Abstract

Conceptual modeling is the activity of producing a conceptual model of an actual or desired version of a universe of discourse (UoD). In this paper, two methods of conceptual modeling are compared, structured analysis (SA) and object-oriented analysis (OOA). This is done by transforming a model produced by the one into a model produced by the other method, using heuristics from several sources, such as Jackson system development and formal specification. It is shown that SA and OOA diverge in three important respects. First, the ordering of tasks in SA is shown to be virtually opposite to the task ordering in OOA. Second, a model produced by SA mixes information about the communication between objects as well as about the life cycle local to an object, which is separated in a model produced by the OOA method we propose in this paper. Third, the heuristics in SA are shown to be data-oriented, which leads to quite different modularization decisions than the object-oriented heuristics proper to OOA. The different approach taken by OOA on all three points is shown to lead to simpler models that better reflect the structure of the UoD.

Keywords: Object-oriented analysis and design


1 Introduction

In conceptual modeling, we specify an explicit conceptual model (CM) of the actual or a desired version of the UoD. Any method of conceptual modeling consists of advice on how to go about this in a systematic way. In particular, a conceptual modeling method should give advice on which tasks to perform to produce a CM, in which order these tasks should preferably performed, and what the principles and guidelines are that can help the modeler to make decisions in performing these tasks.

There are several attempts to combine SA and OOA, either by incorporating OOA in SA (Ward [35]), or by incorporating SA in OOA (Bailin [4]), or by using the output of SA as the input to object-oriented design (Alabiso [2], Seidewitz and Stark [31]). In this paper I show that these approaches all have problems, because SA and OOA are fundamentally incompatible. However, this incompatibility is argued by comparing the two modeling approaches, so that by its very argument, this paper shows that the two methods can be compared. Comparison is useful, first, for showing the difference between these models. Second, it helps us to judge the relative merits of the models, and therefore of the methods by which the models are produced. Third, a comparison of the design heuristics and model structures in SA and OOA facilitates the transition from SA to OOA.

The approach in this paper is to take elements of OOA that can be found in Booch’s [7, 8] work on object-oriented development and design, and combine this with elements of a traditional information system (IS) development method, Jackson System Development (JSD) [10, 21, 29, 34]. In a companion paper, JSD is analyzed in detail as a method for developing formally specified object-oriented models [39]. In section 2, we look at an example CM from Gane and Sarson [15]. Using this example in section 3, I contrast the task ordering advised by SA and OOA with respect to specifying functions of an IS and a model of the UoD. In section 4, I then compare the example SA model given in section 2 with a model produced by OOA, with respect to the structure of the model as well as with respect to the heuristics used in finding this model. This is illustrated by transforming the example SA model into an object-oriented model. Section 5 contains a discussion and some conclusions.

2 A structured analysis example

Gane and Sarson [15] use the CBM (Computer Books by Mail) corporation as UoD for their example CM. Very briefly, CBM acts as a postbox between computer professionals and publishers. It accepts book orders from professionals, orders the books from publishers at a discount, and sends them to the professionals. Professionals may have an account with CBM. Those without an account must pay in advance.

SA analyzes a UoD by following the data through it. In their movement through the UoD, data are transformed, and the connections between the different transformations of the data are shown in a data flow diagram (DFD). Figure 1 shows the part of the DFD given by Gane and Sarson that deals with the customer ordering process of CBM corporation.

A rectangle in a DFD represents an external entity, which is an entity outside the system boundaries that acts as a source or sink of data. An arrow in a DFD represents a data flow and
Figure 1: A data flow diagram.
A bubble represents a data transformation. A data transformation is a function that produces output from input. Bubbles in a DFD are numbered for easy reference. The numbers have no ordering significance. Finally, a data store holds data that can be read out by a data flow after it has been put in by a data flow, and is represented by two parallel lines.

A DFD is intended to show the modular structure of an IS. The arrows show the interfaces of the modules, and their direction shows the flow of data between the modules. They do not indicate the order of processing. One can start reading a DFD from any bubble and follow the flow of data from it.

The CBM customer order process accepts orders (bubble 1), checks whether any prepayment that is included is OK (bubbles 3 and 5), checks whether the ordered books are in store (6), and sends the books if available (7). For books that are out of stock, a back order is created (13). When books are received from the publisher, they are sent to the customer (16). An order history file is kept to answer customer queries (10) and there is an algorithm that determines when books must be reordered from a publisher (9). In the following sections, we transform the complete DFD given by Gane and Sarson, including the fragment shown in figure 1, into an object-oriented model.

