Jackson System Development, Entity-Relationship Analysis and Data Flow Models: A comparative study

R.J. Wieringa*

January 17, 1994

Abstract

This paper compares JSD with ER modelling and with data flow modelling. It is shown that the JSD method can be combined with ER modelling and that the result is a richer method than either of the two. The resulting method can serve as a basis for a practical object-oriented modelling method and has some resemblance to parts of well-known methods, like OMT. It is also argued that JSD and data flow modelling rest on opposite philosophies and cannot be combined in one modelling effort. This is illustrated by transforming a JSD model into a data flow model and listing the differences between the models. The results of this analysis are extrapolated to object-oriented models.

1 Introduction

In recent years, there has been a rising interest in the possibility to combine different conceptual modelling methods. For example, the possibility to integrate object-oriented modelling with Structured Analysis has received the attention of a number of researchers [1, 3, 20, 26, 28]. The possibility to combine Jackson System Development (JSD) with an object-oriented approach has also roused interest [5]. The possibility to combine JSD modelling with Entity-Relationship (ER) modelling is briefly discussed by Sutcliffe [25], but is not studied there in detail. A comparison of JSD with other methods is useful for at least the following reasons.

JSD plays an interesting role in these comparisons, for it This makes comparisons between JSD and other methods relevant in two ways.

- First, JSD shares with object-oriented methods like OMT [19] and the Shlaer/Mellor life cycle modeling method [21] the emphasis on the encapsulation of state and behaviour in objects. (A difference between JSD and object-oriented methods is that JSD is weak in its structural object model.) This means that to the extent that the results of comparative studies depend on this encapsulation property only, these results are applicable to object-oriented methods.
- Second, because JSD is used in commercial environments, the results of comparative study can be used to enhance JSD in real-life projects by combining it, where possible, with other methods, and to avoid trying combinations which comparative study has revealed to be impossible.

In this paper, I argue that JSD can be fruitfully combined with ER modeling but should not be combined with data flow modeling. In section 3, it is shown in detail how ER models and JSD models

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

can be combined, and which (slight) adaptations must be made to certain concepts in both methods to achieve such a combination. The results of this section may be relevant for practical applications.

The possibility to integrate JSD models with data flow models in a coherent way has, to my knowledge, not yet been studied before. It is argued in section 4 that JSD models and data flow models rest on opposite philosophies that do not combine well, because they rest on opposite philosophies. Bringing out the reasons why these models cannot be combined enhances our understanding of both kinds of models. It explains, for example, in which way the transition from data flow models in the early stages of SSADM to the use of JSD-like entity life histories in the later stages, may cause conceptual problems [11]. In addition, as argued in section 5, these reasons for incomatibility carry over to modern object-oriented methods and show why using data flow models in combination with object-oriented models may cause problems.

2 Jackson System Development

Although there are several versions of JSD [7, 15, 17, 25], their differences do not concern us here. In this section, I describe what is common to these different versions.

2.1 UoD models

JSD distinguishes models of a database system (DBS) from models of the universe of discourse (UoD) represented by that database system. A UoD model represents the world as a set of communicating entities, each with its own life cycle. Entities have local state and behavior, and they communicate by sharing actions. To make a DBS model, we duplicate the UoD model, so that we have two copies of it, called the *level 0* and the *level 1* model. The level 0 model is still a model of the UoD, but the level 1 model is the initial version of a DBS model. This is because the DBS must register data about the UoD, so that it will contain a "shadow", which we will call a *surrogate*, for each represented UoD *entity*. Each surrogate and each entity is a process, and I will speak of entity- and surrogate processes. In fact, the level 0 and level 1 models are completely identical in their appearance. Their difference lies in the eye of the beholder, who interprets the level 0 model as a model of the UoD and the level 1 model as the initial version of the DBS. The level 1 model is extended to a model of DBS functions, as illustrated in the next subsection.

First, as an example of UoD models, I give a very simple model of the UoD of a student administration. 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. Figures 1, 2 and 3 give a model of this UoD.

Following Sutcliffe [25], I call the diagrams in these figures *process structure diagrams* (PSDs). Each PSD is a tree in which the root is labeled by the name of an entity type and 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 (*) or small circle (o) as marker. An asterisk represents an iteration and a small circle represents a choice. Unmarked boxes represent left-to-right occurrence of processes or actions. A PSD for entity type T represents the life cycle that each of the instances of T goes through. All instances of T thus have an isomorphic life cycle, although at any moment, different instances can, and generally will be in different states.

