Chapter 1

Coordinated Collaboration of Objects

Gregor Engels, Luuk Groenewegen, Gerti Kappel

Keywords:
coordination, collaboration, synchronisation mechanisms, UML, SOCCA

Abstract:
The computer has evolved from the purpose of pure number crunching to supporting the coordinated collaboration between human and/or artificial beings to reach a certain goal. Object-oriented modelling techniques based on the central notions of object interaction and object collaboration should provide the semantic expressivity to model such coordinated collaboration. Based on an investigation of the object-oriented modelling standard UML, however, the weaknesses of the existing modelling concepts are revealed. SOCCA is presented instead, which is an object-oriented specification language supporting the arbitrarily fine-grained synchronisation of processes, i.e., active objects. Based on several examples of the EU rental car system the expressive power of SOCCA is discussed and compared to related approaches.

1.1 Introduction

In modern software systems one can observe a tendency towards supporting ongoing work of concurrently active individuals. These individuals might be human beings, or artificial objects like software components. They are somehow interrelated and interconnected and they communicate with each other to share and exchange information.

This tendency can for instance be observed in the growing importance as well as gradual shift in meaning of the notion of user-friendliness. User-friendliness
is no longer restricted to warnings in case of input errors or to extensive help facilities. Nowadays, it comprises informing a user about possible consequences of an action taken and of possible next steps after such an action. This information also reflects the state of a system as a whole. So possible effects of other users actions are taken into account, too. At this point the original notion of user-friendliness is gradually shifting towards the notion of process support. Typical examples of systems reflecting the above development are workflow systems as successors to office information systems, and software process environments as successors to CASE tools.

This shift of focus in modern software systems is mirrored by current developments in the area of modelling techniques. These modelling techniques should support a more accurate specification of relevant interrelations, interconnections, communications and mutual influences between all objects involved. The objects involved comprise human beings and other problem domain objects as well as (artificial) software objects. The problem of choosing the “right” modelling techniques and their appropriate conceptual models is still an open research question. Object-oriented approaches with the conceptual model of object societies consisting of interacting objects are a promising candidate to fulfill the above mentioned requirements.

In more detail, object-oriented modelling techniques are based on the notion of an object. Objects are instances of a corresponding class. They have a unique identifier, and a hidden internal state. They communicate with each other by sending messages. Messages have a name, and optionally a number of argument types and a result type.

Each class provides a public interface, which comprises messages which are understood by the instances of this class. Messages are implemented by operations (also called methods). They realize the behaviour of an object. Two kinds of (sub-)behaviour of an object can be distinguished. First, operations realize the local behaviour of an object, i.e., how the internal state of an object is modified during execution. Second, operations realize the global behaviour, i.e., how different objects collaborate with each other by sending and receiving messages. In order to reach a certain goal, objects have to collaborate in a structured and coordinated way. Thus, the coordinated collaboration of objects in a society thereof is a major research challenge [16].

Currently available object-oriented modelling techniques deploy diagrammatic languages to model these aspects of an object society, i.e., the structure of objects, their local behaviour as well as their coordinated collaboration. All techniques agree on the use of class diagrams to model the structure of objects. The modelling of object structures has a long tradition in the field of conceptual modelling of databases. Based on the original work on Entity-Relationship models [3], standard modelling structures have been identified and are nowadays used in object-oriented class diagrams. In contrast to this, standard modelling structures for behaviour modelling do not yet exist. Different diagrammatic techniques are used to model behavioural aspects, such as state diagrams, statecharts, data flow diagrams, Petri nets, message sequence charts and collaboration diagrams. All of them, or rather variants of them, have been
1.2. Classification of Collaboration

Object-oriented modelling techniques use object societies as conceptual models, where a set of objects exist and where objects communicate with each other by message passing. A closer look at these conceptual models reveals differences concerning the allowed number of threads of control concurrently active within the object society and concerning the various kinds of message passing between the objects.

Concerning the number of threads of control purely sequential systems are possible, where at a certain point of time exactly one object is active. Thus, at a certain point of time there exist exactly one thread of control within the whole society. Conversely, all objects may be concurrently active, where each object has its own thread of control. The possibility of concurrently active objects might be even further extended if intra-object concurrency is supported, i.e., an object is involved in several processes, i.e., threads of control, at the same time.

Concerning the kind of message passing, different variants can be identified, too. Message passing in general means that an object, the sender, sends a message to another object, the receiver. In most approaches, two variants are distinguished. The message passing is called synchronous, if the sender...
is blocked after having sent the message to the receiver until the receiver has finished the activity asked by the message and has given an appropriate reaction. This kind of behaviour of two collaborating objects can be found as procedure call mechanism in sequential programming languages. Conversely, message passing is called asynchronous, if the sender is not blocked after having sent the message to the receiver. In this case, the sender continues with other activities without explicitly waiting for a reaction to the message sent. It is possible that the receiver accepts the message sometime in the future, and that it might or might not react to this message by sending an answer to the former sender.

