Combining static and dynamic modelling methods: a comparison of four methods*

R.J. Wieringa†

June 5, 1996

Abstract

A conceptual model of a system is an explicit description of the behaviour required of the system. Methods for conceptual modelling include ER modelling, data flow modelling, 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 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 centers 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 OMT. 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.

*Appeared in *The Computer Journal*, 38(1), 1995, pages 17–30.

†Faculty of Mathematics and Computer Science, Free University, De Boelelaan 1081a, 1081HV Amsterdam. Email: roelw@cs.vu.nl

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 [53]. Recently, object-oriented analysis methods such as Object Modelling Technique (OMT) [34], the Shlaer/Mellor method [39] and Fusion [10] 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 [1, 2, 38, 44, 47]. The possibility to combine Jackson System Development (JSD) with an object-oriented approach has also roused interest [4]. The possibility to combine JSD modelling with ER modelling is briefly discussed by Sutcliffe [42], 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, 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, Davis and

Kung [21] 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 [49].

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 Jackson System Development (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 function decomposition and object-oriented decomposition can be combined, and section 7 concludes the paper.

2 Jackson System Development (JSD)

The major ideas behind JSD are the following [6, 23, 31, 42]:

- 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 [23] 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) [22], 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".

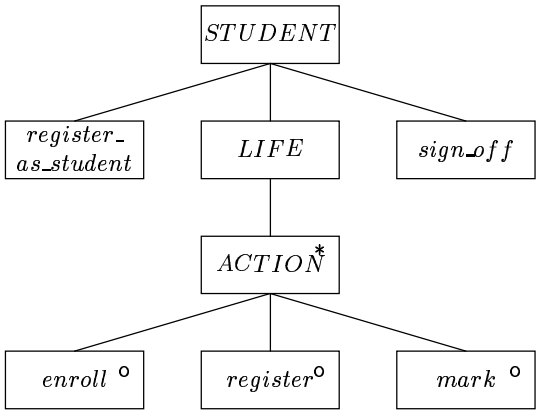

Figure 1: PSD for a simple $STUDENT$ life cycle.

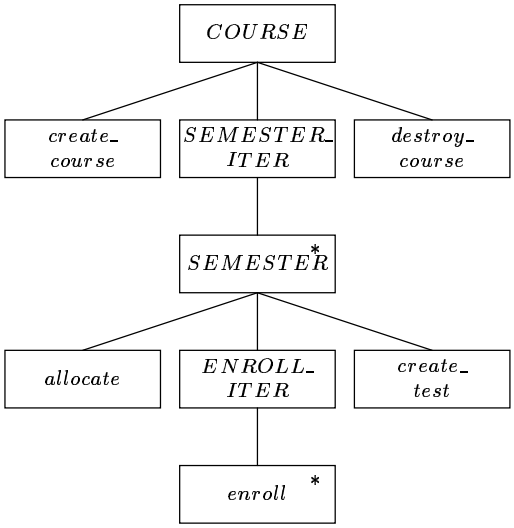

Figure 2: PSD for a simple $COURSE$ life cycle.

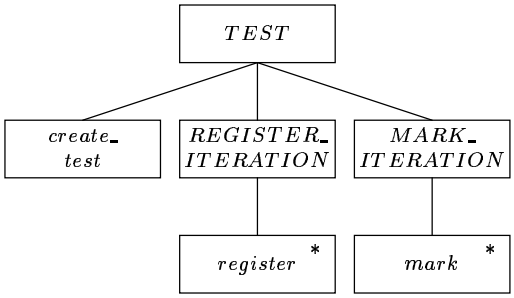

Figure 3: PSD for a simple $TEST$ life cycle.

2.1 The UoD model

To illustrate, 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, 2 and 3. Life cycles are represented by *process structure diagrams* (PSDs), which are trees of which the root is labeled by the name of an entity type. The leaves are labeled by the names of actions in the life of entities of this type. Intermediary nodes are labeled 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 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) [14, page 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

	$COURSE$	$STUDENT$	$TEST$
create_course	X		
allocate	X		
enroll	X	X	
destroy_course	X		
create_test	X		X
register		X	X
mark		X	X
register_ as_student		X	
sign_off		X	

Figure 4: Action allocation table of the student administration system.