Thus, a $STUDENT$ life cycle starts with an action called *register_as_student* and ends with an action $sign_off$. In between, there is an iteration over a choice out of three actions, *enroll*, $register$ and $mark$. The initiative of these actions is not represented. It is merely stated that these actions are performed or suffered in the life of any $STUDENT$.

There is a common action between $STUDENT$ and $COURSE$, viz. *enroll*. Looking at the PSD for $COURSE$, we can see that this enforces the constraints that a $STUDENT$ can only enroll for

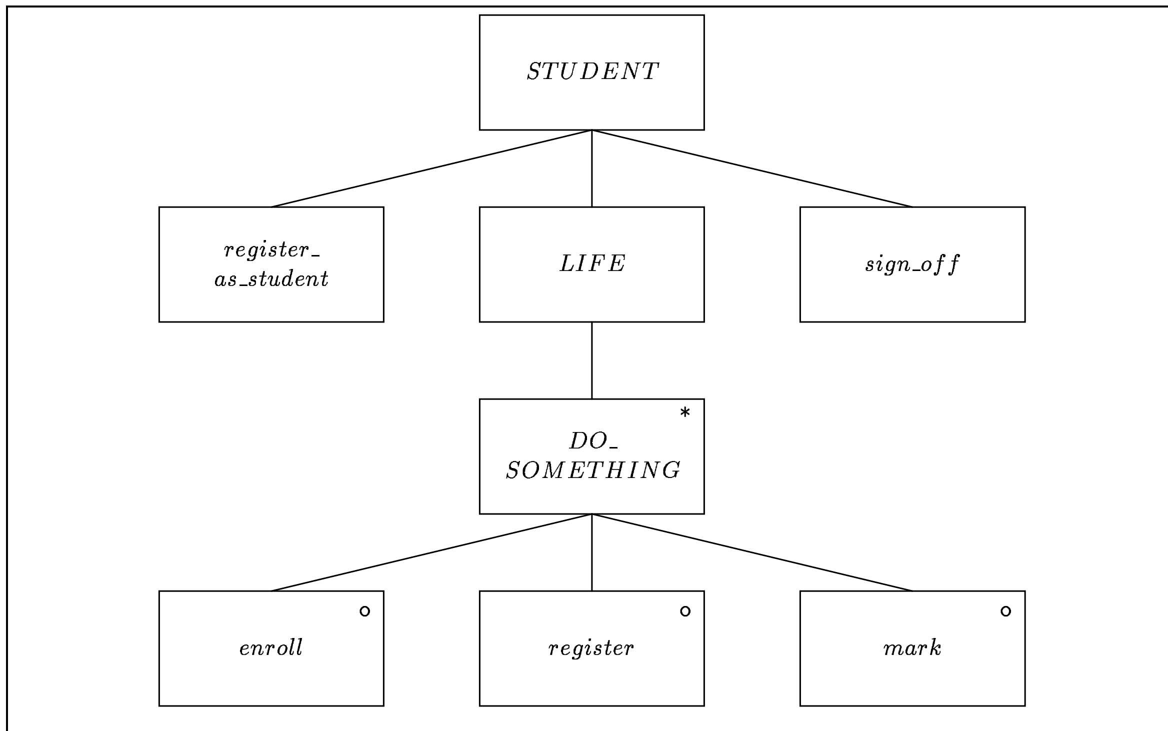

Figure 1: PSD for a simple *STUDENT* life cycle.

an existing *COURSE* (because *create_course* must have taken place) and that the *COURSE* must have been *allocated* to a slot in the timetable. (This meaning of the *allocate* action cannot be read from the PSD but must be specified separately.) As another example of communication, the *register* and *mark* actions are shared by the PSDs of *STUDENT* and *TEST*. Looking at these PSDs, we can see that a *STUDENT* can receive a mark for a *TEST* only after registration for the test has closed. Note that in this model, a *STUDENT* can receive several marks for one *TEST*, which is not true in reality. This will be repaired when we combine the PSDs with an ER diagram.

