Combining Static and Dynamic Modelling Methods: A Comparison of Four Methods

R. J. WIERINGA
Faculty of Mathematics and Computer Science, Free University, De Boelelaan 1081a, 1081HV Amsterdam, The Netherlands

A conceptual model of a system is an explicit description of the behaviour required of the system. Methods for conceptual modelling include entity-relationship (ER) modelling, data flow modelling, Jackson System Development (JSD) and several object-oriented analysis methods. Given the current diversity of modelling methods, it is important for teaching as well as using these methods to know what the relationships between them is and to be able to indicate what the (im)possibilities of integrating different methods are. This paper compares three classical modelling methods (ER, data flow, JSD) on their possibilities for integration and combination. It is shown that there is a common core of these methods, which centres around the concept of system transaction and that unifies the static view of a system taken by ER modelling, with the dynamic view taken by JSD and the functional view taken by data flow modelling. Several object-oriented analysis methods integrate these three views. This paper illustrates how this is done in the analysis stage of Object Modelling Technique. Finally, it is shown that the transaction decomposition table can be used as a pivot around which to combine different methods. The results of this paper can be used in teaching to explain the relationships and differences between the methods analysed here, and in system development practice to ease the transition from structured to object-oriented methods.

Received October 15, 1994, accepted November 16, 1994

1. INTRODUCTION

In recent years, there has been a rising interest in the possibility to combine different conceptual modelling methods. Historically, the first combination is that of entity-relationship (ER) modelling and data flow (DF) modelling in structured analysis (Yourdon, 1989). Recently, object-oriented analysis methods such as Object Modelling Technique (OMT) (Rumbaugh et al., 1991), the method of Shlaer and Mellor (1992) and Fusion (Coleman et al., 1994) have arisen, that all adopt extensions of the ER modelling method and combine it with DF models, state machine models or with pre/postcondition style specifications. In addition, there has been a lot of interest in the possibility to integrate object-oriented modelling with Structured Analysis (Seidewitz and Stark, 1987; Alabiso, 1988; Bailin, 1989; Ward; 1989, Wieringa, 1991b). The possibility to combine Jackson System Development (JSD) with an object-oriented approach has also roused interest (Birchenough and Cameron, 1989) The possibility to combine JSD modelling with ER modelling is briefly discussed by Sutcliffe (1988), but is not studied there in detail.

These attempts at integration can be taken one step further by showing that there is an underlying idea of these different methods, that can be used as a guideline for combining different methods. It is the aim of this paper to show that there is such an underlying idea, that can be represented by something called the transaction decomposition table. This table allows us to represent the connection between the static and dynamic system structure in a simple way and provides a useful entry point to the analysis of different methods to see whether and where they can be combined.

In addition to allowing us to see how different methods can be integrated, showing what the underlying idea of the different methods is has at least three other advantages. As Hsia et al. (1993) remark in their recent status report on requirements engineering, complex system development probably requires several requirements engineering methods, and we therefore need a precise understanding of the relations between different methods and notations, so that it will be easier to do consistency checking across and translations among them. This paper presents a step along the road to such an understanding.

Second, an improved understanding of the underlying idea of different methods can help analysts to make the transition from current structured analysis methods to object-oriented methods. If we see what the common core of structured and object-oriented methods is, we can also see what the (real) differences are and therefore which steps to take to move to object-oriented modelling. A third advantage of isolating a common core of conceptual modelling methods is that it allows teachers to give a more principled exposition of these methods, that goes beyond a dull enumeration in the style of 'method A does this, method B does that, and method C does it differently again'. In fact, this is what motivated the research reported in this paper (Wieringa, 1994b).

The paper focuses on the duality between static and dynamic modelling of a system. The prime example of a static modelling approach is ER modelling. The other
three methods discussed here, DF modelling, JSD and OMT, all include the dynamic aspects of the system in one way or another. It is convenient to take JSD as our starting point, because, as shown below, it contains the common core of the different methods—the transaction decomposition table—in its purest form. Section 2 therefore contains a very brief introduction to the essentials of JSD and explains the meaning of the transaction decomposition table. Section 3 shows how to combine ER with JSD. This ER extension of JSD allows us to show in Section 4 what the relationship between JSD and DFD modelling is. In Section 5, it is shown that OMT is a sophisticated version of a combined JSD–ER–DF method. Throughout the argument, the transaction decomposition table plays a pivotal role. Section 6 discusses how the transaction decomposition table allows us to combine function decomposition and object-oriented decomposition in a simple way. Section 7 concludes the paper.

2. JSD
The major ideas behind JSD are the following (Jackson, 1983; Cameron, 1986; Michael Jackson Limited, 1986; Sutcliffe, 1988):

- A system model must be partitioned into a model of the universe of discourse (UoD) of the system and a model of the functions of the system. In a database system, the UoD is that part of the world about which the system registers data. In a control system, the UoD is that part of its environment whose behaviour it registers and controls.
- The UoD is modelled as a network of communicating entities. Each entity in the UoD has a life cycle. Entities communicate through shared actions in their life cycles.
- The system is modelled by a network of communicating processes too. The nodes in this network correspond to UoD entities or to system functions. Communication in the system network is more complex than communication in the UoD network and may be synchronous or asynchronous.

Jackson (1983) argues that the separation of a UoD model from a function model gives a better system structure than a design that would start from required system functions, because the UoD model is more stable than the list of required system functions. The modular structure of the UoD model allows for an easier change to system functions than a functionally designed system does.