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 [28]. 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 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

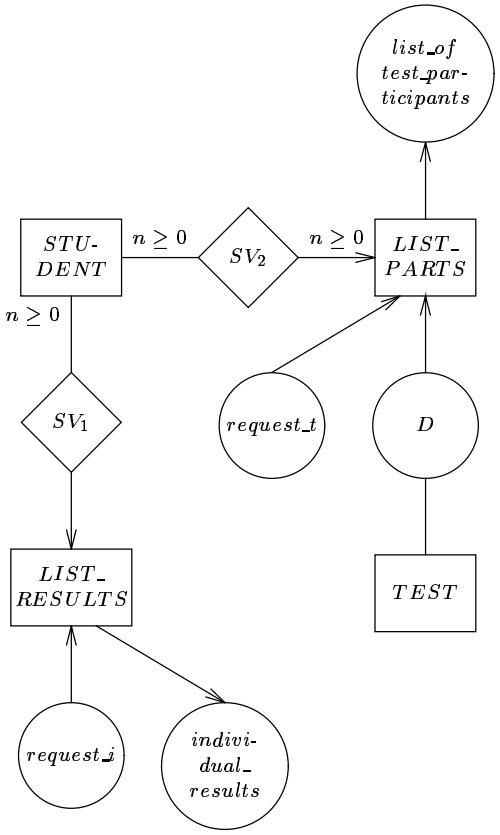

Figure 5: A simple system network.

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 realise 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 $request\_t$ 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 SV_1 and SV_2, 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 to put the system represent the 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 upper part of the table decomposes input transactions and 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.

	$COURSE$	$STU\text{-}DENT$	$TEST$
$create\_course$	C		
$allocate$	U		
$enroll$	U	U	
$destroy\_course$	D		
$create\_test$	U		C
$register$		U	U
$mark$		U	U
$register\ as\_student$		C	
$sign\_off$		D	
$list\_participants$		R	R
$list\_results$			R

Figure 6: Transaction decomposition table of the student administration 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 [23, page 66]. There is no such clear-cut definition of what an ER entity is. A search of the literature reveals four different definitions:

- An individual in the UoD (Chen [8, page 10] and Elmasri and Navathe [13, page 40]).
- A class of individuals in the UoD (Batini et al. [3, page 31]).
- An individual in the system, i.e. a surrogate (Elmasri and Navathe [13, page 42]).
- A class of individuals in the system (Storey [41]).

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 UoD (not in the system) but drops the characteristic that the entity must have something to do. This is a minor extension that makes life easier and does not affect the JSD part of the combined approach. It merely means that in the action allocation table, we allow some columns to be empty. (We don't worry how instances of those entity types are created or deleted.)

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

Figure 7: A relationship that corresponds to a JSD entity type.

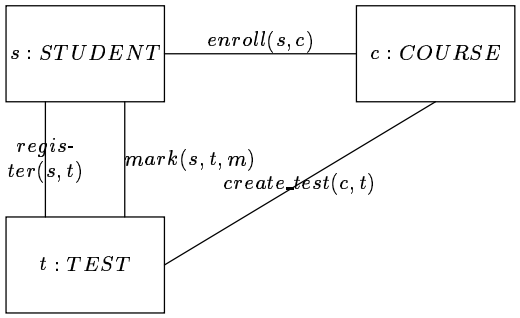

Figure 8: UoD network showing the common actions between $STUDENT$, $COURSE$ and $TEST$ instances. For clarity, the action parameters are shown in the diagram.

a relationship type $LOAN$ between $MEMBER$ and $DOCUMENT$, whose instances represent the fact that a member borrowed a document (figure 7). 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*. The cardinality constraint 0, 1 written next to $LOAN$ means that for each existing document, there is at most one existing $LOAN$ instance. (Other conventions can be used, but that is not the point here.)

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 the 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 that 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 *enroll* action becomes the $ENROLLMENT$ relationship type, *register* becomes $TEST\_REGISTRATION$, *mark* becomes $TEST\_RESULT$, and *create_test* becomes the many-one relationship type *course*. We can make the following observations about the corre-

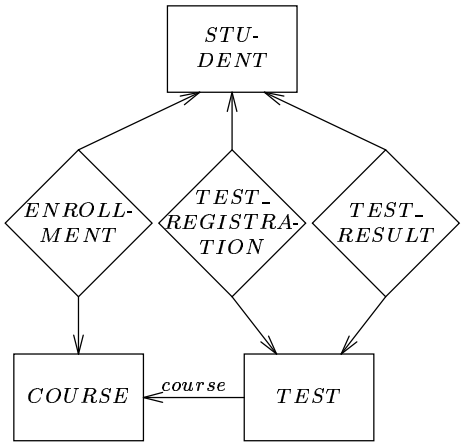

Figure 9: The ER version of the UoD network of figure 8.

spondence between the communication diagram and the ER diagram.

