently active execution paths are possible, equally known from concurrent languages. Figure 5 illustrates those different kinds of collaborations for the running example. It employs so-called event trace diagrams where the collaboration between single objects is described on the instance level (Rumbaugh 1991).

Up to now, the collaboration between a USER object and an ATM object has been (partially) modelled by the functional model STD of USER.fetchMoney in figure 3 and the dynamic model STD of figure 2. In particular, the dynamic model STD of ATM
Figure 5  Four different collaboration variants.

requires that the enterAmount operation has to be received before PINok reaches the ATM. This means that in the top left collaboration situation (synchronous, sequential) the following — perhaps intuitively more logical — order of calls: (1) enterPIN, (2) checkPIN, (3) PINok, (4) enterAmount, is not allowed. On the other hand, it remains open which of the four visualized collaboration situations from figure 5 are allowed. It is, for instance. not expressed by the current model that the USER is not allowed to proceed with other activities before the ATM object reacts to enterPIN.

Thus, the diagram types introduced so far do not allow to specify the number of threads of control, i.e., sequential or parallel behaviour of objects, on one side, and the degree of synchronization, i.e., synchronous or (restricted) asynchronous behaviour in case of operation calls, on the other side. But, these differences in behaviour modelling are important, particularly in case of systems where the kind of collaboration is of great relevance.

This is, for instance, the case in software development processes, where human agents (e.g., project managers, developers) and non-human agents (e.g., tools) are collaborating in developing and maintaining different types of documents (e.g., programs, documentations). In this context, the specification language SOCCA (Specification of Coordinated and Cooperative Activities) has been developed (Engels 94). SOCCA is related to OMT. but extends it considerably by addressing the collaboration more precisely. This is done by a sublanguage of SOCCA, now called Paradigm, which specifies collaborations between STDs by using the notions of manager (process), employee (process), subprocess and trap (Steen 87).

Each dynamic model STD of a class becomes a manager process of the following employee processes: all functional model STDs of operations of this class as well as all functional model STDs, in which an operation of this class is called. In order to act as
a manager process, the dynamic model STD is enriched with collaboration constraints restricting the behaviour of collaborating objects. This restricted behaviour is described by subprocesses, which are subdiagrams of the functional model STDs. A trap of a subprocess is a subset of the subprocess’ states that cannot be left as long as this subprocess is the current behaviour restriction. For instance for the synchronous, parallel collaboration situation, in figure 6 two subprocesses \( P_{sp}-1 \) and \( P_{sp}-2 \) are shown, which are subSTDs of the left-hand STD of figure 3. Trap examples are visualized by a shaded area around the trap’s state(s). A USER object, currently being restricted to subprocess \( P_{sp}-1 \), reaches, for instance, state s3 after sending the message \( \text{enterPIN} \) to an ATM object. As s3 is the only state of trap \( t_{sp}-1 \), the USER object has to wait until its allowed behaviour is changed to subprocess \( P_{sp}-2 \). An analogous division into subprocesses is done for the functional model STD of ATM.\( \text{enterPIN} \) (cf. figure 7, the subprocesses \( P_{sp}-4 \) and \( P_{sp}-5 \)).

Each state within the manager process is additionally labelled by the subprocesses the collaborating objects are restricted to. For instance, for the synchronous, parallel collaboration situation, within state s1 the USER object has to be in subprocess \( P_{sp}-1 \) and the ATM object has to be in subprocess \( P_{sp}-4 \). In the case that the USER object is in trap \( t_{sp}-1 \) (having called \( \text{enterPIN} \)) and the ATM object is in trap \( t_{sp}-4 \) (waiting for execution of \( \text{enterPIN} \)), the transition from state s1 to state s2 within the manager process is done. Within the new state s2, the subprocess \( P_{sp}-2 \) is prescribed for the USER object, thus allowing to proceed with calling \( \text{enterAmount} \) or \( \text{cancel} \). The prescribed subprocess for the ATM object is \( P_{sp}-5 \), thus allowing to start the execution of \( \text{enterPIN} \).
Concerning the call of \texttt{enterAmount} by a \texttt{USER} object, it is not required that the \texttt{ATM} object has already finished the execution of \texttt{enterPIN}. This is due to the fact that trap $t_{sp^{-5}}$ comprises nearly all states of subprocess $P_{sp^{-5}}$. Thus, this part of the model describes a synchronous and parallel behaviour.

Another kind of collaboration can be modelled by choosing the subprocesses and the sizes of traps differently. For instance, figure 9 shows a different subprocess $P_{ap^{-1}}$ and traps, replacing the two subprocesses $P_{sp^{-1}}$ and $P_{sp^{-2}}$ from figure 6. In this case, a restricted asynchronous behaviour is modelled, as the larger trap $t_{ap^{-1}}$ allows the \texttt{USER} object to proceed with \texttt{enterAmount} without waiting for an acceptance of the \texttt{enterPIN} call by the \texttt{ATM} object. Finally, by restricting the trap $t_{sp^{-5}}$ to state $s2$, a sequential behaviour can be modelled.

To conclude with, SOCCA supports the specification of various numbers of threads of control and various degrees of synchronization.

### 3.3 Cooperation Contracts

It is claimed that object-oriented specifications concentrating related behaviour around object classes are easy to extend and easy to maintain. This is true, if the behaviour to be modelled depends on only one type of object. However, this is false, for example, in the \texttt{ATM} case if the calculation of the withdrawal rate depends on the type of service station the withdrawal is ordered at, and on the type of person ordering it. Thus, even in
object-oriented models it is necessary that the number of objects acting together to fulfill a certain task is varying.