3 Specifying functions of an IS

Both SA and OOA aim at defining a modular structure of a CM, but they use radically different principles of modularization. Booch [7] points out that SA encapsulates algorithms, not data, whereas OOA encapsulates the state and behavior of objects. SA is characterized by

1. its orientation on the tasks (functions) to be performed by the IS in the UoD, and
2. its view of these tasks as data transformations.

We will call this functional orientation and data orientation, respectively. OOA is characterized by

1. its orientation on the objects in the UoD performing or suffering tasks, and
2. its view of these objects as encapsulating state and behavior.

The result is, Booch notes, that interesting data end up global to the entire system in SA, but are encapsulated as objects in OOA. This leads to an increased modularity of systems produced by OOA.

The functional orientation of SA leads to a process of functional decomposition, which produces a DFD that contains a model of the UoD as well as a specification of the functions of the system. Almost as an afterthought, Ladden [24] notes, functions are packaged into modules that are the input to the design stage. The object-orientation of OOA turns this task ordering around. It first packages tasks and data into objects, and then defines the functions of the system to be implemented in terms of these objects. Thus, in OOA we first define a model of

---

1Data transformations are usually called processes or functions in business analysis. I follow the terminology used by Ward and Mellor [36], because I use the words “process” and “function” in a more specialized way later in this paper. I also use DeMarco’s [14] notation for data transformations instead of the Gane and Sarson notation.
the UoD, and then define the functions of the IS in terms of this model. These two concerns are not separated in SA and this leads to more complex models. In OOA, (1) these concerns are separated, and (2) specification of the models precedes specification of the functions. It is important to list the reasons why these are good ideas.

First, as stated by Jackson [21, pages 8-13], if we define functions before defining a model, the functional specification is stated in undefined terms and therefore is necessarily ambiguous. Second, a model of the UoD is more stable than the functions to be performed by the system. In information engineering terms, functions change as business procedures change, but the subject areas and their structure remains stable [27]. Third, changes to functions are more easily implemented because they all use the same underlying model. Fourth, a model specified independently from functions corresponds more intuitively with the structure of the UoD, so that the specification will be more understandable to the user. This is not surprising, because object-orientation grew out of a concern with modeling the UoD in the language Simula [13] back in 1967.

It is obvious that we cannot draw model boundaries if we do not know what the system is to be used for, i.e. what its functions are going to be. However, these functions can be specified only after the model is specified, and this should therefore be the order of specification activity.

JSD recognizes three kinds of functions. Input functions filter typing errors and other noise in the messages from the UoD to the system. An example is part of the function of bubble 1 in figure 1, which eliminates errors in book details in a customer order. Output functions report on the state of the system to a user. All queries, such as bubble 10 in figure 1, belong to this class. Interactive functions consist of a condition on the state of the system, and a trigger to initiate an event when the condition is satisfied. Bubble 9, for example, triggers a reordering process when the number of book copies in store for a particular book title falls below a certain level.

We eliminate all three kinds of function from the object-oriented model in which figure 1 will be transformed.

4 Object-oriented analysis

4.1 Objects

4.1.1 Distinguishing features of objects and heuristics to find them

We remarked that OOA analyzes the UoD as consisting of objects and not as functions, as SA does. Objects have the following characteristics.

1. Objects are discrete.

2. An object has a behavior that consists of performing or suffering events.

3. An object has a local state that determines its possible future behaviors.

4. Each object is capable of receiving a globally unique proper name, called its identifier.

5. Objects exist in the UoD and not merely in the IS.
Characteristic 1 contrasts objects with *masses*. Following Bunt [9], we define a mass as an entity that can be split or merged while preserving type. For example, masses like water, wood, gold, or profit can be split into two amounts of substance of that type, and still remain two amounts of water, wood, etc. By contrast, discrete objects like trees, persons, and employees cannot be split into two and yield two objects of the same type (see also Abbott [1] for this distinction). Closely related to this property of objects is the fact that objects can be put in a set. Because a set has a cardinality, it makes sense to say of a collection of objects how many of them there are. By contrast of masses we must say how much of it there is, according to a certain measure.