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

- If we would make an ER model without looking for common actions to be modelled as relationship types, we would probably have found a relationship type $TEST\_REGISTRATION$ with attribute *result*. This attribute would have had to be initialised 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 $create\_test(c,t)$ action in figure 8. This is an interesting observation, for $TEST$ is an entity type with *dependent existence*, i.e. one whose instances are created by another entity in the model [23, page 168]. We may generalise 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 modelling. They are modelled as many-one relationship types in the classical ER approach.

- The ER diagram adds cardinality constraints to the model. For example, by translating the $create\_test(c,t)$ action into a many-one relationship (represented by the *course* arrow), we made explicit the information that $create\_test(c,t)$ can only be performed once per test but can be performed many times per course. This information was already implicit in the PSDs. The $TEST\_RESULT$ relationship *adds* however 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 $TEST\_RESULT$ instance is a relationship between a student s and a test t, and for each pair $\langle s,t\rangle$ there is at most one such relationship in $TEST\_RESULT$.

In general, not all relationship types will correspond to common actions. Relationships can stand for part-of relations, element-of relations, contractual obligations, permissions, authorisations, 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 *start* and *stop*, this need not give rise to two relationship types between the $ELEVATOR$ and the $MOTOR$ 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 $MEMBER$

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 localized to one entity. Moving from a JSD/ER method to 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 entities, together with the constraint that these actions must occur synchronously:

- If an action needs to change the local 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 [11, 50, 18, 35, 46, 48]. It resembles specification of object interaction at the programming language level by means of contracts [20]. We now returning to the guidelines of the JSD/ER approach.

- 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.

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.

- 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 above discussion about relationship types and common actions 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 [43]: 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, viz. structured programming. However, JSD follows JSP by generalising JSP's data-orientation to what we may call UoD-orientation, viz. 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 [12, 33, 53]

- 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. More in 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 utilised.