In this subsection, we suggest a solution by extending the traditional message passing paradigm. We introduce the concept of cooperative message handling. A cooperative message is a message which is sent jointly to several objects. Note: such a message is not received at distinct points of time by each object separately, but once by all objects together. All addressed objects act together in answering the cooperative message.

An example of a cooperative message is the message withdrawalRate sent jointly to a user \( u \) and a service station \( s \).

Cooperative messages are defined with cooperation contracts which are established between several object classes. Cooperation contracts have been originally introduced in (Schrefl 1991) and extended in (Schrefl 1996). A cooperation contract fulfills a similar purpose in behavior modelling as the symmetric, undirected relationship does in data modelling. Symmetric, undirected relationships are used in data modelling to express relations between several objects. They make a structural dependence between several objects explicit in a natural way and avoid to spread such a dependence over binary directed relationships. Similarly, cooperation contracts make a behavioural interaction between instances of several object classes explicit. They avoid that one of several objects has to be singled out arbitrarily as target of a message the answer to which depends on an interaction of several objects. Note, that the receivers of a cooperative message need not be connected by a structural relationship. In our example, one can ask the total amount to withdraw if a certain user \( u \) asks some service station \( s \) to withdraw some money from some account. It is not necessary that all possible pairs of users and service stations, i.e., all possible withdrawal orders, are stored.

A cooperation contract is established between several object classes, which are referred to as partner classes of the cooperation contract. It defines a set of cooperative operations, the interfaces of which are defined the same way as they are for operations of a single object class.

The example in figure 10 depicts the cooperation contract MONEY_WITHDRAWAL between object class USER and object class SERVICE_STATION with two cooperative operations withdrawalRate and totalAmount. The operation withdrawalRate computes the costs for some user \( u \) withdrawing money from some service station \( s \). The operation totalAmount computes the total amount to be withdrawn based on the amount required from \( u \) and the withdrawal rate. Since cooperation contracts support inheritance and overriding along the partner classes it is straightforward to override these cooperative operations which change their behavior depending on the partner classes and their respective subclasses. For example, figure 10 depicts the following two class hierarchies. First, the object class USER, its subclasses CLERK and CLIENT, and the subclass VIP of CLIENT. And second, the object class SERVICE_STATION with its subclasses ATM and COUNTER. Since a “normal” client has to pay the full withdrawal rate of 15, no own cooperation contract is needed between the object class CLIENT and the various kinds of service stations. However, since VIPs do not have to pay any withdrawal rate independent of the kind of service station, an own cooperation contract MW.VIP.SERVICE.STAT is specified overriding the withdrawal rate of the original cooperation contract to 0. Clerks of a bank do not have to pay the full withdrawal rate such as “normal” clients, but pay different withdrawal rates at different kinds of service stations (cf. cooperation contracts MW.CLERK.COUNTER and
\textbf{Figure 10} Cooperation contracts.

\texttt{MW\_CLERK\_ATM}). Note, that the implementation of the operation totalAmount is the same for all kinds of USERs and all kinds of SERVICE\_STATIONs, thus, it is inherited unchanged along the inheritance hierarchy of the cooperation contracts depicted in figure 10.

Similar to the concept of enforced usage collaboration, cooperation contracts refine interaction diagrams in the sense that not only the import of operations from a single object class but also the import of operations from cooperation contracts may be specified. With the latter, it is possible to specify a 1 to n collaboration where one object invokes a cooperative operation on n other objects together. Thus, cooperation contracts support the specification of a varying number of collaborators working intimately together to fulfill a certain task.

4 DISCUSSION AND FUTURE WORK

The three approaches introduced above can now be compared along the four dimensions of collaboration specification, which are degree of usage restriction, number of threads of control, degree of synchronization, and number of collaborators. These dimensions are depicted in figure 11 in terms of a Kiviat graph. Note, that the various values of one dimension are not interpreted as being mutual exclusive but that the “higher” value includes the “lower” one. For example, if a system supports asynchronous collaboration (depicted as “2” in the Kiviat graph) than it supports synchronous collaboration, too.

Analyzing the Kiviat graph shows the strength and weaknesses of the discussed collaboration formalisms. OBD supports restricted and enforced usage collaboration and a synchronous sequential collaboration between two objects. SOCCA supports parallel as well as asynchronous collaboration between two objects. In addition, it is also possible to enforce some collaboration in the sense that a certain operation must be and may only be invoked by some other operation. The advantage of OBD is that it provides a concise
formalism to express collaboration enforcement, whereas the advantage of SOCCA is that it has higher expressive power — collaboration enforcement is just one mechanism which can be specified in terms of subprocesses, traps, and manager processes. Cooperation contracts have been introduced to investigate 1 : n collaboration in the sense that one object sends a cooperative message to several other objects together, leaving apart the other dimensions of collaboration specification.

Based on this discussion it becomes obvious that the goal should be to have a single collaboration formalism supporting the various dimensions together. We are currently investigating such a single formalism also taking related work in this area, such as timing constraints (Booch 1994) and transverse activities (Kristensen 1996), into account.

REFERENCES


Pierre Cointe), Springer, LNCS.

**BIOGRAPHY**

Gregor Engels is full professor for software engineering and information systems at Leiden University since 1991. He graduated in computer science in 1980 at the University of Dortmund (Germany), and received his Ph.D. in computer science in 1986 at the University of Osnabrück (Germany). He has worked in the field of software development environments and graph grammar based specification languages. His current research interests are object-oriented concepts and software process modelling languages. He participates with his group in several European research and development projects in the fields of software process modelling, requirements engineering, environmental information systems, and graph grammar based specification languages.