As stated earlier, the PSD's jointly are a model of the UoD as well as an initial version of a model of the DBS. They represent the life cycles of entities in the UoD as well as of the surrogates for these entities in a DBS. We next show how in JSD the level 1 model is extended to a model of DBS functions.

2.2 DBS models

The level 1 model is an initial version of a model of the DBS. It contains a surrogate process for each real-world process. Where a JSD entity in the UoD suffers or performs an *action*, its surrogate in the DBS will suffer an *update* that corresponds to the action. The level 1 model is extended to a more complete model of the DBS by adding specifications of system *functions*. JSD distinguishes three kinds of functions, input, output and interactive functions. An *input function* accepts data about the UoD and updates the appropriate level 1 processes, an *output function* reports about the state of the DBS, and an *interactive function* is a kind of trigger that, when a certain state of the DBS occurs, immediately updates the DBS. Functions are processes, just like UoD entities and DBS surrogates are processes, and their structure can be represented by PSDs.

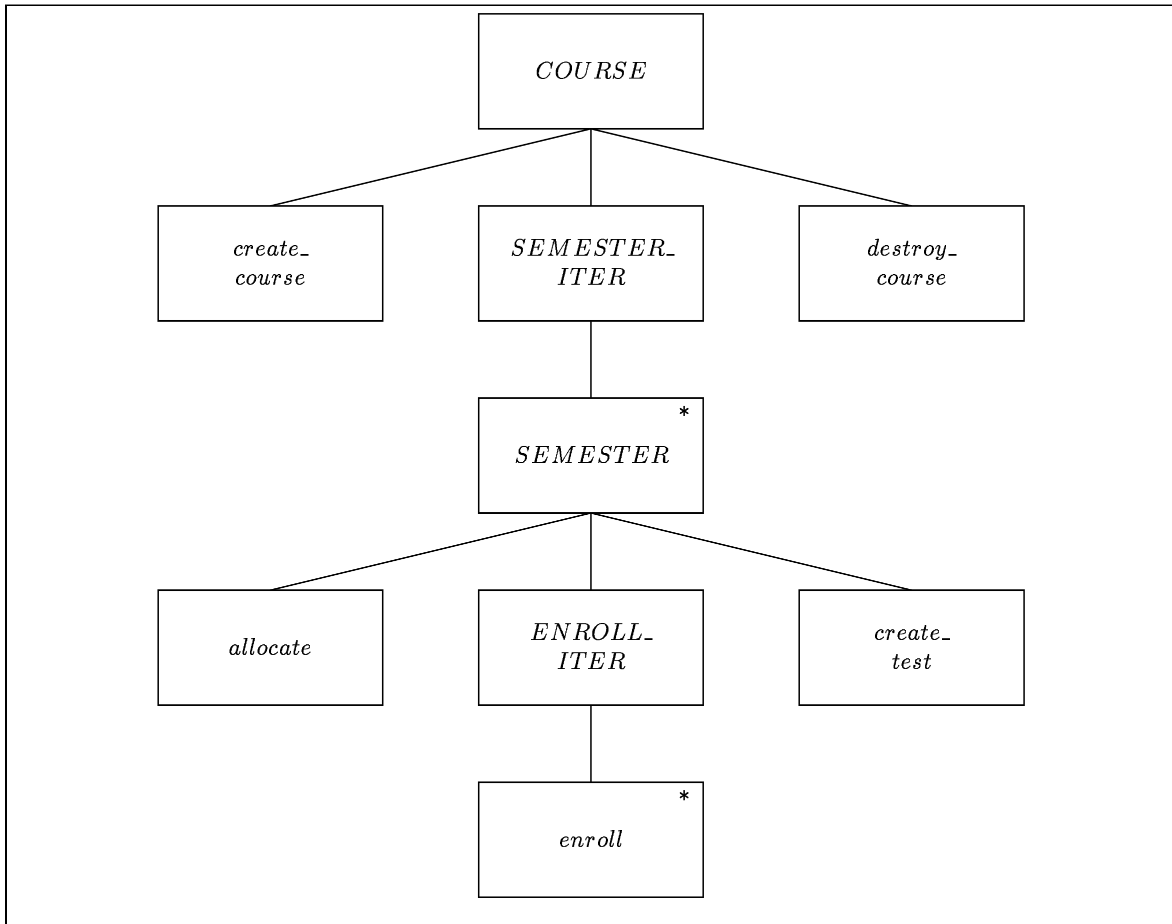

Figure 2: PSD for a simple $COURSE$ life cycle.