This brief and informal explanation of the differences between synchronous and asynchronous message passing corresponds to explanations as they can be found in text books on object-oriented modelling techniques. This superficial style of explanation as well as the limited expressive power of these two kinds of synchronisation, however, are not sufficient for a software analyst to model realistic situations in an appropriate way. For example, the restriction to choose only between synchronous and asynchronous collaboration does not allow to model situations like the one, where a sender sends a message to a receiver asynchronously, continues with other activities, but would like to get an answer at some point in the future before it continues with a certain activity.

In order to improve this situation, possible interaction patterns between two objects are studied in detail in the remaining part of this section. In particular, similar to the framework of interaction rule patterns in [14] a distinction is made between the start and the end of an interaction between a sender and a receiver. The resulting classification of interaction patterns will be used in Section 1.3 to investigate whether and how UML, as typical representative of widely used object-oriented modelling techniques, supports the modelling of such a behaviour. In section 1.4, the object-oriented modelling language SOCCA is presented, which provides explicit means to model all variants of synchronous and asynchronous behaviour.

In the following it is assumed that in the underlying object model each object has its own thread of control. This implies that two objects which are involved in an interaction may be concurrently active.

For a detailed study of the start of an interaction we distinguish the following constituents. Each object has an associated set of states. In a state, an object has three actions to choose from, namely an object may send a message, it may receive a message, and it may perform some local computations. By choosing an action, the object enters another state or reenters the same state. Any allowed state/action sequence is called behaviour of an object. The execution of some behaviour of an object is called the execution of the object in the following. In addition, each object has an associated buffer, where incoming messages are queued. The execution of a sender may be blocked after sending some message to a receiver, if the continuation is depending on whether the receiver has reached a certain state. Figure 1.1 illustrates the different situations at the start of an interaction. Possible states of an object are drawn as circles, and time proceeds from top to bottom. Thus, the sequence of states from top to
1.2. CLASSIFICATION OF COLLABORATION

Figure 1.1: Patterns for the start of an interaction

Bottom displays the effect of the execution of one behaviour of an object solely in terms of states entered. The sending of a message is shown by an arrow, and a blocking situation is indicated by a horizontal bar.

Four different situations at the start of an interaction can be distinguished:

1. The start of an interaction is called synchronous, if the sending object is blocked immediately after having sent the message until the receiving object accepts that message by reading it from the input buffer (cf. Figure 1.1.a). (Note, for the purpose of this paper we do not further distinguish different blocking intervals, such as infinite, time bound, and balking [1]).

2. The start of an interaction is called asynchronous, if the sending object continues with other actions independently from whether the receiver accepts the message immediately or sometime in the future or not at all (cf. Figure 1.1.b).

3. The start of an interaction is called future synchronous, if the sender is allowed to continue with other actions after having sent the message, but
Figure 1.2: Patterns for the end of an interaction

is blocked at some state until the receiver accepts that message by reading it from the input buffer (cf. Figure 1.1.c).

4. The start of an interaction is called restricted asynchronous, if the sender is allowed to continue with other activities after having sent the message, but is restricted to a certain (sub)behaviour until the receiver accepts the message sent (cf. Figure 1.1.d). The restricted behaviour is indicated by a loop of states in front of the horizontal blocking bar.

Analogously, the end of an interaction can be studied in detail. As before, we distinguish four different situations, but now at the end of an interaction, i.e., when the receiver has finished the execution of the operation invoked by the message sent.

1. The end of an interaction is called synchronous, if the sender is blocked from the start of the invoked operation’s execution on, originally initiated by the message sent, until the end of that execution (cf. Figure 1.2.a).

2. The end of an interaction is called asynchronous, if the end of the execution
of the invoked operation has no effect on the sender (cf. Figure 1.2.b). In fact, the sender becomes not aware of the end.

3. The end of an interaction is called *future synchronous*, if the sender of a message is allowed to continue with various actions, but is blocked at some state to wait until the receiver of the message informs the sender that the execution of the invoked operation has been finished (cf. Figure 1.2.c).

4. The end of an interaction is called *restricted asynchronous*, if the sender of a message is restricted to a certain (sub)behaviour until the receiver of the message informs the sender that the execution of the invoked operation has been finished (cf. Figure 1.2.d).

The combinations of one out of the four patterns for the start of an interaction with one out of the four patterns for the end of an interaction sum up into 16 different interaction patterns for the interaction between the sender and the receiver of a message. In Figure 1.3, we depict three different combinations as examples. We will use these examples in the forthcoming sections to explain

Figure 1.3: Three examples of interaction patterns
how such coordinated collaborations are modelled with UMLs collaboration diagrams and with SOCCA, respectively. The three examples are

- synchronous start and synchronous end (cf. Figure 1.3.a).
- asynchronous start and asynchronous end (cf. Figure 1.3.b), and
- asynchronous start and future synchronous end (cf. Figure 1.3.c).

The first example models what is usually called a procedure call mechanism. The second example models what is commonly referred to as asynchronous communication. In this case, the sender does not mind that it has sent a message. It does neither wait for the start nor the end of the receiver’s execution of the operation invoked by the message. The third example shows that an asynchronous start does not necessarily imply for the sender to forget about having sent a message. In this case, the sender is not forced to wait until the receiver accepts the message sent. But it is forced to wait at a certain state until the receiver has finished the execution of the operation invoked by the message sent.