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 [47]. In the next sub-

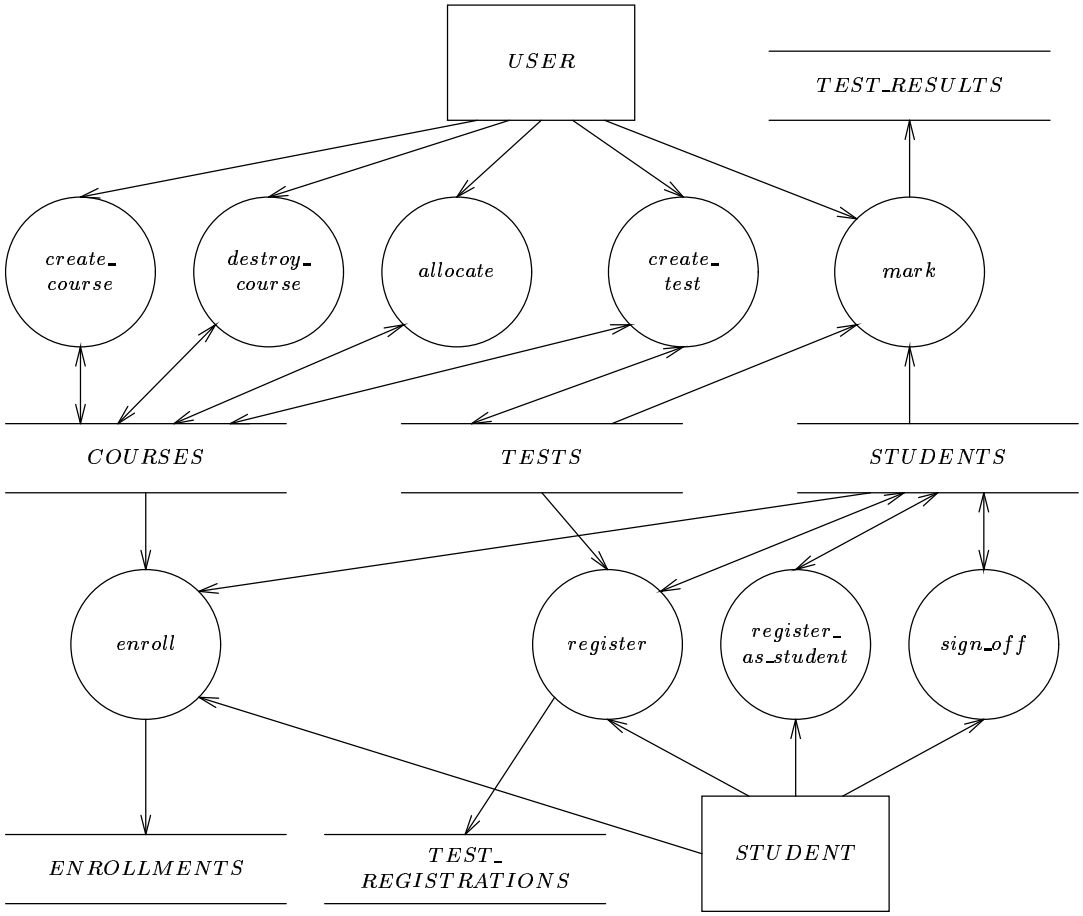

Figure 10: Transformation of the JSD model of the test registration UoD into a DF diagram. The input data flows carry the parameters of the action to which they are connected. This should be documented in the data dictionary, but this is not shown here.

section, this DF diagram is compared with the JSD/ER model of the UoD.

4.2 The UoD network and DF models

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.

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 [27].

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 [16, 17] 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 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.

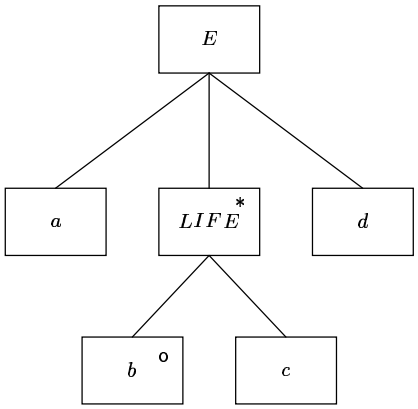

Figure 11: A simple PSD.

The JSD model, by contrast, shows system behaviour in its PSDs. Interfaces between PSDs are also represented, viz. by common actions. Interfaces between PSDs can be represented by a UoD network (e.g. figure 8).

When DF models are used for the specification of reactive systems, then they can be extended with control processes [19, 45]. In the Ward/Mellor approach [45], a control process is specified by a Mealy-style state transition diagram whose transitions are triggered by the occurrence of an input to the control process and that produce output of the control process. The input and output are called event flows, and these can be connected to a data transformation, from another control process, or from an external entity. Thus, a control process reacts on events produces by a data transformation, control process, or external entity. The reaction to an event may be to change state, to trigger, enable or disable a data transformation, to send a command to an external entity, or to send a message to another control process.

It is possible to transform a PSD into a Ward/Mellor DF model. The simple PSD of figure 11 corresponds to the DF diagram in figures 12 and 13 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. The state of E is split into what we may call a *control state* (the position in the PSD) and a *data state*, the values of local attributes. This corresponds in the DF diagram of figure **??**

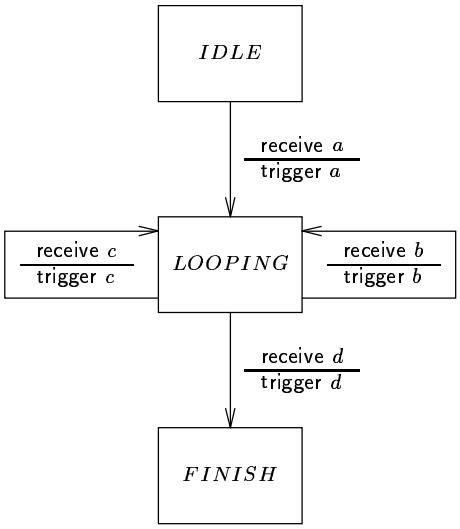

Figure 12: DF diagram corresponding to the PSD of figure 11.