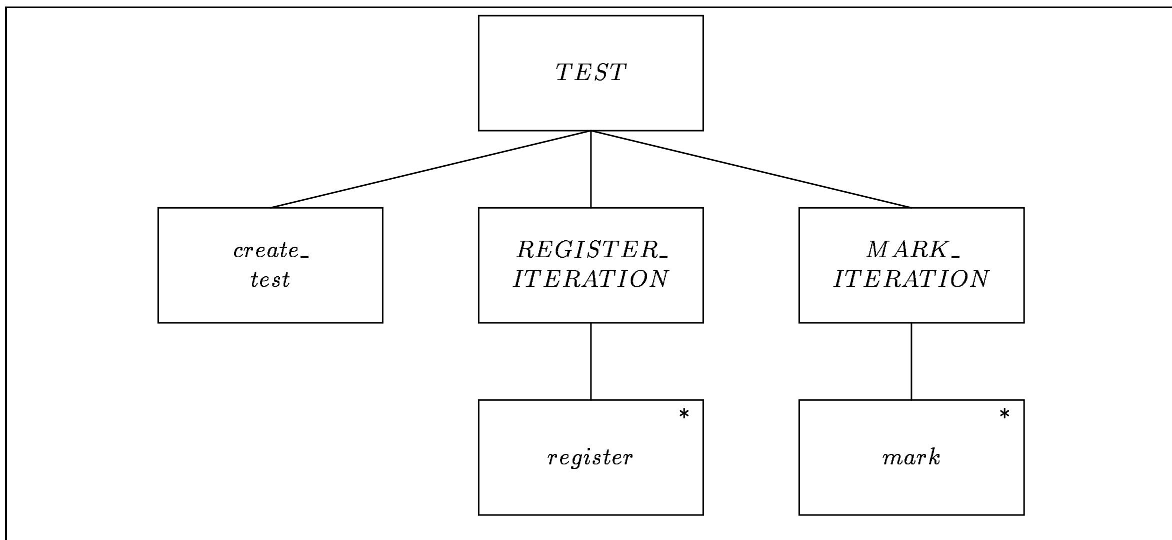

Figure 3: PSD for a simple $TEST$ life cycle.