Having said this, we must add a bit of nuance by noting that many entities in the real world are hybrid in that they have characteristics of a mass as well as of a discrete object. Is a company an object or a mass? It can be split into two companies and merge with other companies and preserve type. Yet, companies can be counted, and we do not say how much of a company there is according to a measure. This does not mean that the distinction between objects and masses is invalidated, but it does mean that the distinction is a simplification of the real world. All conceptual modeling approaches I am aware of apply this simplification, and they all have problems with counting hybrid objects like companies. If two companies merge, how many companies must we say there are in a historical query? We do not propose to solve this problem, but point out that this is an important area in which theoretical work could be done.

Characteristics 2, 4 and 5 are taken from JSD [21, page 66]. The identifier mentioned in 4 is a surrogate for the real world entity represented by the object, that serves as the carrier of all properties assigned to the object. Codd [11], Hall et al. [18] and Khoshaﬁan and Copeland [23], among others, all proposed to make object identifiers an integral part of the model. In philosophical logic, identifiers (known under the barbarous name of “bare particulars”) have been used for basically the same purpose as they are used in conceptual modeling, as can be learned from the work of Grossman [17] and Loux [26]. The primary functions of identifiers are

1. to distinguish objects that have identical state and
2. to represent the preservation of identity through change of state.

If the identifier would not be treated as the object itself, but merely as a (key) attribute of the object, then changing this key would destroy identity information. Moreover, we would not be able to express the fact that “two” objects (say a student and an employee) with different keys (student number and employee number) are really the same.

Feature 3 is added to the JSD list to stress the encapsulation of state in objects. We allowed for nondeterminism by saying that a state determines future behavior. In deterministic objects, the current state of an object is uniquely determined by the past events in the life of an object. In nondeterministic objects, on the other hand, there is a random element in the current state of the object, that is not uniquely determined by the past. In both cases, however, the current state is all there is to determine possible future behaviors.

Finally, feature 5 is crucial for object-orientation and contrasts with the data orientation that we already noted is present in SA.

---

2Note, incidentally, that the name of a variable in a programming language has the same two functions as those of an object identifier, viz. to keep indistinguishable variables (which have the same value) apart and to maintain identity information through change of values.
The characteristics of objects are at the same time heuristics to find objects in a UoD. Other heuristics that are common are

- listing common nouns in descriptions of the UoD [1],
- looking for real-world objects with which the system must interact (these are the external entities in SA), and
- listing the principal actors that help to solve the problem which the IS is designed to solve [34].

### 4.1.2 Objects in SA

Although SA is function-oriented, it cannot help encountering objects. For example, following Alabiso [2], obvious candidates for objects in a DFD are external entities and data stores in a DFD. However, applying the criteria for objects listed above, we can do better than just listing all external entities and data stores as candidate objects, because we can reduce the number of objects considerably (table 1).

First, note that in SA we start by listing the entities external to the system to be implemented and then follows from the of data these entities feed to the system [14, 15]. This has the result that external entities are often duplicated as data stores, such as CUSTOMER. Relevant behavior of the external entities is represented as update operations on data stores (bubble 2). OOA simplifies this by just specifying objects and their behavior in the real world without duplicating this as data stores and update operations.
A second reduction of the number of objects follows from the functional orientation of SA, which leads to CM’s that contain the user as external entity. An example is the MANAGEMENT entity in the complete DFD given by Gane and Sarson for CBM corporation. We omit this from the list of objects as well.

A third reduction is accomplished by eliminating data stores that contain a trace of event occurrences. For example, ORDER HISTORY contains a trace of dated event occurrences that represent customer orders. We eliminate this by modeling the customer order process explicitly in subsection 4.5, and by regarding the trace of this process as part of the model execution. At a later stage, we may want to decide which traces to actually store in a computer.

A fourth reduction is effected by omitting all data stores that contain the set of currently existing instances of a class, such as INVENTORY, BOOKS and BACK ORDERS. The object classes involved are BOOK in the first two cases and CUS ORDER in the third case. In the DFD, a data store is used to store the current state of each existing instance of these classes. BACK ORDERS is special, for it contains customer order objects only in a certain state and not in any other state, i.e. in the state of waiting for a book that was not available to arrive from a publisher.