1.3 UML

The Unified Modeling Language UML has been approved by the OMG on November 17, 1997 as the standard notation for object-oriented analysis and design [18]. Quickly, industry and research alike have adopted UML for their every day work, although parts of UML are still in their infancy [13]. After providing a short intro into UML, we will concentrate on UMLs mechanisms for collaboration modelling.

The basic concepts and description techniques of UML support the modelling of the three interrelated perspectives of an information system, which are the structural perspective, the behavioural perspective, and the process perspective [17]. Although the object-oriented approach clearly emphasizes structural and behavioural modelling and diminishes the importance of process modelling compared to traditional approaches it is commonly accepted that the functionality of the system at hand has to be specified at first place. For this purpose, UML introduces use cases as integral part of object-oriented analysis. Based on use case diagrams the required object classes with their static and dynamic features are identified and depicted in class diagrams. The necessary interaction between objects is represented in terms of sequence diagrams and collaboration diagrams in turn. Use cases are further formalised in terms of sequence diagrams and activity diagrams alike. In contrast to the global system behaviour, which is described with the before mentioned diagrams, local object behaviour is depicted in terms of statechart diagrams. To model both application independent and application dependent constraints, UML provides the OCL, the object constraint language.

To introduce the object classes involved in our running example taken from the “EU-Rent Car Rentals Case Study” [24], Figure 1.4 depicts the corresponding class diagram fragment in UML notation. We see the most important classes
involved in car renting according to the above mentioned case study. Classes are depicted in terms of rectangles showing their names, attributes (if any), and operations in three different compartments. As we are going to discuss some collaboration details based on an example built around a so-called walk-in request, we see customers as well as desks in that class diagram fragment. Furthermore, we see some other problem domain objects as far as they are being administered by such a desk when handling a walk-in request, namely the general customer information concerning malicious customer behaviour, the list of currently available cars, and the cars themselves. Associations between these classes have been indicated by linking the respective rectangles, but only to the extent they are interesting for such a walk-in request.

### 1.3.1 Collaboration Modelling with UML

For interaction respectively collaboration modelling, UML provides two kinds of diagrams, namely sequence diagrams and collaboration diagrams. Following the standard document [18], both diagrams are equal concerning their semantic expressiveness, but they stress two different views. Whereas the sequence diagram emphasises the temporal perspective of interaction, the collaboration diagram emphasises the various kinds of relationships between the interacting objects. For the purpose of this paper, we concentrate on the latter. We will explain the intrinsics of collaboration diagrams by means of the collaboration examples introduced in Section 1.2.

Figure 1.5 depicts a collaboration diagram describing the case of synchronous start and synchronous end with respect to the message \textit{WalkInRequest} sent from some customer to some desk (cf. the operation \textit{WalklnRequest} in Figure 1.4). A collaboration diagram consists of prototypical objects depicted as rectangles with the (optional) name of the object and the name of its class prefixed with a colon shown inside the rectangle. The links between the objects are annotated with the messages sent between the objects. The direction of a message sent is given by an annotated arrow. Two kinds of arrows allow to distinguish between synchronous and asynchronous collaboration. The “normal” arrow head
stands for synchronous collaboration, the half arrow head for asynchronous one. Messages are numbered concerning their execution order. Nested numbering refers to the implementation of the message with the corresponding number. Conditional execution of a message may be represented in two ways. First, by so-called guards in square brackets, and second, by preceding message numbers indicating that the execution of the corresponding messages has to be finished before the execution of the message under investigation may start.

Back to our example in Figure 1.5. Synchronous start and synchronous end of some collaboration actually comes down to the well-known procedure call mechanism, in the sense that as soon as the message asking for an activity has been sent the sender waits not only until the start but even until the end of the activity. In Figure 1.5 this is reflected by using the synchronous arrow belonging to the message WalkInRequest numbered 1 and directed from object thisCustomer to an object of type Desk.

![Collaboration diagram - synchronous start and synchronous end of WalkInRequest](image-url)

Figure 1.5: Collaboration diagram - synchronous start and synchronous end of WalkInRequest
As we can conclude from that diagram, after sending the message WalkInRequest thisCustomer has to wait until the end of the activity asked for by that message. This is synchronous start and synchronous end. In particular, the diagram specifies that the implementation of WalkInRequest consists of the messages 1.1, 1.2, and 1.3, which are sent by some object of type Desk. The result of WalkInRequest is returned as parameter answer. It is later on used as precondition for sending the messages PickUpCar (numbered 3) and ReturnCar (numbered 4). The message GetInsuranceInfo is independent of the result of WalkInRequest. But due to the synchronous behaviour modelling in this case, the message GetInsuranceInfo will only be sent after the implementation of WalkInRequest has been completely finished.