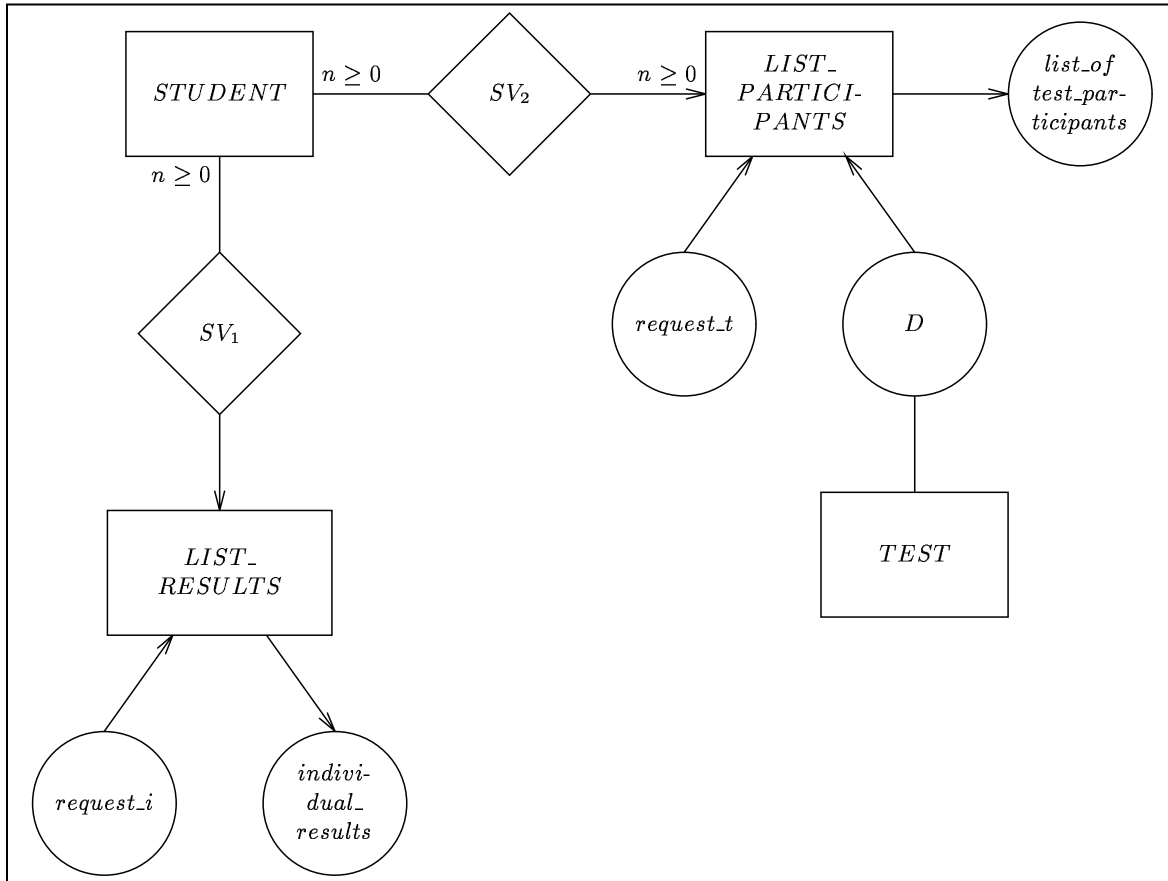

Figure 4: A simple system network.

Functions communicate with surrogate processes, and with each other, not through common actions, but through *process connections*. By connecting function processes to surrogate processes and/or each other, we get a graph in which the nodes are processes and the edges process connections. This graph is called the *system network* of the DBS. There are three kinds of process connections in JSD. *Data stream* connections are unbounded first-in first-out buffers between two processes, and can be used to realise asynchronous communication. 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. State vector connections realise synchronous communications in which the observed process is not disturbed. 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. The observed process is locked from access by other processes from the moment that the observation is made to the moment that the message is received. If no message needs to be sent, the observed process is released.

For example, suppose that we want to specify two output functions: *LIST_RESULTS* should, upon request, produce the results of a student, and *LIST_PARTICIPANTS* should, upon request, list all participants of a test. Figure 4 contains a system network for these functions. The boxes labeled *STUDENT* and *TEST* represent types whose instances are surrogates (which are in the DBS) and the other two boxes represent function processes of the DBS. Circles represent data streams, diamonds represent state vector connections. *LIST_RESULTS* takes as input a data stream with

requests, looks at the state of student surrogates in the system, and produces a data stream with a list of aggregate results obtained by the students. The structure of the two data streams and of the data passing through the state vector connection should be specified in a separate documentation of the network. The structure of the $LIST_RESULTS$ process must be specified by means of a PSD, not shown here. The cardinality $n \geq 0$ says that one $LIST_RESULTS$ instance is connected by SV_1 to arbitrary many $STUDENT$ instances. Absence of cardinality information stands for a cardinality of exactly 1, so for each $STUDENT$ there is exactly one $LIST_RESULTS$ process. (In fact, there is altogether only one $LIST_RESULTS$ process, whose task is to handle $request_i$ inputs.)

The $LIST_PARTICIPANTS$ reads a request from a data stream to list all registrations for a test, and then reads all registrations for the test from another data stream. To get the names of the registered students, the appropriate $STUDENT$ state vectors are scanned. 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. Note that there is one $LIST_PARTICIPANTS$ instance for each $TEST$ but that each such instance may scan arbitrary many $STUDENT$ instances.

A DBS model consists of a system network in which the nodes represent entity types or function processes and the edges represent process connections. All nodes have a process structure that must be specified by means of a PSD and all connections must be specified by giving the data structures passing through them. The network interfaces with the external world through "dangling data streams" that provide it with input and receive output. The external entities to whom these dangling data streams are connected, are not shown.

3 Combining JSD Models with ER Models

There are four basic constructs in ER models: entities, relationships, attributes and cardinality constraints. We first look at the correspondence between these concepts and concepts used in JSD models of the UoD, i.e. in level 0 models. There is nothing corresponding to an ER cardinality constraint in JSD models of the UoD, so we ignore these for the moment. However, the other three concepts can be correlated with JSD concepts.