The idea to build a UoD model before building a model of system functions comes from Jackson Structured Programming (JSP) (Jackson, 1975), where a program structure is designed starting from a representation of the structure of its input and output files. In JSD, a system structure is similarly designed by starting from the structure of the system environment. The environment in this case does not consist of files but of the UoD of objects to be registered or controlled by the system. In the next two subsections, the structure of the UoD model and of the the model of system functions are discussed. For brevity, in what follows, 'system model' stands for 'system function model'.

2.1. The UoD model
For illustration, a very simple model of the UoD of a student administration is given. In the UoD registered by this administration, there are students who may enroll for courses, register for tests, do a test they registered for and receive a mark for their performance on a test. In this UoD, we distinguish three entity types, STUDENT, COURSE and TEST. Typical life cycles for instances of these entities are shown in Figures 1–3. Life cycles are represented by process structure diagrams (PSDs), which are trees of which the root is labelled by the name of an entity type. The leaves are labelled by the names of actions in the life of entities of this type. Intermediary nodes are labelled by a name for a part of the life of the entity, plus possibly an asterisk (*) to represent iteration or small circle (o) to represent choice. Unmarked boxes represent left-to-right occurrence of processes or actions.

An important step in the JSD method is the allocation of actions to entities. The result of this step can be represented by an action allocation table such as shown in

![Diagram of PSD for a simple STUDENT life cycle.](image1)

![Diagram of PSD for a simple COURSE life cycle.](image2)
Figure 4. The table is convenient to discover common actions between different entities. A similar table is used in SSADM as a heuristic to find life cycle diagrams (called entity life histories there) (Eva, 1992, p. 187). The action allocation can be refined by replacing each entry by a C, U or D according to whether the entity is created, updated or deleted by the action; this will be done at the end of the next subsection for the student administration example. Note that the resulting table would be very similar to the entity–function matrix used in Information Engineering and can be used to verify that each entity is created and deleted (Martin, 1982). The action allocation table is a rudimentary version of the transaction decomposition table, discussed at the end of the next subsection.

2.2. The system model

The UoD is represented in JSD as a network of communicating entities. To get a system (function) model, we imagine each JSD entity in the UoD to be duplicated by a surrogate in the system. Each surrogate has a life cycle that mirrors the life cycle of the UoD entity that it represents. The UoD network is accordingly reinterpreted as an initial version of the system network. JSD does not represent the shared actions in the system network, so the system network initially consists of a set of disconnected nodes that represent surrogate types. The instances of these types are surrogates, that represent UoD entities.

JSD then proceeds by adding nodes to this system network that represent function processes. These are connected to nodes already in the network by special communication links. Figure 5 shows a simple system network containing two surrogate processes from our example (STUDENT and TEST) and two function processes. LIST.PARTICIPANTS should, upon request, list all participants of test and LIST.RESULTS should, upon request, produce a report about the aggregate results of a student. This network is explained in the following paragraphs.

JSD distinguishes three kinds of functions, input, output and interactive functions. An input function accepts data about the UoD and updates the appropriate system processes, an output function reports about the state of the system and an interactive function is a trigger that, when a certain state of the system occurs, immediately updates the system. The two functions in Figure 5 are output functions. Functions are processes, just like entities in the UoD, and surrogates in the system are processes, and their structure can be represented by PSDs.

Function processes are represented by nodes in the system network and can be connected to other nodes by three types of connections. A data stream connection is an unbounded first-in first-out buffer between two processes, and can be used to realize asynchronous communication. Data stream connections are represented by circles. For example, in Figure 5, the LIST.PARTICIPANTS function removes data from a data stream D that is filled by a TEST instance. To make this work, we should extend the PSD for TEST with a write statement that writes the relevant data to D every time a student registers for a test. The input data stream request.J is filled by the environment of the system, and the output data stream list.of.test.participants is emptied by the environment. These external connections are not shown. Note that there is one LIST.PARTICIPANTS instance for each TEST.

A state vector connection is a ‘window’ that one process may have on another, by which the observer can see what the current state of the observed process is, without disturbing the observed process. Communication through state vector connections is synchronous. State vector connections are represented by diamonds in the system network. For example, LIST.RESULTS and LIST.PARTICIPANTS read student state vectors through SV1 and SV2, respectively. The cardinalities indicate that each of these functions can read the state vectors of many students. In addition, one student state vector can be read by many LIST.PARTICIPANTS instances.

A controlled data stream connection is a communication in which one process, the observer, checks the state of another, the observed process, and if the state satisfies a certain condition, sends a message to the observed process. All of this occurs as one, atomic communication. Controlled data streams are often used to connect interactive functions to surrogate processes. For example, an interactive function that monitors a stock level could check the level each time stock is withdrawn and send a message that stock must be reordered when this condition occurs.
It is remarkable that the network of shared actions in the surrogate processes of the system is not shown. After all, it is important for the integrity of the system that surrogates which suffer a common action are updated together, so that there is no system state visible in which one surrogate is updated and the other is not. For example, we would not want the system to represent a state in which a student has registered for a test but in which the test does not have this student as a participant. In other words, the common actions must be system transactions that either occur entirely or do not occur at all. We can make this visible by the transaction decomposition table shown in Figure 6. The part of the table that decomposes input transactions is a reinterpretation of the action allocation table. As noted before, the C, U and D entries can already be put in the action allocation table. What makes this a transaction decomposition table is that we interpret the rows as input transactions, which correspond to UoD actions. The table shows that input transactions may affect several surrogates in the system simultaneously. The output transactions never change the system state but read the state of surrogates in the system.