![Collaboration diagram](image)

Figure 1.6: Collaboration diagram - asynchronous start and future synchronous end of WalkInRequest

The second collaboration diagram given in Figure 1.6 visualises the asynchronous start and future synchronous end case, again with respect to the message WalkInRequest. To this end, message 1 has an asynchronous arrow directed from...
object thisCustomer to an object of type Desk. This indicates that thisCustomer's next message GetInsuranceInfo numbered 2 may be sent without any waiting for activity WalkInRequest to start or end. Thus, this modelling comes much closer to the real life situation, where a customer may already proceed to ask for information about insurances, while it is checked whether a car of a certain type is available for the desired period.

The message GetInsuranceInfo is sent asynchronously, too. Therefore, in principle the following message, numbered 3, could be sent without any waiting for start or end of the execution of GetInsuranceInfo. But in this case the sending of message PickUpCar depends on the result of the previously sent message WalkInRequest. As the message WalkInRequest has been sent asynchronously, the resulting answer is not contained as return parameter in the parameter list of WalkInRequest, but explicitly sent in a separate message (numbered 1.4) from the Desk object to the object thisCustomer. The waiting of message numbered 3 for the result of message numbered 1.4 is a typical example for the future synchronous end case. In the collaboration diagram, this is indicated by prefixing the message number 3 with "1.4 / " . So, here is the delayed wait for the end of activity WalkInRequest, whereas there is no wait at all for its start.

Summarising, UML allows to model other interaction patterns next to plain synchronous and asynchronous communication. However, there is no explicit support for those patterns, and they have to be specified from scratch.

## 1.4 SOCCA

In this section, we will discuss how SOCCA models coordinated collaboration. The next subsection gives a brief introduction to SOCCA. Subsection 1.4.2 presents the synchronous start and synchronous end case of collaboration, while subsection 1.4.3 covers the asynchronous start and future synchronous end case. From these fragment examples together one can get a good impression on how all other cases may be covered. Section 1.4.4 discusses pros and cons of using UML versus SOCCA.

### 1.4.1 Introduction to SOCCA

SOCCA is an object-oriented specification approach [2, 6]. SOCCA is an eclectic formalism, whose constituent formalisms are somewhat comparable to the formalisms used in OMT [19] and UML [18]. Thus, there is a substantial relationship between SOCCA and UML. In its specifications, SOCCA concentrates on four perspectives, namely data, behaviour, functionality, and communication. How this is done, will be discussed successively.

As in UML, SOCCA's data perspective is covered by a class diagram. This description not only contains the classes with their operations and attributes but also the various relationships between them, such as associations, inheritance relationships, and aggregations. Similar to UML, SOCCA supports also the uses relationship for specifying which of a class' public visible operations are
1.4. SOCQA

actually imported in the functionality of another class (cf. the class diagram fragment in Figure 1.7). In contrast to UML, the annotated imported operations not only may be used in the implementation of the functionality of the client object class, but they really have to be used. The class diagram fragment in Figure 1.7 shows the classes of our running example an EU-Rent branch office as well as the uses relationships between these classes.

Also similar to UML, SOCCA’s behaviour perspective is covered by a state transition diagram (STD) for each class (called statechart diagram in UML). Such a STD specifies the allowed sequences of operations which get executed when answering messages sent to the corresponding object. As SOCCA is meant to express multi-threading, handling a message in the order of such a sequence has to be understood as starting the execution of the corresponding operation, but not as completely executing that operation.

Different to UML, the functionality perspective in SOCCA is covered by STDs, too, one separate STD for each operation of a class. These STDs are being called the internal STDs, whereas the STDs used in the behaviour perspective are being called external. An external STD specifies what is usually called the visible behaviour of a class. An internal STD specifies the hidden implementation of an operation. In other words, the functionality of an operation is described in terms of its hidden behaviour.

To reflect the differences in perspective, the two different kinds of STDs obey two different sets of rules for labeling the transitions.

An external STD has as labels for its transitions the operation names of the corresponding class. As an example see Figure 1.8, where the external visible behaviour of class Customer and of class Desk from Figure 1.7 has been visualised in terms of two external STDs. Note that unlabeled transitions may occur, too. In the case of Desk, the unlabeled transition reflects the possibility that a WalkInRequest results in a negative answer, so WalkInRequest then is not to be followed by GetInsuranceInfo and PickUpCar. State labels (or state names) are indicated within the state symbol, which is a circle. They are used for discriminating purposes, only. A starting state is indicated by means of an incoming directed edge without a source node.

Figure 1.7: Class diagram fragment depicting the uses relationship
The labeling rules for internal STDs are as follows. An internal STD, corresponding to an operation \texttt{OpA} belonging to a class \texttt{A}, has its first transition, the one leaving the starting state, labeled with \texttt{`act\_OpA'}. This actually reflects the activation of operation \texttt{OpA} inside whatever instance of class \texttt{A}. Furthermore, as transition label \texttt{`call B\_OpB'} may occur, provided that \texttt{OpB} is an operation belonging to class \texttt{B}, and this \texttt{OpB} is actually imported in \texttt{A}. This means that \texttt{OpB} is to be found in the list of operations annotating the uses relationship in the class diagram pointing from class \texttt{A} to class \texttt{B}. In addition, transitions may be unlabeled, or may have a comment. Comments are indicated between \texttt{<} and \texttt{>.