So far, we have looked at data stores and external entities as candidate objects. Turning to the relation between data flows and data transformations on the one hand and objects on the other, the picture that arises from the literature is a bit confusing. Alabiso [2] transforms data transformations convincingly into methods that may change the state of objects, but Ward [35] just as convincingly turns them into objects that contain methods. Seidewitz and Stark [31] encapsulate data transformations together with data stores into objects, and Bailin [4] does the same, but only in high-level DFD’s. I think the last approach is to be preferred, for the following reason.

In SA, we can “zoom in” on a data transformation B in a DFD by replacing it by a DFD that has the same data entering and leaving it as B. This zooming in can be repeated, until we reach the level of data transformations considered to be elementary. These bottom level transformations are memoryless functions that transform their input into output. A top-level transformations B, on the other hand, is likely to reveal data stores as well as transformations if we zoom in on it. This suggests that a top-level data transformation is likely to be an object, whose internal state and events are revealed as data stores and data transformations as we zoom in on it.

In figure 1, low-level transformations are shown, which are all candidates for events to be considered below. If we zoom out, we would see two bubbles CUS ORDER and PUB ORDER, that deal with the customer order and the publisher order process, respectively. These are reasonable candidates for objects because they satisfy all our criteria, so they have been added to table 1. This brings the number of object classes to 7, which is half the number of entries in the right-hand column of table 1. In this respect at least, the object-oriented model is considerably simpler than the DFD.

4.2 Classes and the identity of objects

A class is the largest set of objects that share a similar description, and an instance of a class is just an element of a class. It is important to realize that each class comes with a principle
of identity, as is known in philosophical logic for a long time. ³ For example, if we count the number of passengers a bus carried in a week, then we will come up with a different answer from when we count the number of people carried by the bus in the same week, because passengers and persons individuate differently. The principle that says when two instances of PASSENGER are identical is apparently different from the principle that says when two instances of PERSON are identical, even though all passengers are persons. Problems with the principle according to which instances should be counted have been noted in data modeling as well, for example by Kent [22, pages 2-8], Jackson [21, page 72], and Ward and Mellor [36, page 109].

In our example, we view a customer as a person of flesh and blood. Two customers can have the same address, but a single person is a single customer and vice versa. Publishers cannot be customers according to this decision. An employee of a publisher can, but if a job is shared by two people, then only one of them can be a customer representing the publisher.

An instance of BOOK is a book with a title, and can have many copies. number_in_store is an attribute of instances of BOOK that tells us how many copies of the book are in store. bulk_factor and safety_factor are two others, and are used in the computation of the reorder level of the book.

A PUBLISHER instance is an object registered as publisher according to the applicable legal procedures. Each customer and each publisher has exactly one account. A CUS_ORDER object is created when a customer orders a set of book copies. We will define an indivisible event that starts this process such that each occurrence of this event creates a different CUS_ORDER instance. Similar remarks apply to PUB_ORDER instances, except that this time, CBM initiates the process.

### 4.3 Attributes

At any time during OOA, we will discover attributes applicable to the instances of a class. We take a functional view of attributes and treat them as functions on classes. Because each possible object is identified uniquely by an identifier, we can view a class as just a set of object identifiers. An attribute then is a function that accepts an identifier and yields either a value or another identifier. For example, figure 2 says that in each state of the CM, the copies_in_store attribute of an existing book assigns a natural number to the identifier of that book, and publisher assigns a PUBLISHER identifier to that book.

The books attribute of CUS_ORDER assigns a finite set of books to an order identifier, shown by drawing an ellipse around BOOK. More details on set-valued attributes are found in [38].

Attributes are subject to static integrity constraints, which constrain the set of possible attribute values of objects in allowable states of the CM. I ignore this important topic here, because it is not a big issue in SA nor JSD. However, see [37, 40] for a detailed analysis.

³See for example van Leeuwen and Zeevat [25] and Wiggins [43].
4.4 Events

4.4.1 Distinguishing features of events and heuristics to find them

Objects perform or suffer events. By “event” we mean a general concept whose instances are smallest units of change. For example, the event \textit{send confirmation} (bubble 8) is a general concept, which is instantiated each time CBM sends a confirmation to a customer. An event instance is also called an event occurrence.

Each event must satisfy the following characteristics.

1. It is atomic, i.e. its occurrences takes exactly one tick of the clock.

2. It occurs in the life of an object.

3. It is capable of receiving a globally unique proper name.

4. Its occurrences exist in the UoD and not merely in the IS.

This list is adapted from the JSD characteristics of events [21, page 65], so that it partly parallels the list of distinguishing features of objects.