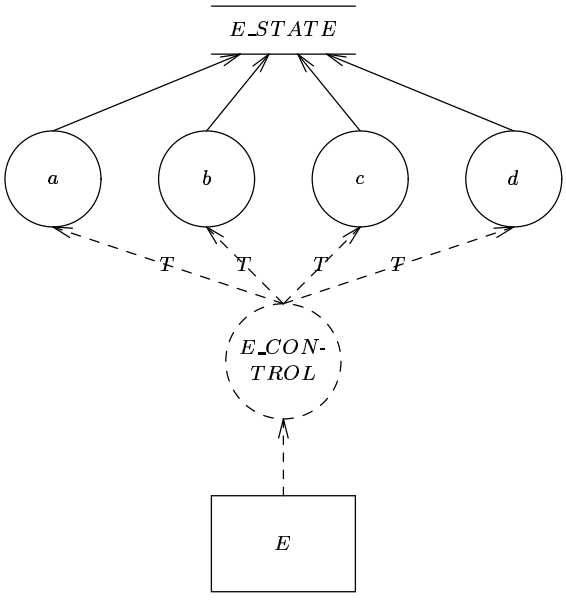

Figure 13: DF diagram corresponding to the PSD of figure 11.

with the state of the control process and the contents of the data store, respectively. Assuming that in the PSD, E is an entity type with possibly many instances, the external entity E in the DF diagram represents an arbitrary instance of this type, and the data store E_DATA contains the data state of each instance. The control process in figure **??** is specified by a state transition diagram that corresponds to the PSD of figure 11. It accepts events from an external entity E and in response triggers the appropriate data transformation, which in turn updates the data state of the appropriate instance.

From this brief comparison we can conclude the following that without control processes, DF diagrams do not represent the behaviour of a system at al. With the addition of control processes, DF diagrams can represent the same behaviour that can be represented by PSDs, but in a less perspicuous way, because information is spread over different parts of the diagram.

Object-oriented versus data-oriented modularisation. 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 don't 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 modularisation.*

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 modularisation.*

A consequence of this difference in modularisation 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 [5], 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.

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 response of the transformation to input. Such a complex transformation is therefore a reactive system. DF modelling offers no guidelines for the specification of such reactive complex transformations other than the usual one of loose coupling with other transformations. The modeler can therefore specify reactive transformations whose behaviour depends in an arbitrary complex way on its past. This makes DF models unnecessarily hard to understand.

By contrast, in JSD models of the UoD, all reactive systems are either JSD entities or function processes, and it is perfectly clear what these stand for. All subsystems of the initial system network are reactive subsystems that are surrogates for real-world objects. These are easy to understand, or at least as hard or easy to understand as the UoD counterparts of the reactive systems are.

The only other kind of reactive system in a JSD model is a function process, and this too has a clear intuitive semantics in terms of required system functions. In terms of structured design guidelines, reactive systems that represent real world entities or that are function processes have the maximal degree of cohesion, viz. *functional cohesion* [33, 40]. According to these guidelines, JSD models thus have the best kind of modularity.

4.3 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 an *event* which occurs in the environment of the system, and 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 [15, 30, 53]. 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 an 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 as well as in JSD (action allocation), that 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 Shlaer/Mellor method use DF models to represent system processing. In the next subsection, it is analysed how this is done in OMT.

5 Object Modelling technique (OMT)

In this section, only a brief analysis of OMT is given. A more detailed analysis of OMT is given in a separate paper [51]. OMT represents a system by using three models [34].

- 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 [16, 17],

which are an extension of finite state machines with, among others, the ability to represent substates, 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 recognise an advanced version of the combined JSD/ER approach. The PSDs of JSD can be represented equivalently by finite state automatons, 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. suggest that a specialised life cycle should be such that the generalised life cycle from which it inherits, should be retrievable from it by projection [34, page 111]. They do however not state what the projection of a state chart is. Recently, some approaches to life cycle specialisation have been proposed that make the concept of life cycle projection more precise [26, 37, 29].

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 appears elsewhere in the object-oriented literature as that between semantic classes and application classes [32] and between entity objects and control objects [24].

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 summarised 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 [34, pages 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 action, and the effect of one action may be specified by several data transformations.

If we replace "object class" with "JSD entity type", then figure **??** 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 modularisation, and this does not cohere well with the object-oriented modularisation of the rest of the model. In fact, it is this difference in modularisation 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. [10], it

is preferable 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 [10] and in a number of formal methods that are currently under development, such as Troll [25, 18, 36] and MCM [50, 48].