The transaction decomposition table is the core of a conceptual model of the system, because it shows external system behaviour and relates it to our conceptual model of the UoD. In Section 4, it is shown that it also allows us to indicate the connection with functional decomposition and DF modelling.

3. COMBINING JSD MODELS WITH ER MODELS

In order to facilitate comparison of JSD with DF modelling and with OMT in the next two sections, JSD is extended with ER modelling in this section. The intention is to extend the UoD model with an ER diagram and to indicate consistency requirements on the life cycle model of JSD and the ER model.

3.1. Entities

In JSD, an entity is an individual in the UoD that is capable of performing or suffering actions and which can be given a unique name (Jackson, 1983, page 66). There is no such clear-cut definition of what an ER entity is. A search of the literature reveals four different definitions of what an ER entity is:

• An individual in the UoD (Chen, 1976, p. 10; Elmasri and Navathe, 1989, p. 40).
• A class of individuals in the UoD (Batini et al., 1992, p. 31).
• An individual in the system, i.e. a surrogate (Elmasri and Navathe, 1989, p. 42).
• A class of individuals in the system (Storey and Goldstein, 1988).

It is convenient to use the first entity concept in a combined JSD–ER approach. This mentions the essential characteristic that the entity must exist in the

FIGURE 5. A simple system network.

FIGURE 6. Transaction decomposition table of the student administration system.

FIGURE 7. A relationship that corresponds to a JSD entity type.
The action parameters end up as relationship components and attributes. For example, \( \text{mark}(s, t, m) \) corresponds to an instance \((s, t)\) of \( \text{TEST\_RESULT} \) with attribute \( \text{result} \). This attribute would have had to be initialized to null and would receive a value only if the student receives a mark for the test. Figure 9 splits this relationship into two and so avoids the need for null values of the type 'will get a value, but has not received one yet.'

- The arrow labeled course is a many–one relationship that assigns to each existing test \( t \) the course to which \( t \) belongs. It corresponds to the \( \text{create\_test}(c, t) \) action in Figure 8. This is an interesting observation, for \( \text{TEST} \) is an entity type with dependent existence, i.e. one whose instances are created by another entity in the model (Jackson, 1983, page 168). We may generalize this to the observation that an entity type with a dependent existence will have an entity-valued attribute that points to its creator in the model. Entity-valued attributes are an extension of the classical ER approach that is common in object-oriented modeling. They are modeled as many–one relationship types in the classical ER approach.

- The ER diagram adds cardinality constraints to the model. For example, by translating the \( \text{create\_test}(c, t) \) action into a many–one relationship (represented by the course arrow), we made explicit the information that \( \text{create\_test}(c, t) \) can only be performed once per test but can be performed many times per course. This information was already implicit in the PSDs. However, the \( \text{TEST\_RESULT} \) relationship adds cardinality information. We saw earlier that in the PSDs one student can receive several marks for one test. In the ER diagram, by contrast, each \( \text{TEST\_RESULT} \) instance is a relationship between a student \( s \) and a test \( t \), and for each pair \((s, t)\) there is at most one such relationship in \( \text{TEST\_RESULT} \).

In general, not all relationship types will correspond to common actions. Relationships can stand for part-of relations, element-of relations, contractual obligations, permissions, authorizations, etc. These are not normally viewed as common actions. Conversely, not all common actions in the JSD model will correspond to relationship types in the ER model. For example, if an elevator and an elevator motor share the actions \( \text{start} \) and \( \text{stop} \), this need not give rise to two relationship types between the \( \text{ELEVATOR} \) and the \( \text{MOTOR} \) entity types. If an action occurrence need not be remembered, it will not correspond to a relationship type in the ER model.

### 3.2. Relationships

There are two ways a relationship type can appear in a JSD model of the UoD—as a JSD entity type or as a common action. Consider first the appearance of an ER relationship type as a JSD entity type. In a model of a library, we may have a relationship type \( \text{LOAN} \) between \( \text{MEMBER} \) and \( \text{DOCUMENT} \), whose instances represent the fact that a member borrowed a document (Figure 7). The cardinality constraint \( 0, 1 \) written next to \( \text{LOAN} \) means that for each existing document, there is at most one existing \( \text{LOAN} \) instance. (Other conventions can be used, but that is not the point here.) This relationship type may appear in a JSD model of the same UoD as a JSD entity type, for it has a life cycle with such actions as borrow, extend, lose, and return.

Next, consider the appearance of a relationship type as a common action in the JSD model. Figure 8 shows the UoD network of the JSD model of the student administration UoD. The nodes in this network represent typical instances of an entity type and the edges in the diagram represent common actions. Each of these common actions happens to be an action whose occurrences we want to remember, and hence each of them corresponds to a relationship type in the ER model of the same UoD (Figure 9). The \( \text{enroll} \) action becomes the \( \text{ENROLLMENT} \) relationship type, \( \text{register} \) becomes \( \text{TEST\_REGISTRATION} \), \( \text{mark} \) becomes \( \text{TEST\_RESULT} \) and \( \text{create\_test} \) becomes the many-one relationship type \( \text{course} \). We can make the following observations about the correspondence between the communication diagram and the ER diagram.