I assume there is one global clock, accessible without delay to all objects. The CM abstractly represents the UoD as a set of dynamic objects, which during their life perform or suffer events. These event occurrences are time-stamped by the global clock, and we can therefore imagine the CM to generate, during its life, a trace of time-stamped event occurrences, where the event occurrences of different objects are interleaved.

Implicit in the above is that each event occurrence maps a valid state of the CM (i.e. satisfying all static constraints) to a valid state of the CM. Event occurrences should maintain
static integrity. Also implicit in the above is that a state transition of the CM can consist of a set of event occurrences, all performed synchronously. Such a set is called a communication event and its elements are called messages. It is atomic in the sense that it is a single CM transition, without intermediary states. Thus, a communication event is similar to a DB transaction.

In this modeling approach the initiative of an event is not distinguished. Thus, that a CUS ORDER process is initiated by a CUSTOMER instance is not modeled. Similarly, we already decided not to specify in the CM the fact that the PUB ORDER process is initiated by an interactive function of the IS. We study event initiative elsewhere [41].

Heuristics to find events, in addition to the four characteristics above, are

- to list the verbs in a description of the UoD, and
- to consider events in the UoD that trigger system events only [34].

4.4.2 Communication between objects

A communication event is called a global event and is required to consist of a set of two or more events, all executed by different objects (i.e. with different identifiers, but possibly of the same class). For example, each CUS ORDER object asks the ordered BOOK objects whether sufficient copies of them are available in bubble 6. This then is a communication event, which we will model as consisting of two messages, performed by a CUS ORDER instance and BOOK instances. Since we do not model initiative, we do not say who actually sent the message and who received it.

Because events are required to be atomic, communication events must be synchronous (i.e. sending and receiving takes place at the same time). If asynchronous communication is needed (i.e. there is a time delay between sending an receiving, and the sender does not have to wait on the reception of its message in order to continue with its process), just put a mailbox between sender and receiver, which communicates synchronously on one end with the sender and at the other end with the receiver.

Given that we want synchronous communication, there are two ways to model this. One way is to let events be shared among several process. This is the approach taken in CSP [20], JSD [21], and the Oblog specification language [12, 32]. The other way to model this is to view sending and receiving as different events, that occur synchronously in a global event. This is the approach in CCS [30] and more explicitly in ACP [5, 6], and this is the approach I adopt. The reason for this is that we can then specify that sending takes place without receiving, which can occur if the communication connection is uncertain. In JSD, if the connection between sending and receiving is not absolutely certain, no communication is modeled at all [21, page 78].

Looking at figure 1, we see that virtually every data transformation represents a communication event. This is not strange, for a DFD is intended to show interfaces between modules. For example, a CUS ORDER object receives an order from a CUSTOMER object, verifies whether customer credit is OK with a CUS ACCT object, checks if the payment is OK with the ordered BOOK objects, checks if there are sufficiently many in stock with the same objects, etc. Not all of these communications need be modeled, though. We use an easy heuristic for this, which is that an object only participates in a communication if (read or write) access to its state is required. If only the identity of the object need be known during an event, then the
identifier can appear as parameter of this event and there is no need to model the event as a communication between objects.

For example, \texttt{receive\_order} is the first event in the life of a \texttt{CUS\_ORDER} process, and needs to know only the identifier of the customer who sent the order. We therefore do not model it as a communication with a customer. On the other hand, checking the number of copies of a book in store requires read access to the state of an ordered book to see what the value of the \texttt{copies\_in\_store} attribute is, and it is therefore a communication between a \texttt{CUS\_ORDER} object and a \texttt{BOOK} object. Similarly, actually sending the book in a \texttt{CUS\_ORDER} process requires updating the \texttt{copies\_in\_store} attribute of a book and this requires a communication between the \texttt{CUS\_ORDER} process and the book sent to the customer. Again, reception of a payment changes the state of a \texttt{CUS\_ACCT} object, but it also changes the state of the \texttt{CUS\_ORDER} process, because it determines the further course of events in this process such as whether to send the books immediately or ask for payment first. It is therefore modeled as a communication between a \texttt{CUS\_ORDER} object and a \texttt{CUS\_ACCT} object.