Figure 1.9 depicts the internal STDs for the operation \texttt{RentACar} of class \texttt{Customer}, and for the operation \texttt{WalklnRequest} of class \texttt{Desk}. Note that \texttt{WalklnRequest}, \texttt{GetInsuranceInfo}, \texttt{PickUpCar} and \texttt{ReturnCar} may be called from within \texttt{RentACar}, i.e., may occur as transition labels in \texttt{RentACar}'s internal STD (cf. Figure 1.7 for the corresponding uses relationship). They also must be called, i.e., must occur as such labels, since \texttt{RentACar} is the only operation of \texttt{Customer}. Similarly, \texttt{Check of CustomerInfo} as well as \texttt{Availability} and \texttt{KeepAvailAnswer} of \texttt{Desk} are called from within \texttt{WalklnRequest}, as they occur in one of the lists of the corresponding uses relationships. The operations \texttt{Add}, \texttt{CarTaken}, \texttt{StatusCheck} and \texttt{CarReturned} are not called here, although they also occur in the related lists of used operations. This is allowed, provided that each of them is at least called from within \texttt{GetInsuranceInfo}, \texttt{PickUpCar} or \texttt{ReturnCar} (due to space limitations. we have omitted the internal STDs of the latter three operations). The internal STDs of \texttt{RentACar} and \texttt{WalklnRequest} give a more detailed description of the functionality compared to the above given UML collaboration diagrams, as in addition the cases of negative answers are treated, too.

The communication perspective in SOCCA is covered by a third formalism of SOCCA known as Paradigm, which specifies collaborations between STDs by using the notions of manager (process), employee (process), subprocess, trap and collaboration constraints\textsuperscript{1} [23].

Managers as well as employees are just STDs, but with a certain role in coordinating their collaboration. Subprocesses and traps are some special restrictions on employees.

\textsuperscript{1}Collaboration constraints have been called state-action-interpreter in earlier publications on Paradigm.
Each external STD of a class B is considered a manager of the following employees: all internal STDs of operations belonging to B (the callees), as well as all internal STDs from which a call to one or more of B's operations occurs (the callers), i.e., containing a transition labeled with 'call B OpB', where OpB is an operation of B. Such a group of STDs, consisting of a manager and its employees, actually constitutes a group of collaborating classes. Such a group or team is gathered around the class with the manager role, i.e., which has the external STD that is the manager. In order to act as a manager, the original external STD is extended in two directions.

The first direction is a refinement. States and transitions are added for representing possible results, even intermediate results of the actually called export operations. In [2] this refinement has been called the communicative view of the external STD. The new transitions might be labeled with some <comment>. These labels have no formal meaning at all, they only serve as some intuitive clarification.

The second direction of extending a manager is a completely new labeling of all its states and transitions, included those added in the refinement. These labels express the actual collaboration constraints. Collaboration constraints restrict both the visible behaviour of the manager and the hidden behaviour of its employees, i.e., they restrict (mostly hidden) behaviour of collaborating objects in one collaboration team.
The restricted behaviour of each employee is specified by means of subprocesses, which are subdiagrams of the internal STD being that employee. An (eventual) effect of such restricted behaviour is represented by the notion of a trap of a subprocess, which is a subset of the subprocess’ states such that this subset cannot be left as long as that subprocess is the current behaviour restriction. The manager actually restricts the behaviour of its employees, by telling them what is their current subprocess. On the other hand, the manager’s behaviour is restricted by its employees. As long as a certain trap has not been entered, the manager may not change the current subprocess into a next one. It is the collaboration constraint that expresses these dependencies. It does so by relating through labeling manager states to subprocesses, and by relating manager transitions to traps. A manager, by being in a state, exactly indicates the subprocesses related to that state, as the current ones for its employee - therefore a manager is said to prescribe the subprocesses. Conversely, one or more employees, by being in a trap, exactly indicate the transitions related to the combination of those traps, as currently permitted - therefore the manager is said to obey the traps. Furthermore, to start consistently, the first prescribed subprocess for an employee is such that the employee’s starting state belongs to that subprocess’ set of states. In addition, a trap not only is a trap of a
certain subprocess. A trap is also a trap to a next subprocess, which means the following. The states of that trap also belong to the set of states of this next subprocess. So a manager, prescribing this next subprocess after obeying that trap, does not force its employee to continue its behaviour in a discontinuous manner. The states to which this employee has been restricted most recently also occur in the currently prescribed next subprocess.

Figure 1.10 illustrates by an abstract example the above explanation of how to model coordinated collaboration in SOCCA. Here, the (extended) external STD of class B acts as the manager process for coordinating the call of operation \( \text{opB1} \) within the internal STD of operation \( \text{opA1} \) with the actual start of the execution of \( \text{opB1} \). In particular, the start state of the manager labelled by \( (s2, s3) \) enforces that the execution of operation \( \text{opA1} \) (being in subprocess \( s2 \)) has to stop after the call of operation \( B.\text{opB1} \) (trap \( t2 \)), and to wait until the called object of class B has reached the starting situation (trap \( t3 \)) in its current subprocess \( s3 \). Trap examples are visualised by a shaded area around the trap’s state(s). Only if both traps have been reached, the calling object of class A returns into its starting situation (subprocess \( s1 \)) and the called object of class B may start to execute operation \( \text{opB1} \) (subprocess \( s4 \)). Please note that Figure 1.10 gives only a partial description, as far as it is needed to explain the concepts of SOCCA.