- The action parameters end up as relationship components and attributes. For example, \( \text{mark}(s, t, m) \) corresponds to an instance \((s, t)\) of \( \text{TEST\_RESULT} \) with attribute \( \text{mark} \), that has value \( m \).

- If we would make an ER model without looking for common actions to be modeled as relationship types, we would probably have found a relationship type \( \text{TEST\_REGISTRATION} \) with attribute \( \text{result} \). This attribute would have had to be initialized to null and would receive a value only if the student receives a mark for the test. Figure 9 splits this relationship into two and so avoids the need for null values of the type 'will get a value, but has not received one yet.'

- The arrow labeled course is a many–one relationship that assigns to each existing test \( t \) the course to which \( t \) belongs. It corresponds to the \( \text{create\_test}(c, t) \) action in Figure 8. This is an interesting observation, for \( \text{TEST} \) is an entity type with dependent existence, i.e. one whose instances are created by another entity in the model (Jackson, 1983, page 168). We may generalize this to the observation that an entity type with a dependent existence will have an entity-valued attribute that points to its creator in the model. Entity-valued attributes are an extension of the classical ER approach that is common in object-oriented modeling. They are modeled as many–one relationship types in the classical ER approach.

- The ER diagram adds cardinality constraints to the model. For example, by translating the \( \text{create\_test}(c, t) \) action into a many–one relationship (represented by the course arrow), we made explicit the information that \( \text{create\_test}(c, t) \) can only be performed once per test but can be performed many times per course. This information was already implicit in the PSDs. However, the \( \text{TEST\_RESULT} \) relationship adds cardinality information. We saw earlier that in the PSDs one student can receive several marks for one test. In the ER diagram, by contrast, each \( \text{TEST\_RESULT} \) instance is a relationship between a student \( s \) and a test \( t \), and for each pair \((s, t)\) there is at most one such relationship in \( \text{TEST\_RESULT} \).

In general, not all relationship types will correspond to common actions. Relationships can stand for part-of relations, element-of relations, contractual obligations, permissions, authorizations, etc. These are not normally viewed as common actions. Conversely, not all common actions in the JSD model will correspond to relationship types in the ER model. For example, if an elevator and an elevator motor share the actions \( \text{start} \) and \( \text{stop} \), this need not give rise to two relationship types between the \( \text{ELEVATOR} \) and the \( \text{MOTOR} \) entity types. If an action occurrence need not be remembered, it will not correspond to a relationship type in the ER model.

### 3.3. Modelling guidelines

The crucial part of JSD is the allocation of actions to entities. In a combined JSD–ER approach, we can use the following guidelines for this task:

- Allocate an action to an entity if it needs to change the local state of that entity.

For example, the a change of address of a library member should be allocated to the \( \text{MEMBER} \) entity.
type. If an action needs to change the state of several entities, JSD recommends allocating it to all those entity types. For example, if borrow would change the state of a DOCUMENT, we should allocate it to DOCUMENT as well as to LOAN.

This violates the object-oriented principle of encapsulation in which an action must always be localised to one entity. If we want to turn the combined JSD–ER method into an object-oriented method, we should therefore refine this principle. A simple way to do this is to replace any action that must update the state of more than one entity by a set of local actions, each of which is local to a participating entity, together with the constraint that these actions must occur synchronously:

- If an action needs to change the state of more than one entity, split it into as many local actions as there are types of entities of which it needs to change the state and add the constraint that these local actions must occur synchronously.

For example, borrow should be split into, say, doc.borrow and memb.borrow, local to DOCUMENT and MEMBER, respectively, and the constraint should be added that doc.borrow and memb.borrow always occur synchronously. The two local actions only update the local state of two entities but the synchronous occurrence results in a simultaneous update of the state of several objects. This solution is followed in several algebraic and logical specification formalisms (Costa et al., 1989; Ryan et al., 1991; Wieringa 1991a, 1993; Wieringa and Feenstra, 1993; Hartmann et al., 1994). It resembles specification of object interaction at the programming language level by means of contracts (Helm et al., 1990).

Another guideline of the JSD–ER approach is the following:

- Allocate an action to an entity to enforce a sequencing of actions in the life of this entity. Allocate it to several entities to enforce sequencing by means of common actions (or by means of a synchronisation constraint, as outlined above).

An example of this is that a sequencing constraint on register and mark in the life of a STUDENT is enforced by means of sharing the actions register and mark with a TEST life cycle.

Another useful heuristic for action allocation is the following:

- Allocate an action to an entity in such a way that queries about whether, or how often, the action occurred in the life of an entity, can be answered.

The discussion of relationship types and common actions in the previous section gives us the following heuristic:

- If a common action must be remembered, we model it as a relationship type in the ER diagram. Action parameters become relationship components or attributes.

However, we can go further than this and use relationship types to reduce the number of common actions in a JSD model of the UoD. Consider the LOAN relationship type again. All actions in the life of instances of this type can be viewed as common actions between members and documents, just as all attributes of LOAN can be viewed as common attributes of members and documents. By modelling LOAN as a relationship type in the ER model, and hence as a JSD entity type in the JSD model of the UoD, we are able to allocate these actions and attributes to LOAN only. Doing this, we do not violate the JSD rule that an action can only change the state vector of the entity in whose life it occurs, because the LOAN actions need not change the state vectors of members or documents.