3.1 JSD Entities and ER Entities

In JSD, an entity is an object that

1. exists in the UoD,
2. is capable of performing or suffering actions,
3. is an individual, and
4. is capable of being uniquely named [15, page 66].

There is no clear-cut definition of what an ER entity is. It has been defined as an individual in the UoD by Chen [8, page 10] and by Elmasri and Navathe [12, page 40], but as a *class* of things in the UoD by Batini et al. [4, page 31]. Elsewhere, Elmasri and Navathe speak about entities as things in the DBS instead of in the UoD [12, page 42]. Some authors again speak of ER entities as *classes* of things in the DBS [24]. Here, I take the view that an ER entity is an *individual*, not a type, and that it must be an *observable* individual. By this I mean that an ER entity must be able to interact with its environment, where each possible interaction is an event that occurs in time. This view of ER entities excludes abstract things like numbers and truth values from being regarded as ER entities, for these abstract entities are not observable. (We use observable symbols to *represent* these abstractions.)

In order to combine ER and JSD models, we should fix on a single concept of an entity that is compatible with both kinds of models. We look at the four criteria for JSD entities to see which of these four we can use in a combined approach.

1. The criterion that entities must exist in the UoD, not in the DBS, is fundamental to JSD, so we keep this in a combined approach.

2. The requirement that a JSD entity must be capable of performing or suffering actions, is also fundamental to JSD. However, we always model the world at a certain level of abstraction, and we may choose a level of abstraction at which no actions occur at all in the life of an entity. For example, in a certain model used by a travel agency we may decide to have an entity type $COUNTRY$, even though we may decide not to model any significant actions in the life of a $COUNTRY$. This is not because countries cannot perform or suffer actions: they can wage wars, change their borders, accept or refuse refugees etc. We decide not to model any significant actions in the life of a $COUNTRY$ because we choose a level of abstraction at which we ignore those actions. In an ER model, this is no problem, as long as we assign descriptive attributes to $COUNTRY$, such as *name*, *area*, *population*, etc. In a combined ER/JSD approach, we can safely drop the criterion that entities should *be modeled as* capable of performing or suffering actions. It is sufficient that they can do so in reality, even if we do not represent any of those actions in the model.

3. The requirement that a JSD entity must be capable of being regarded as an individual just means that we do not represent the type/instance relation in the model. For example, in a model of a library we can choose to have an entity type $TITLE$, whose instances are documents, as well as an entity type $DOCUMENT$, whose instances are physical copies of documents. Both types have instances that can be regarded as individuals, but each $TITLE$ instance is in addition a subtype of $DOCUMENT$. There is no construct in JSD (nor in ER) to represent the fact that $TITLE$ is a metatype of $DOCUMENT$. We keep the individuality criterion in a combined approach.

4. The requirement that a JSD entity must be capable of being uniquely named is important, for it is its unique identifier that allows us to *identify* an entity among other entities, some of which may be in the same state, and it also allows us to *re-identify* an entity when it has changed state [29]. Note that this criterion does not apply to entities, but to ourselves: *we* must be able to assign unique identifiers to entities. It is important that for each entity type, an attribute is defined that can function as identifier for instances of the type. We keep this criterion in a combined approach.

The conclusion is that, in a combined modeling approach, an entity is an individual in the UoD that is capable of performing or suffering actions. We do not require these actions to be represented in the model, but we do require the definition of an identifier attribute for each entity type.

3.2 Relationships

The JSD view of entities includes relationships. A relationship between entities in the UoD is itself an individual in the UoD. Relationships are capable of performing or suffering actions. For example, a $LOAN$ relationship between a library $MEMBER$ and a $DOCUMENT$ may be *extend*ed. Relationships are also capable of being uniquely identified. For example, a $LOAN$ instance could be identified by giving the identifiers of the $MEMBER$ and the $DOCUMENT$ that are related by the $LOAN$ instance. This would give us a composite identifier for relationships, consisting of one identifier for each component entity.