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 Shlaer/Mellor [39] and Fusion [10] 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 modularise the system. In an object-oriented approach, the classes and their inheritance hierarchy show an object-oriented modularisation 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 organised in a **function decomposition tree** as is well-known from Information Engineering (figure 14). 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 modularisation 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 modularisation. The reason why this can be done is that the two ways of modularisation are orthogonal to each other — 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 column of the tree. This makes the function decomposition tree different from modularisation constructs like subjects in the Coad/Yourdon method [9], subsystems in the method of Wirfs-Brock et al. [52], and the ensembles defined by De Champeaux [7]. 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.

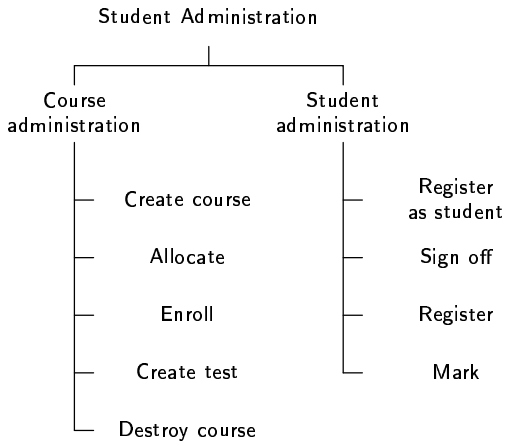

Figure 14: A possible function decomposition tree of the student registration system.

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, viz. 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 modularisations 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 specification [50, 48].

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, Shlaer/Mellor 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 Shlaer/Mellor): 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

[1] B. Alabiso. Transformation of data flow analysis models to object oriented design. In N. Meyrowitz, editor, *Object-Oriented Programming Systems, Languages and Applications, Conference Proceedings*, pages 335–353. ACM Press, 1988. SIGPLAN Notices, volume 23.

[2] S.C. Bailin. An object-oriented requirements specification method. *Communications of the ACM*, 32:608–623, 1989.

[3] C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach.* Benjamin/Cummings, 1992.

[4] A. Birchenough and J.R. Cameron. JSD and object-oriented design. In J. Cameron, editor, *JSP & JSD - The Jackson Approach to Software Development*, pages 292–304. IEEE Computer Science Press, second edition, 1989.

[5] G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12:211–221, 1986.

[6] J.R. Cameron. An overview of JSD. *IEEE Transactions on Software Engineering*, SE–12:222–240, 1986.

[7] D. de Champeaux and P. Faure. A comparative study of object-oriented analysis methods. *Journal of Object-Oriented Programming*, pages 21–33, March/April 1992.

[8] P.P.-S. Chen. The entity-relationship model – Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[9] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press/Prentice-Hall, 1990.

[10] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The FUSION Method.* Prentice-Hall, 1994.

[11] J.F. Costa, A. Sernadas, and C. Sernadas. *OBL-89 User's Manual, version 2.3.* Instituto Superior Técnico, Lisbon, May 1989.

[12] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press/Prentice-Hall, 1978.

[13] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings, 1989.

[14] M. Eva. *SSADM Version 4: A User's guide.* McGraw-Hill, 1992.

[15] S. Goldsmith. *Real-Time Systems Development*. Prentice-Hall, 1993.

[16] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[17] D. Harel. On visual formalisms. *Communications of the ACM*, 31:514–530, 1988.

[18] T. Hartmann, G. Saake, P. Hartel, and J. Kusch. Revised version of the modelling language TROLL (TROLL version 2.0). Technical Report 94-03, Abt. Datenbanken, Tech. Universität Braunschweig, P.B. 3329, Braunschweig, Germany, April 1994.

[19] D. Hatley and I. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House, 1987.

[20] R. Helm, I.M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In N. Meyrowitz, editor, *Conference on Object-Oriented Programming: Systems, Languages and Applications/European Conference on Object-Oriented Programming (ECOOP/OOPSLA '90)*, pages 169–180, Ottawa, October 21-15 1990. *Sigplan Notices* 15(10), October 1990.

[21] P. Hsia, A. Davis, and D. Kung. Status report: requirements engineering. *IEEE Software*, 10(6):75–79, November 1993.

[22] M. Jackson. *Principles of Program Design*. Academic Press, 1975.

[23] M. Jackson. *System Development*. Prentice-Hall, 1983.

[24] I. Jacobson, M. Christerson, P. Johnsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Prentice-Hall, 1992.