Moreover, the structure of the model is simplified by doing this. The communication structure of a UoD model can be quite dense, which leads to the so-called ravioli problem noted in object-oriented modelling (Taylor, 1990): each class specification is easily understandable in isolation, but the interaction between classes is a dense bundle of communications that is hard to keep track of. The modelling guideline we can extract from this is the following:

- If this is possible without violating the rule that action effects must be local, represent relationships as JSD entity types in the JSD model of the UoD and allocate common actions to this relationship type.

Following these guidelines should lead to a more informative and simpler UoD model than JSD and ER can provide separately. It tells us how to avoid null values of the type 'will get a value in the future' and allows us to simplify the communication structure of the UoD network.

4. JSD MODELS AND DATA FLOW MODELS

Data flow modelling is part of structured analysis and has the same general pedigree as JSD, i.e. structured programming. However, JSD follows JSP by generalising JSP's data-orientation to what we may call UoD-orientation, i.e. model a system after its environment. By
contrast, DF modelling follows ideas from general systems theory such as that of close cohesion and loose coupling and does not have JSD's UoD-orientation. The major ideas behind DF modelling are the following (DeMarco, 1978, Page-Jones, 1988; Yourdon, 1989):

- There is no separate UoD model. A DF model is always a model of system behaviour, not of UoD behaviour.
- The system is modelled as a hierarchy of subsystems, which are either data transformations or data stores. Each event is received by a data transformation and may side-effect the system's data stores, and each response is produced by a data transformation, possibly using the contents of the system's data stores.

This means that a DF model should be compared to the system network of JSD—both are networks that represent the system—and not to the UoD model. Nevertheless, there is a relationship between DF models and the UoD model of JSD, and it is useful for an understanding of the differences between the two methods to compare these models. In the next subsection, this is done by transforming our combined JSD–ER model of the student administration UoD into a DF model of the administration itself.

4.1. Transforming a UoD model into a data flow model

A DF model represents a system as consisting of data transformations and data stores. A data store is a passive entity that remembers data written to it until the data is destroyed. A data transformation is an active entity that accepts input data and produces output data. The interfaces between transformations and stores are data flows, which can transport data items. A complex data transformation can be specified by giving a DF model of its internal processing. This results in a hierarchical structure of which the top level represents the entire system as a single data transformation and the leaves represent primitive data transformations.

The basic idea of transforming a JSD–ER model into a DF model is to transform all entity and relationship types into data stores, and all actions into primitive data transformations that update these stores. In more detail, the guidelines for transformation are as follows.

1. Transform all ER entity types and relationship types into data stores, changing their names into the plural.
2. Transform each action into a primitive data transformation, giving the transformation the same name as the action.
3. Connect a transformation by an input data flow to the external entity that generates the input. This external entity may have to be added to the DF diagram. This may very well be an entity already represented by a data store (e.g. a STUDENT external entity corresponds to the STUDENTS data store). Finding an external entity that is the source of an input flow is an addition to JSD. The data sent along the data flow consists of the parameters of the action.
4. Connect the transformation by read/write data flows to the data stores corresponding to each entity and relationship in whose life it occurs. (There is more than one such entity if the action is shared.) The read is necessary to check whether the entities exist and to fetch their current state in their life cycle. If it is necessary to create or delete the entity or relationship, or to update the state and possibly other attributes, then the write access to the data store is utilized.

Figure 10 shows the result of applying these rules to the student administration model. This transformation procedure is the reverse of a procedure to transform a DF model into an object-oriented model given elsewhere (Wieringa, 1991b). In the next subsection, this DF diagram is compared with the JSD–ER model of the UoD.

4.2. The UoD network and DF models

4.2.1. UoD orientation versus system orientation

The JSD model represents the UoD, whereas the DF model represents the system. This is visible by the fact that the DF diagram contains data stores and by the fact that some JSD entity types are duplicated as data stores and external entities. Conversely, the DF model also includes external entities, such as USER, that are not JSD entity types because they do not exist in the UoD. They are present in the DF model because they provide the system with input or are the destination of output.

4.2.2. Behaviour representation

JSD is a method that is well-suited to model the behaviour of reactive systems. These are systems whose response to an input may depend upon (part of) their history of past inputs. This is contrasted with functional systems, whose response to an input only depends upon that input. Reactive systems are more difficult to understand than functional systems. A functional system can be represented by a mathematical function, a reactive system must be represented by a more complex technique like finite state diagrams. The concept of reactive system was proposed by Manna and Pnueli (1992).

The PSDs used in JSD are perhaps not the best way to represent the behaviour of reactive systems; other techniques, such as state transition diagrams or state charts (Harel, 1987; Harel, 1988) may be more well-suited. However, DF models are certainly not well-suited to model the behaviour of reactive systems. All sequencing information is (intentionally) lost in the DF model. The DF model only shows which actions occur and how these interface with data stores and external entities. McMenamin and Palmer recommend modelling a system by a set of data transformations, one for each system transaction, that have no direct interfaces with
R. J. Wieringa

COURSES TESTS STUDENTS

FIGURE 10. Transformation of the JSD model of the test registration UoD into a DF diagram. The input data flows carry the parameters of the action to which they are connected. This should be documented in the data dictionary, but this is not shown here.

each other and only have interfaces with data stores and external entities. This may be well-suited for systems that are data-intensive and have a simple control structure, but it is not sufficient for reactive systems.

The JSD model, by contrast, shows system behaviour in its PSDs. Interfaces between PSDs are also represented, i.e. by common actions. Interfaces between PSDs can be represented by a UoD network (e.g. Figure 8). In Subsection 4.3, we briefly look at DF models extended with control processes and compare those with JSD models of the UoD.