\subsection*{4.4.3 Events in SA}

Figure 3 shows the communications between objects as modeled in the DFD of figure 1. It can be viewed as a higher-level version of figure 1, with direction of data flow eliminated, data transformations replaced by events, and events grouped into objects. Object classes are shown as rectangles and events that can be performed or suffered by objects of a class are shown as smaller rectangles in it. As in Booch diagrams \cite{booch}, events that are visible to other objects protrude from the class rectangle. In our view, these are messages. Events that are not messages are \textit{local} and, in contrast to Booch diagrams, are drawn inside the class rectangle. All events are shown in figure 3. Finally, messages that participate in the same communication are connected by an undirected line.

One message may participate in several communications. For example, the \texttt{change\_copies\_in\_store} event in \texttt{BOOK} is performed when book copies are sent to a customer, and when books are received from a publisher. This involves two different communications.

Figure 3 was actually obtained by

1. first listing roughly which events occur in the life of instances of a class,
2. noting for each event whether it is part of a communication, and then
3. checking this by going through the life of instances of each class from start to end to see whether these events make sense.

This is the JSD way of determining the life cycles of objects (called entity structures there). Note that one data transformation can contain several events, as bubble 1, which contains \texttt{receive\_order} (in \texttt{CUS\_ORDER}) as well as \texttt{receive\_payment} (in \texttt{CUS\_ACCT}). On the other hand, one event can be distributed over several data transformations. For example, updating the customer account with the prepayment details is distributed over bubbles 1 (receive the payment) and 5 (update the account). This represents a lack of atomicity in the DFD that can cause violations of integrity, such as when the payment is received (so the customer fulfilled his or her obligations) but the account is not updated.
Figure 3: Communication between objects.
Figure 4: Showing identifiers in a communication.

Figure 5: The CUS_ORDER life cycle.

We could be more informative in figure 3 by using variables to indicate the identifiers of the participating objects and the parameters of the events. For example, the payment communication could be shown as in figure 4. payment_received(o, a, c, m) says that order o receives the message that an amount m of money was deposited by customer c in account a, and receive_payment(a, o, c, m) says the event that account a received amount m of money from customer c to pay for order o. A variable is bound to an object identifier in a communication, and is bound only once in the communication. The informal meaning of the events just sketched, and the meaning of the communication parameters, should be formally defined in the model. This is a matter of the next step in CM specification, and I ignore it here. More details on this can be found in [39].

4.5 Life cycles

In addition to communications between objects, the DFD in figure 1 shows sequencing of events as well as choices between different courses of action. With a tentative list of CUS_ORDER events in hand, we can walk through the life of a CUS_ORDER object by tracing figure 1 from bubble 1. For each data transformation, we identify the event(s) it encloses and for each event, we ask whether it is a message or a local event. The result for the CUS_ORDER class is shown in figures 5-7. Communications are shown in figure 3. The graphs in figures 5-7 are called process graphs, which are a generalization of finite state machines because they can have
in infinitely many states. (See Baeten and Weijland [3] on process graphs.)

A CUS.ORDER object begins his life by suffering a receive.order event, possibly synchronous with a payment.received event. Figure 3 shows that payment.received is part of a communication with a CUS.ACCT object, because both objects change state in this event. We imagine that CUS.ACCT actually receives the money and sends this message to the CUS.ORDER object. This is a matter of choice (or company policy) and not a necessity.

The CUS.ORDER object next loops over the event of getting the price of the books ordered, to find out what total price of the order is. Details like the stop condition for this loop, and the books addressed in this query, must be specified in the next stage of model development. For credit orders, the amount is used to charge the customer account in the CHECK.CREDIT subprocess, for prepaid orders, it is used to check the prepayment in the CHECK.PAYMENT subprocess. Subprocesses are an extension to process graphs made by Spruit [33].

In the next few steps, the DFD in figure 1 implicitly represents sequence, choice, and communication, the basic operators of any process algebra (cf. [3, 30]). First, there is a sequence implied in the data flows from transformation 1 to transformations 3 and 5, for these are intended to be performed only after transformation 1 is performed. This sequence is made explicit in the process graph of figure 5, in which the next step of the CUS.ORDER process is to check the credit of customers who did not prepay their order, and to verify the payment of those who did. The CHECK.CREDIT process is a refinement of transformation 3 and CHECK.PAYMENT of transformation 5 in figure 1.