[25] R. Jungclaus, G. Saake, and C. Sernadas. Formal specification of object systems. In S. Abramsky and T.S.E. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91)*, volume 2, pages 60–82. Springer, 1991. Lecture Notes in Computer Science 494.

[26] A. Lopes and F. Costa. Rewriting for reuse. In *Proceedings ERCIM Workshop, Nancy, November 2-4*, pages 43–55. INRIA, 1993.

[27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent System Specification*. Springer, 1992.

[28] J. Martin. *Strategic Data-Planning Methodologies*. Prentice-Hall, 1982.

[29] J.D. McGregor and D.M. Dyer. Inheritance and state machines. *Software Engineering Notes*, 18(4):61–69, 1993.

[30] S.M. McMenamin and J.F. Palmer. *Essential Systems Analysis*. Yourdon Press/Prentice Hall, 1984.

[31] Michael Jackson Limited. *JSD Course Notes*, 1986.

[32] D.E. Monarchi and G.I. Puhr. A research typology for object-oriented analysis and design. *Communications of the ACM*, 35(9):35–47, 1992.

[33] M. Page-Jones. *The Practical Guide to Structured Systems Design*. Prentice-Hall, 2nd edition, 1988.

[34] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design*. Prentice-Hall, 1991.

[35] M. Ryan, J. Fiadeiro, and T. Maibaum. Sharing actions and attributes in modal action logic. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 569–593. Springer, 1991. Lecture Notes in Computer Science 526.

[36] G. Saake. *Objektorientierte Spezifikation von Informationsystemen*. Teubner, 1993.

[37] G. Saake, P. Hartel, R. Jungclaus, R.J. Wieringa, and R.B. Feenstra. Inheritance conditions for object life cycle diagrams. In U.W. Lipeck and G. Vossen, editors, *Formale Grundlagen für den Entwurf von Informationsystemen*, pages 79–88. Institut für Informatik, Universität Hannover, Postfach 6009, D-30060, Hannover, May 1994. Informatik-Berichte Nr. 03/94.

[38] E. Seidewitz and M. Stark. Toward a general object-oriented software development methodology. *ADA Letters*, 7(4):54–67, july/august 1987.

[39] S. Shlaer and S.J. Mellor. *Object Lifecycles: Modeling the World in States*. Prentice-Hall, 1992.

[40] W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Systems Journal*, 13:115–139, 1974.

[41] V.C. Storey and R.C. Goldstein. A methodology for creating user views in database designs. *ACM Transactions on Database Systems*, 13(3):305–338, September 1988.

[42] A. Sutcliffe. *Jackson System Development*. Prentice-Hall, 1988.

[43] D.A. Taylor. *Object-Oriented Technology: A Manager's Guide*. Servio Corporation, 1420 Harbor Bay Parkway, Alameda, CA 94501, 1990.

[44] P.T. Ward. How to integrate object orientation with structured analysis and design. *IEEE Computer*, pages 74–82, March 1989.

[45] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems*. Prentice-Hall/Yourdon Press, 1985. Three volumes.

[46] R.J. Wieringa. A formalization of objects using equational dynamic logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *2nd International Conference on Deductive and Object-Oriented Databases (DOOD'91)*, pages 431–452. Springer, 1991. Lecture Notes in Computer Science 566.

[47] R.J. Wieringa. Object-oriented analysis, structured analysis, and Jackson System Development. In F. van Assche, B. Moulin, and C. Rolland, editors, *Object Oriented Approach in Information Systems*, pages 1–21. North-Holland, 1991.

[48] R.J. Wieringa. A method for building and evaluating formal specifications of object-oriented conceptual models of database systems (`MCM`). Technical Report IR-340, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.

[49] R.J. Wieringa. *Requirements Specification: A Framework for Understanding.* Wiley, To be published.

[50] R.J. Wieringa and R.B. Feenstra. The university library document circulation system specified in LCM 3.0. Technical Report IR-343, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1993.

[51] R.J. Wieringa, R. Jungclaus, P. Hartel, G. Saake, and T. Hartmann. OMTROLL — Object Modeling in Troll. Proceedings of the International Worskshop on Information Systems — Correctness and Reusability (IS-CORE'93), Udo W. Lipeck and G. Koschorrek (eds), pages 267–283. Institut für Informatik, Universität Hannover, Postfach 6009, D-30060, Hannover., September 1993.

[52] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

[53] E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, 1989.

Contents