4.2.3. Object-oriented versus data-oriented modularization.

The JSD model and the DF diagram both represent systems, and are concerned with finding modular system boundaries. However, the kind of systems in both models are very different and, consequently, the resulting modularisations of the system are very different. The DF diagram contains two kinds of subsystems: functional primitives, which do computations but have no memory that survives a single execution, and data stores, which have a memory but do not do computation. This is the state of the art in manual administrations, where people do the processing but rely for their memory on paper. It is also the state of the art in traditional file-based administrative applications. Let us call this data-oriented modularization.

This contrasts with the JSD approach, in which each module (which is a PSD) in a UoD model corresponds with a real-world object and has local state as well as behaviour. Let us call this object-oriented modularization.

A consequence of this difference in modularization criteria is that in a DF model whose data stores correspond to objects, the state of an object is separated from its actions. The state ends up as a record in a data store and an action ends up as a software component that accesses the data store.

Another consequence is that the state of an object, in the words of Booch (1986), is globally accessible in the DF diagram. Any data transformation that accesses a data store, has access to all records in the data store. This contrasts with the encapsulation of state and behaviour in JSD. In a JSD model of the UoD, an action can only access the local state of the object in whose life it occurs. This also holds for common actions, which can only access the states of the objects sharing them. Note however that in the JSD implementation stage, state vector separation is practiced and the state vectors of an entity type may all end up in a single file.

4.2.4. Reactive systems and objects

A complex data transformation can be specified by a DF diagram, which may itself contain data stores. These data stores remember (part of) the past history of the complex transformation, and may determine the

THE COMPUTER JOURNAL, VOL. 38, NO. 1, 1995

Downloaded from https://academic.oup.com/comjnl/article-abstract/38/1/17/353839 by guest on 17 January 2019
COMBINING STATIC AND DYNAMIC MODELLING METHODS

4.3. The UoD network and control-extended DF models

When DF models are used for the specification of reactive systems, then they can be extended with control processes (Ward and Mellor, 1985; Hatley and Pirbhai, 1987). In the method of Ward and Mellor, a control process is specified by a Mealy-style state transition diagram whose transitions are triggered by the occurrence of an input and can produce output. The input and output of a control process are called event flows. A control process can be connected through input or output event flows with data transformations, other control processes, and external entities.

It is possible to transform a PSD into a Ward–Mellor DF model. We show how this can be done under the assumption that all actions in the PSD are initiated by the environment. It is easy to see how the DF diagram should be adjusted if some actions are commands to an external entity. Take the simple PSD of Figure 11. According to the transformation guidelines given earlier, we define a data store that holds the state of all instances of $E$ and define a primitive transformation for each action of $E$ that accesses the data store. We now extend this with a control process whose state transition diagram corresponds with the PSD of $E$ (Figure 12). By assumption, each JSD action is received by the control process from the environment and in the DF model it triggers the appropriate data transformation, that updates the state vector in the data store. Common actions of different PSDs can be represented in a DF model by means of an action sent from one state transition diagram to another, where each of the state transition diagrams corresponds to a PSD.

This example illustrates that the extension DF models with control processes improves the behaviour representation of the DF model. However, whereas the behaviour representation of control-extended DF models is more comparable to the behaviour representation of JSD models, the other differences between the

FIGURE 11. A simple PSD.

FIGURE 12. DF diagram corresponding to the PSD of Figure 11.
two kinds of models remain. For example, there is no need at all to define a control process for each type of UoD entity. There are no guidelines in DF modeling for the modularization of the system in control processes other than that interfaces should be minimal. Control-extended DF models are still system-oriented and use a data-oriented modularization. Moreover, control-extended DF models can contain complex reactive subsystems that do not correspond to UoD objects or function processes.

4.4. The system network and DF models

We now turn to the comparison of the JSD system network with a DF model of the system. There is a more than superficial resemblance between the two kinds of models. Both represent the system as a network of communicating processes, that react to incoming events by producing responses. There are some differences between the models, but these are not as deep as the differences between the UoD network and a DF model:

- Process connections in the system network are specified in more detail, but not in a way incompatible with DF models. For example, data streams correspond to data stores (if they hold their data for some time) or to data flows (if they pass their data immediately).
- The nodes in a system network are types (whose instances are surrogates or function processes), the nodes in a DF diagram are individual transformations or data stores. Consequently, a system network must show the cardinality of process connections, a DF model does not.
- A system network is not leveled, as a DF model is. All processes in a system network are therefore 'primitive'. However, they are not functional primitives but reactive systems, because they have a local state. Furthermore, they are specified by PSDs, whereas the functional primitives of a DF model can be specified by pseudocode, decision tables or by other techniques.

Underlying these syntactic differences between representation techniques, there is a commonality between the two kinds of models, that can be brought out by using the transaction decomposition table. Each row in this table represents a system transaction. In DF modelling, each transaction is viewed as consisting of an event which occurs in the environment of the system, and of a response of the system to this event. The event may be generated by an external entity or by the passage of time (a temporal event). In the method of event partitioning, a DF model of a system is built by listing all possible events to which the system must respond and then defining a data transformation for each response that the system must produce (McMenamin and Palmer, 1984; Yourdon, 1989; Goldsmith, 1993). In terms of the transaction decomposition table, the event list is the list of transactions in the second leftmost column, and the response to an event is the processing done along a row of the table.