Second, the data flows from transformation 1 to transformations 3 and 5 represent a choice
between prepaid orders and credit orders (the initiative of this choice lies with the customer and this is not represented). The two flows marked “Prepayment request” and “Orders with credit OK” leaving transformation 3 also represent a choice. These choices are represented explicitly in the process graphs. To save space, the two events \textit{credit\_ok} and \textit{credit\_not\_ok} in the \textit{CHECK\_CREDIT} process (figure 6) are abbreviated in the communication diagram of figure 3 as the message \textit{credit\_ok}.

Third, the bi-directional flow between transformation 3 and the “ACCTS RECEIVABLE” data store is a communication, modeled explicitly in figure 3.

The \textit{CHECK\_PAYMENT} process subsumes transformations 4, 5 and part of 6 and is more explicit in its handling of payments that are not in order. Note that we may have a recursive process call. Upon reaching an end node, a subprocess execution returns to the appropriate node in the “calling” process.

The final nodes of \textit{CHECK\_CREDIT} and \textit{CHECK\_PAYMENT} all represent the same state, which is the state in which \textit{SEND\_BOOK\_COPIES} is performed (figure 7). We show this process graph as part of a state chart (see Harel [19]). There is one entry into this process, marked by the small arrow from \textit{SEND\_BOOK\_COPIES} to the starting state. The dashed line represents parallel composition, and the right-half of the box contains the name of the process performed in parallel with the process in the left-half of the box. Since this is \textit{SEND\_BOOK\_COPIES} itself, this recursively represents a set of instances of this process, executed in parallel. The intention is that for each book for which copies are ordered, a \textit{SEND\_BOOK\_COPIES} subprocess is started. Specification of further details of this must...
be done in the next stage of model development.

The SEND_BOOK_COPIES subprocess subsumes transformations 6, 13, 16, 7, and 8 and deals with sending the books that are available and placing back-orders for those that are out of stock. After performing the place_back_order event, the process waits until a PUB_ORDER object tells it the books are received. It then sends the book copies received to the customer. Each PUB_ORDER starts by ordering books from a publisher and does so by executing small sequence of events for each book it orders (figure 8). Upon receiving a set of copies of a book it has the choice of sending it to store or filling back-orders first and sending the rest to store. A precondition to be specified next should prohibit sending books to store when there are outstanding back-orders.

The objects for which the life cycle are not shown explicitly simply perform their events in any order.

5 Discussion and conclusion

The model of figure 1 was analyzed into two types of diagrams, a communication diagram and process graphs. In this way, we could separate the information about the interfaces of modules from information regarding sequences and choices of events. By grouping events into objects, we could draw the diagrams at a higher level of abstraction, that corresponds more closely with our intuitive understanding of the UoD.

This paper shows that SA puts more information into a DFD than merely the interfaces between data transformations (also called “functions” or “processes” in SA). Procedural information like sequence, choice and even parallelism is hidden in DFD’s as well. By factoring out this procedural information and placing it in process graphs as we did, we are able to reduce the complexity of the remaining part of the DFD. The remaining part of the DFD more accurately represents the communication interfaces between data transformations. In addition, by choosing
the object as modularization principle, we were able to simplify the remaining diagram (figure 3) even further. For example, the duplication of external entities as internal data stores could be eliminated. The result is a diagram that is similar to the ones drawn by Booch [8], with the added accuracy that communications are precisely specified.

The argument in this paper is built upon one particular case study and, if successful, can only show that that example can be transformed into an OO model. However, by referring to principles more general than the particular example used, I hope to have made plausible that other DFD’s can be similarly transformed. At the same time, I hope to have shown that the modularization principles as well as the elements of the models produced by OOA and SA are so different, that these two methods should not be used simultaneously in a single modeling effort. Thus, to repeat a statement made in the introduction, the two methods can be compared, but are incompatible in the sense that they should not be used in a single modeling effort. The modularization principles and diagram techniques are too different to be used in a single project.

The specification of the object-oriented model in this paper stopped before the specification would get formal. The specification of identifiers in the messages, preconditions for all events and the formal specification of classes and taxonomy have all been illustrated, using the same formalism [38, 39, 40, 42].

Further work should include the introduction of broadcasts to better represent get queryset and the initiative of events in a communication. Some preliminary work on the last topic has been done already [28, 41]. Another interesting extension of the formalism is to improve the representation of events that really are tests, like credit ok and credit not ok. Some interesting work has already been done on this topic by researchers in process algebra [16].

References