In the next two subsections we give two concrete examples of SOCCA specifications, i.e., for the purely synchronous case as well as for the asynchronous start and future synchronous end case.

1.4.2 Modelling of synchronous collaboration in SOCCA

The subprocesses and traps relevant for the coordinated collaboration in the synchronous start and synchronous end case are described in Figure 1.11. The figure presents the subprocesses of the internal STDs for caller RentACar and for callee WalkInRequest. Although the employees for GetInsuranceInfo, PickUpCar and for ReturnCar belong to the same team of collaborating classes, their subprocesses and traps have been omitted for the sake of brevity. For similar reasons of brevity we have omitted the transition labeling as it is present in the complete internal STDs (cf. Figure 1.9). First of all, we see the subprocesses Pss-1 and Pss-2 of WalkInRequest. Pss-1 represents the phase where the actual operation execution has been stopped and a new execution has not yet been started. Pss-2 represents the phase where the call to the operation is being executed. As we see, Pss-2 has three different traps, reflecting the three possible answers to the request: ‘no’ (because of malicious customer behaviour), ‘no, but’ (perhaps a car from a different category is available), ‘OK’ (a car from the asked for category is available). Pss-1 has one trap only, simply indicating the end of that phase, so the operation can be restarted if needed. The trap Tss-1 of subprocess Pss-1 is a trap to subprocess Pss-2. The three traps Tss-2, Tss-3 and Tss-4 of subprocess Pss-2 are traps to Pss-1.

Second we see the subprocesses Pss-5, Pss-6, Pss-7, Pss-8, Pss-9, and Pss-10 of caller RentACar. Pss-8, Pss-9, and Pss-10 are mainly relevant with respect to
Figure 1.11: Subprocesses and traps of caller RentACar and callee WalkInRequest (synchronous communication)

the callees GetInsuranceInfo, PickUpCar and ReturnCar, which will be left out of our discussion. Pss-5 represents the phase of RentACar where a call for WalkInRequest is permitted as far as Desk is concerned. Its trap indicates the actual call. As this is a case of synchronous start, the trap consists of exactly one state, only allowing for waiting until at least the asked for operation execution has been started. Pss-6 reflects the phase of the behaviour after WalkInRequest’s execution where the answer is a plain ‘no’. Pss-7 reflects a similar phase, but now the answer is a ‘no, but’, as a car from a different category might be an option. Pss-6 as well as Pss-7 have as large traps as possible, as immediately after receiving the answer a new walk-in request is to be permitted as soon as possible, although in case of a plain ‘no’ this means an explicit restart of the operation RentACar. Pss-8 reflects the phase after the answer ‘OK’ has been received. Pss-9, and Pss-10, resp., reflect an even later phase, after call GetInsuranceInfo and call PickUpCar have been called. Here trap Tss-5 is trap to Pss-6 and also to Pss-7 and to Pss-8. The traps Tss-6, Tss-7, and Tss-10 are traps to Pss-5. Trap Tss-8 is trap to Pss-9, and trap Tss-9 is a trap to Pss-10.

Although the above explanation already suggests that this is indeed a synchronous start and synchronous end case with respect to calling WalkInRequest, the formal indication for it comes from the actual collaboration coordination by the manager. This coordination is enforced through the collaboration visualised in Figure 1.12. The state-related subprocesses are indicated as state labels, and the transition-related traps are indicated as transition labels. This labeling is a
1.4. SOCRA

Figure 1.12: Manager Desk of employees Customer, RentACar and Desk.WalkInRequest (synchronous communication)

rather complicated matter because of the generally large number of employees belonging to the manager’s collaboration group. Thus, some organisation has been added to the labeling. Labels are being represented according to a fixed ordering of the employees. Here we choose the upmost label as referring to employee WalkInRequest, and the undermost label as referring to employee RentACar. As the other employees have not been discussed, we also omit the labels referring to their subprocesses and traps. Moreover, where the trap does not matter, this is being indicated through “—”. Formally this refers to a so-called trivial trap, consisting of all states of the (current) subprocess. Usually a trivial trap of any subprocess is trap to only the subprocess itself. In the figures we have omitted these trivial traps.