For example, the create_test transaction shows a U and a C in the transaction decomposition table of the student administration (Figure 6) for COURSE and TEST, respectively. Correspondingly, the DF model shows that the create_test transformation accesses the data stores COURSES and TESTS. This is the way the DF model was constructed in the first place.

If we want to build a DF model using the method of event partitioning, we would do well to build an event list and an ER model of the system first. Using this, we can fill in the transaction decomposition table by asking, for each event, which entity or relationships are created, updated or deleted. This gives a first hint at the processing done by the system in response to the event and hence at the DF model of the system. The transaction decomposition model is thus a core element in DF modelling.

On the other hand, it is also a core element in JSD, because it shows the allocation of actions to entities. The table therefore allows us to see what the underlying connection between these two kinds of methods is. The underlying connection is simply that in both kinds of models, the system is represented as engaging in transactions with its environment. In JSD, each transaction is modelled as a set of one or more local events in the life of system surrogates. In DF models, each transaction is modelled as an event and a response, that may update the data stores of the system. In both cases, this internal processing can be represented in rough form by the transaction decomposition table.

The transaction decomposition table also draws attention to the underlying differences. The system must respond to events that occur in its environment. It is therefore natural to make a model of this environment first, as done in JSD, and to model in particular the objects in the environment to whose events the system must respond, i.e. whose actions it must register or control. This approach gives a more modular structure to the system than the classical DF modelling approach.

The transaction decomposition table draws attention to yet another characteristic of DF modelling, that has not yet been pointed out. DF models represent interfaces at a level of abstraction that is too low. They represent data flow interfaces, whereas the behaviour of the system consists of transactions. A data flow between the system and its environment is a set of input parameters or a set of output parameters of an event or of a response. Data flows therefore only make sense in the context of an event or a response. These in turn only make sense in the context of a system transaction. For example, the input data flow document_nr cannot be interpreted if we do not know if it occurs as parameter of a borrow, extend, return or lose transaction. These transactions all have the same data flow interface. To be meaningful, the system model should show the transactions, not the data flows. The
data-orientation of DF models contrasts here with the transaction-orientation of the transaction decomposition table.

We can conclude from this discussion that DF models add little clarity to a combined JSD–ER approach. Nevertheless, current object-oriented modelling approaches like OMT and the method of Shlaer and Mellor use DF models to represent system processing. In the next subsection, it is analysed how this is done in OMT.

5. OMT

In this section, only a brief analysis of OMT is given. A more detailed analysis of OMT is given in a separate paper (Wieringa et al., 1993). OMT represents a system by using three models (Rumbaugh et al., 1991).

- The object model represents the class structure of the system. The object model of OMT is an extension of the classical ER model with taxonomic structures and aggregation. It also shows object operations in addition to object attributes.
- The dynamic model describes the behaviour and interaction of objects. This is represented by using state charts (Harel, 1987, 1988), which are an extension of finite state machines with, among others, the ability to represent substrates, parallelism and interaction between machines. For each class whose instances have interesting behaviour, a state chart is made.
- The functional model shows the meaning of the operations of the objects by showing how values are transformed. It is represented by a DF diagram. For each operation of each object, there should be a DF model that specifies what the operation does.

In the first two models, we can recognize an advanced version of the combined JSD–ER approach. The PSDs of JSD can be represented equivalently by finite state automata, which are simple versions of state charts. The major addition of OMT to the JSD–ER approach is the notion of inheritance in the object model. This creates a complication for the dynamic model, because it is not yet fully understood how life cycles are inherited. Rumbaugh et al. (1991, p. 111) suggest that a specialized life cycle should be such that the generalized life cycle from which it inherits, should be retrievable from it by projection. They do however not state what the projection of a state chart is. Recently, some approaches to life cycle specialization have been made that make the concept of life cycle projection more precise (Lopes and Costa 1993; McGregor and Dyer, 1993; Saake et al., 1994).

A methodological difference between OMT and the combined JSD–ER approach is that OMT does not distinguish a UoD model from a system model. Following Jackson's argument, this makes the system structure less modular and less maintainable. The distinction between UoD objects and function processes has been recognized elsewhere in the object-oriented literature as that between semantic classes and application classes (Monarchi and Puh, 1992). It resembles the distinction between entity objects and control objects (Jacobson et al., 1992).

The functional model roughly corresponds to a DF model that is made using the method of event partitioning. If we assume that each transaction leads to a number of system actions that are summarized in the transaction decomposition table, the functional model defines the meaning of the system actions shown in each row of the table. However, the situation is more complex here, because the columns now correspond to classes that are not orthogonal. One class may be a subclass of another. If a transaction has an entry under C in the table, it will have entries under all subclasses of C because these inherit the operation. (More optimal representations of the table can be found, but this is not the point here.) This complexity does not invalidate the use of the transaction decomposition table as a common core of different methods.

The presence of subclasses not only adds complexity to the transaction decomposition table, but also to the functional model. We can now find several data transformations that deal, at different levels of the inheritance structure, with the same transaction. The situation is even more complex in OMT, because the correspondence between the functional model and the other two models allows considerable freedom. Just as in the JSD–ER approach, external entities ('actors' in OMT) correspond one–one to object classes. However, in addition we have the following, more liberal correspondence rules Rumbaugh et al. (1991, pp. 137–139):

- Data stores correspond many-one to object classes. One object class may correspond to several data stores, each of which holds certain attributes defined for (instances of) the class.
- Data transformations correspond many-many to actions. One data transformation may specify the effect of several actions, and the effect of one action may be specified by several data transformations.