If we want, we can still distinguish entity– and relationship types in an ER diagram, but because in a combined approach a relationship is just a particular kind of JSD entity, we could still specify a life cycle for a relationship. We could for example specify a PSD for the life of $LOAN$ instances, consisting of a start action $borrow$, an iteration over zero or more $extend$ actions, and terminating in a $return$ action. This is illustrated later.

3.3 Attributes

A JSD entity attribute is a local state variable of a JSD entity [15, page 241], i.e. any observation that can be made of an entity through a state vector connection. We will keep this attribute definition in a combined approach. Every entity has at least two attributes, viz. its position in its PSD and its identifier. Action parameters are called attributes in JSD, but we will not follow this practice in a combined approach. These is merely a terminological matter.

3.4 Using relationships to remember common actions

Relationships can stand for many different kinds of things, such as part-of relations, element-of relations, contractual obligations, permissions, authorisations, etc. Relationships can also stand for common actions, and it is here that a special correspondence with JSD models of the UoD can be found. Briefly, we can represent any common action between PSD's by a relationship between the entities that share the action. Consider figure 5, which I will call a *communication diagram.* This a a graph in which the nodes represent entity types and the edges common actions. Since the initiative of an action is not represented, the edges are undirected. Action parameters have been added to the diagram in figure 6, so that it is clear that the actions are shared by *instances*, not by types. The variables c, s and t can be thought of as the identifiers of an individual $COURSE$, $STUDENT$ and $TEST$, respectively. The variable m in $mark(s, t, m)$ stands for the mark received by s on t.

As an aside, note that a communication diagram is really a precursor of the system network. Where the system network represents the three types of process connections but ignores synchronous communication through common actions, the communication diagram represents common actions only. If we want an overview of all four kinds of communications, we could merge the two diagrams.

Now, suppose we want to remember the occurrence of each of the common actions, for example because we want our DBS to be able to answer questions about them. Then we can turn all of them into a relationship type, as shown in figure 6. I use the classical representation of relationship types by diamonds, but another representation could also have been used. The arrows leaving the relationship diamonds in the ER diagram represent projection functions that retrieve the components of the relationship. There is a superficial resemblance between the ER diagram and the system network diagram, because rectangles and diamonds are used in both. Such overloading of graphical symbols can only be avoided by introducing an different graphical symbol for each different concept, which would make the model hard to read. We can make the following observations about the correspondence between the communication diagram and the ER diagram.

- The event parameters end up as relationship components and –attributes. For example, as a relationship, $mark(s, t, m)$ has components s and t (these form a compound identifier) and attribute 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 $REGISTRATION$ 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 6 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."

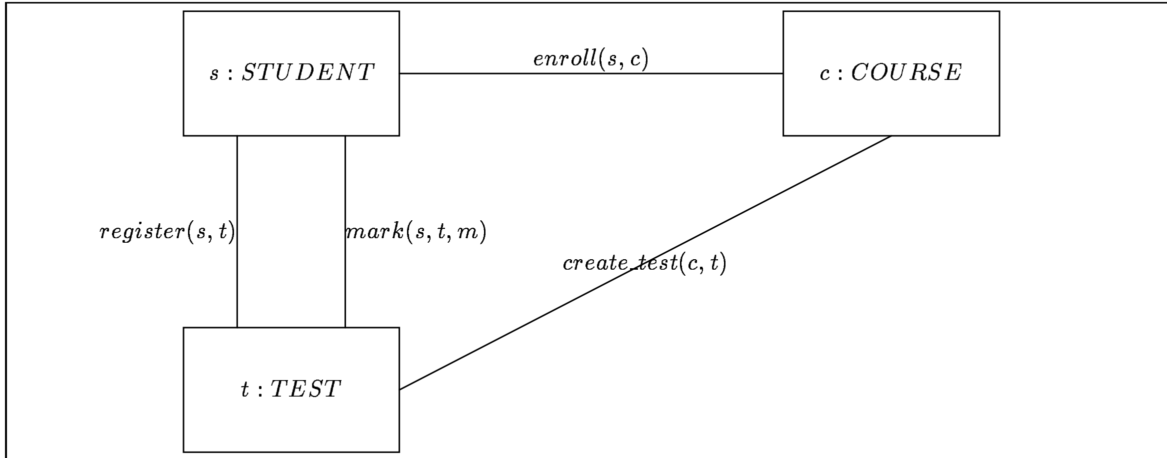

Figure 5: Communication diagram for the common actions between $STUDENT$, $COURSE$ and $TEST$ instances. For clarity, the action parameters are shown in the diagram.