According to the above formulation of the collaboration constraints, Desk in its state d1 either indicates Pss-5 as the current subprocess to RentACar of Customer or Pss-10. In addition, Desk prescribes Pss-1 to WalkInRequest of Desk. As RentACar’s starting state only occurs in Pss-5 (and not in Pss-10), this is the first subprocess indicated by Desk for RentACar. Only after both RentACar has entered its trap Tss-5 through the transition ‘call Desk.WalkInRequest’, and WalkInRequest has entered its trap Tss-1, Desk transits to state d2, thereby prescribing Pss-2 to WalkInRequest, and still keeping RentACar in its old subprocess in the trap. Thus, WalkInRequest actually starts a new execution and RentACar continues to wait for the result. As RentACar could do nothing else then waiting since it entered its one-state trap, the collaboration case certainly is of the type synchronous start. Subsequently, only after WalkInRequest has entered one of its traps Tss-2, Tss-3 or Tss-4, Desk prescribes Pss-1 as the next subprocess to WalkInRequest and simultaneously either Pss-6 or Pss-7 or Pss-8 to RentACar, depending on what is Desk’s next state (d3, d4, or d5). This in turn depends on the exact trap entered in Pss-2. The caller RentACar is indeed kept in its single state trap Tss-5 until WalkInRequest is finished, i.e., has entered either trap Tss-2, Tss-3 or Tss-4. So we
see here that the collaboration case is of type synchronous end. Moreover, the transition \(<\text{no}>\) indeed corresponds to malicious customer information found (Tss-2), the transition \(<\text{no, but}>\) indeed corresponds to the required category is not available (Tss-3), and \(<\text{OK}>\) indeed corresponds to the request can be honoured (Tss-4).

This finishes the discussion of the SOCCA modelling of purely synchronous collaboration.

### 1.4.3 Modelling of asynchronous collaboration in SOCCA

Mainly by choosing the size of traps differently, different types of collaboration cases can be modelled, although sometimes also subprocesses and even STDs have to be adapted accordingly. For instance, consider the collaboration case of asynchronous start and future synchronous end, again with respect to calling of \(\text{WalkInRequest}\).

Figure 1.13 presents a slightly adapted internal STD for \(\text{RentACar}\) and for \(\text{WalkInRequest}\), together with their subprocesses and traps with respect to manager \(\text{Desk}\). The small difference with \(\text{RentACar}\) from Figure 1.9 is that after the call to \(\text{GetInsuranceInfo}\) the effect of a refusal to the preceding call of \(\text{WalkInRequest}\) must be taken into account, too. So, there is an extra transition labeled with \(<\text{no}>\). The small difference in the subprocesses mainly consists of combining Pss-5 and Pss-8 into one new subprocess Paf-5, as after the call of \(\text{WalkInRequest}\) the possibility to continue asynchronously is to be offered. This continuation may only be the calling of \(\text{GetInsuranceInfo}\), as for all other possible continuations the result of \(\text{WalkInRequest}\) has to be available. Moreover, in order to be able to continue after that call to \(\text{GetInsuranceInfo}\) the result of \(\text{WalkInRequest}\) has to be available anyway. The trap representing the call to \(\text{WalkInRequest}\) is Taf-5. It consists of two states, one more than the corresponding trap Tss-5 from the previous collaboration case. This very accurately expresses how the behaviour of \(\text{RentACar}\) is indeed to continue asynchronously after the call to \(\text{WalkInRequest}\). The small trap Taf-5 inside trap Taf-5 is the analogue to trap Tss-8 from Figure 1.11. It serves to indicate that Paf-9 might be prescribed as next subprocess, thus signaling that \(\text{PickUpCar}\) has been called. Moreover, trap Taf-8 still allows for Paf-6 or Paf-7 as next subprocess, as even after the call to \(\text{GetInsuranceInfo}\) the result of the preceding call to \(\text{WalkInRequest}\) might be a \(<\text{no}>\) (Paf-6), or a \(<\text{no, but}>\) (Paf-7). This is exactly captured by the requirement that Taf-8 is trap to Paf-6, to Paf-7, and to Paf-9. This is also captured by the manager \(\text{Desk}\) (cf. Figure 1.14), where the external STD of Desk is refined by additional states \(d_2, d_5, d_6, d_2', d_5', d_6'\) to model the different interleaved behaviours of finishing \(\text{WalkInRequest}\) and executing \(\text{GetInsuranceInfo}\). Please note that the modelling does not show the internal behaviour of \(\text{GetInsuranceInfo}\). The overall specification would become even more detailed, if this would be considered, too.

While \(\text{WalkInRequest}\) and \(\text{GetInsuranceInfo}\) might be executed in parallel, \(\text{PickUpCar}\) has to wait for the end of \(\text{WalkInRequest}\). This is the case in state \(d_6'\), where the two subprocesses Paf-1 and Paf-9 have been reached.
Figure 1.13: Internal STD of RentACar and WalkInRequest together with their subprocesses and traps (asynchronous communication)
This finishes the discussion of the asynchronous start and future synchronous end case of collaboration. Both examples have shown how any degree of asynchronous collaboration between several participants can be modelled by choosing the right subprocesses and traps of internal STDs, as well as by refining and extending external STDs appropriately by additional intermediate states and collaboration constraints.

1.4.4 Discussion

In the following, SOCCA and UML are compared concerning their supported collaboration mechanisms.