If we replace 'object class' with 'JSD entity type', then the transformation from Figure 11 into Figure 12 illustrates a simplified version of the above correspondence rules, in which all correspondences are one-one.

In terms of the transaction decomposition table, the functional model specifies the effect of transactions on the objects in the system. It does this using a data-oriented modularization and this does not cohere well with the object-oriented modularization of the rest of the model. In fact, it is this difference in modularization principles that makes it possible to allow the many–one correspondences listed above. Moreover, DF models specify the effect of transactions in an operational way, by specifying the processing done in response to a transaction. As remarked by Coleman et al. (1994), it is...
preference to use a declarative way of specifying the effect of transactions, because this leads to a higher level of implementation-independence. This approach is taken in Fusion and in a number of formal methods that are currently under development, such as Troll (Saake, 1993; Hartmann et al., 1994) and MCM (Wieringa and Feenstra, 1993; Wieringa, 1993, 1994a).

6. DISCUSSION: FUNCTION DECOMPOSITION

The transaction decomposition table allows us to pinpoint a common core of superficially very different methods, such as JSD, ER modelling, DF modelling and OMT. It is easy to show that other object-oriented methods, such as the method of Shlaer and Mellor (1992) and Fusion (Coleman et al., 1994) can also be analysed by means of it. The table is simple to understand and is well-known from such methods as Information Engineering and SSADM. An advantage of the table not yet pointed out is the following: it shows two orthogonal ways to modularize the system. In an object-oriented approach, the classes and their inheritance hierarchy show an object-oriented modularization of the system in which objects communicate with each other only through communications as shown in the table. The transactions, on the other hand, can be organized in a function decomposition tree as is well-known from Information Engineering (Figure 13). The root of this tree gives the overall function of the system, which is decomposed into lower-level functions until we reach atomic system transactions. Grouping the transactions into functions is a modularization that is highly significant for the system user. It turns out to be useful to divide the transaction decomposition table into chunks that correspond to nodes in the function decomposition tree. This gives another way to deal with the ravioli problem, because the communication structure in each chunk is relatively simple.

It may seem surprising that we can combine a function decomposition tree with an object-oriented modularization. The reason why this can be done is that the two ways of modularization are orthogonal to each other—

![Diagram](image)

**FIGURE 13.** A possible function decomposition tree of the student registration system.

which is literally shown by the form of the transaction decomposition table. The transactions in the second leftmost column of a transaction decomposition table are the leaves of a function decomposition tree and this is orthogonal to the class list in the top row of the table. This makes the function decomposition tree different from modularization constructs like subjects in the method of Coad and Yourdon (1990), subsystems in the method of Wirfs-Brock et al. (1990) and the ensembles defined by de Champeaux (1991). These different concepts share the idea that gather a number of objects classes that ‘belong’ to each other into a higher-level module. By contrast, the functions in a function decomposition tree gather a number of transactions that belong to each other into a higher-level construct.

Secondly, a function (non-leaf node) is in a function decomposition tree only if it contributes to the overall function (the root node of the tree) of the system. They show why the transactions should be performed by the system. Function decomposition is a decomposition of the system function into transactions, object-oriented decomposition is a decomposition of a system into classes.

Thirdly, the function decomposition tree decomposes the overall system function down to the level of atomic system transactions. As was pointed out earlier, this is above the level at which functional decomposition approaches like DF modelling decompose a system into modules. We can now add that subjects, subsystems and ensembles may be defined at any level of aggregation where they are useful, and this may be above or below the level of transactions.

It is possible to use functions as a modularisation construct, i.e. by drawing class diagrams, communication diagrams, etc., per function. However, because the decomposition of the overall system function into subfunctions is orthogonal to the decomposition of the system into classes, one class may appear in different functions. The transaction decomposition table can thus be used to define two orthogonal modularizations of the behaviour of a system. It is the basis of a method for conceptual modelling (MCM) currently under development, and which combines the UoD-orientation of JSD with object-orientation, transaction-orientation and formal specication (Wieringa and Feenstra, 1993; Wieringa, 1993).

7. CONCLUSIONS

We can draw two conclusions from this paper. First, we saw that the JSD and ER methods can be combined to form a conceptual modelling method that allows us to simplify the communication structure and at the same time is more expressive than either method apart. The combined method is a simple form of object-oriented modelling. The major simplification with respect to such methods such as OMT, the Shlaer–Mellor method and Fusion is the absence of inheritance.
Second, the transaction decomposition table is a common core of different methods, that allows us to understand and compare those methods. It is at the heart of JSD (as the action allocation table). It can also be used to show what is actually being achieved by the functional model of OMT (and of the Shlaer–Mellor method): the specification of the effect of transactions. It therefore gives us room to search for other means to achieve the same thing, such as specification by pre- and postconditions, formal specification, etc. The increased understanding provided by the transaction decomposition table can be used in teaching conceptual modelling methods and in easing the transition from older, data-oriented methods to object-oriented methods. In addition, it can be used to develop new methods that (hopefully) improve upon the current state of the art without throwing away what is good in the older methods.

ACKNOWLEDGEMENT

This paper benefited from comments made by Remco Feenstra on a draft of this paper. Thanks are due to the anonymous referees, who gave many useful comments on an earlier version of this paper.

REFERENCES