Looking back at SOCCA, how it establishes the coordination of the various collaborations, we can conclude the following. First, the collaborations are organized in groups. Each collaboration group is centered around one external state transition diagram, the visible behaviour of an object class. Such a group consists of the external STD itself, of the callers of its operations and of the callees, its own operations. Second, the collaborations are explicitly described in terms of effects on the behaviour of the members of such a collaboration group. That is to say, not only the immediate effect in terms of one step of the behaviour, but also the longer run effect in terms of the complete future behaviour gets specified, until the next coordinating action is taken. In that respect the SOCCA approach has two strong advantages. First, SOCCA is
explicitly prescriptive in its collaboration specification instead of declarative. This means that every degree of (a)synchronism is accurately expressed as which part of the full behaviour is permitted at which moment. Second, SOCCA structures the collaboration coordination by not only describing the immediate, short term effect of whatever communication, but also by describing the long term effect through the notions of subprocess and trap.

If one tries to express this information in some collaboration diagram, the notation becomes complicated. For example, in Figure 1.5 the message PickUpCar numbered 3 should have some precondition like `[answer OK] OR (answer ‘not yet received’)`. Moreover, if in the same collaboration diagram one also wants to express the iteration modelled as returning from state r4 to state r2 in Figure 1.9, it is not at all clear how to do this. Similar considerations are found in the literature concerning UML’s interaction diagrams comprising sequence diagrams and collaboration diagrams [8, page 112]: “They are good at showing collaborations among objects; they are not so good at precise definition of behaviour.” Fowler has an even stronger opinion on this point at [8, page 11]: “You can easily see the messages by looking at the diagram. However, if you try to represent something other than a single sequential process without much conditional or looping behavior, the technique begins to break down.”

This is exactly what SOCCA is good at, namely specifying the precise behavioural consequences of coordinated collaborations. SOCCA does so by two different ways of sequentialising separate steps of hidden behaviour, i.e., the steps the functionality consists of. First, SOCCA sequentialises the detailed steps to be taken in view of the eventual implementation of the operations by means of separate internal STDs. Each STD is modelling behaviour needed for one operation. Second, SOCCA sequentialises the global steps to be taken in view of the collaborations between operations of classes by means of communicative views of external STDs together with collaboration constraints.

### 1.5 Conclusions and Related Work

In this paper, we have presented SOCCA as an object-oriented specification language supporting an arbitrarily fine-grained synchronisation of processes, i.e., active objects. Since we are focusing on collaboration concepts between objects, a critique of UML’s collaboration concepts has been given as a precursor and motivation to SOCCA’s advanced synchronisation mechanisms. To complete the picture, out of the huge amount of object-oriented modelling approaches, two areas of related work are briefly discussed in the following, first some competitive approaches to UML, and second two representatives of object-oriented coordination languages.

Concerning the former, Catalysis [22], Syntropy [4], and OML [10] are nowadays the most influential object-oriented modelling approaches next to UML. All three are very strong in providing a development process and design guidelines, however, they fall short when it comes down to specific interaction mechanisms. Concerning the degree of synchronisation, they only support synchronous com-
munication. Concerning concurrency, only Syntropy discusses this issue in detail [4]. In contrast, the real-time object-oriented modelling technique ROOM supports both synchronous and asynchronous communication, and the specification of concurrently active threads of control. There is an ongoing effort to integrate the real-time concepts of ROOM into UML [21]. Last but not least, also Harel’s extension of the statechart mechanism incorporating object-oriented concepts has to be mentioned [11]. Statecharts together with O-charts support multiple-thread concurrency together with a broadcast mechanism for communication between concurrent components.

The second kind of approaches concentrate on the specification of obligatory and/or allowed interaction patterns between objects of the same or of different object classes. Contracts [12] are specifications of such interaction patterns with obligations on the participating objects. Each object fulfilling the corresponding obligation may participate in some contract. Contracts may be inherited and refined by adding participating objects, and by specializing besides others the inherited invariants and obligations. Since contracts are defined independently of object classes, class definitions must ultimately be mapped to participant specifications, which is done through conformance declarations. A contract is instantiated by identifying objects as participants and invoking the methods specified in the contract’s instantiation statement. A similar approach is followed in CoLa (Communication Language) [5], which allows to model the obligatory/prohibited/permitted interaction patterns between several objects. These interaction patterns are specified in terms of contracts between two or more so-called agents. In other words, contracts specify a set of norms which give rise to a certain interaction protocol between different agents, which is implemented in turn. CoLa has been used to implement different kinds of speech acts [5].

Modern software systems require specifications of complex interactions as they are available, for example, in SOCCA, but lacking in UML. Hence, we are investigating means of incorporating SOCCA’s collaboration features in UML by changing UML’s meta model as little as possible. One promising approach is based on UML’s extension mechanism, which is currently investigated in more detail. Our long term goal is the identification of interaction patterns encapsulating the most important recurring collaboration structures.

This paper focussed on object-oriented modelling approaches and did not treat the programming level. But, obviously, any model has to be transformed into a corresponding program in order to yield a complete development process and finally a running system. Currently, different target platforms are investigated, like Java, aspect-oriented programming languages [15] or coordination languages [9].
Bibliography