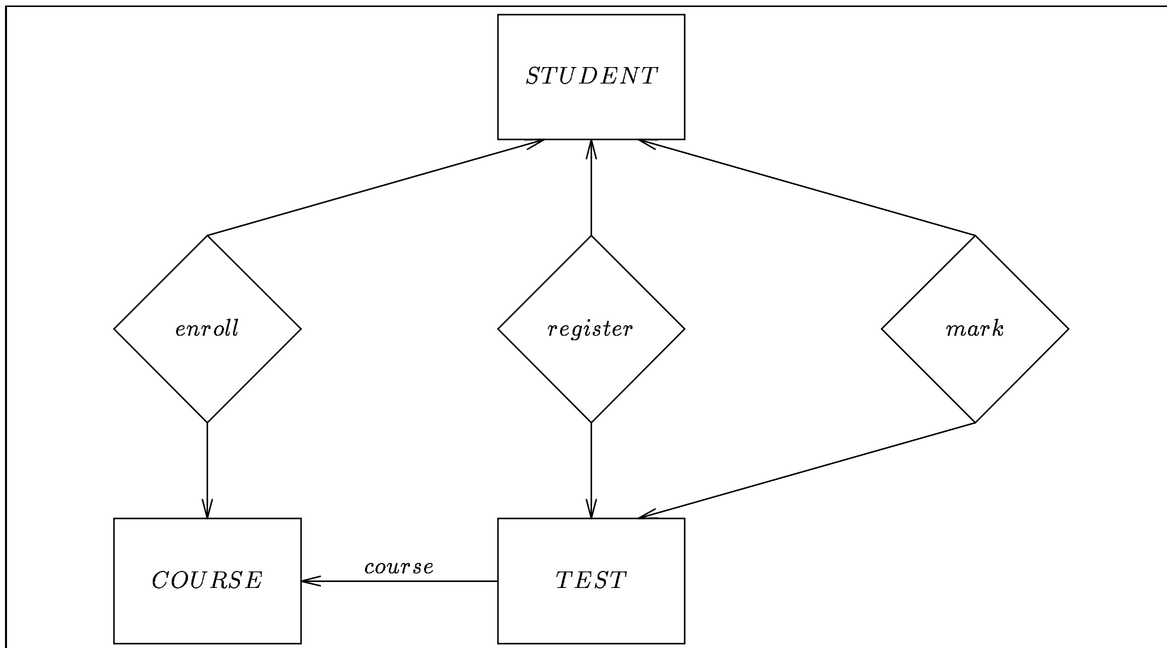

Figure 6: The ER version of the interaction diagram. Each common action has been turned into a relationship. The $create_test$ action has been turned into a functional relationship of $TEST$ called $course$.

- 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 5. 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 [15, 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. This attribute is a many-one relationship that corresponds to the creation event in the dependent entity.

- 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 *mark* 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 *mark* 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 *mark*.

To conclude, in a combined ER/JSD model we can represent the fact that we want to remember that a common action in the JSD model occurred by representing that action as a relationship in the ER model. The simplest thing to do is to add a relationship to the ER model with the same name as the common action in the JSD model. The components of the relationship will be the entities that share the common action. The other parameters of the action will appear as attributes of the relationship. If needed, we can also add a *date* attribute to the relationship, to represent the date and time of occurrence of the action.

Not all common actions need to lead to relationship types in an ER diagram. 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, corresponding to the *start* and *stop* actions. There is simply no need to remember the occurrence of these actions in the state of the DBS. As another example, *extend* is a shared action between library members and documents, but in this model occurrences of *extend* will not create new relationships between the two entities. Searching for common actions is a good heuristic for finding relationship types between entities, but it is not certain to lead to relevant relationship types. Conversely, as stated earlier, not all relationship types correspond to common actions. It is for example not natural to view part-of and element-of relationship types as common actions.

3.5 Using relationships to reduce action sharing

Once we use a relationship type to represent common actions, we can reduce the number of common actions in the JSD model. This reduces the complexity of the model and therefore increases the comprehensibility. To see how relationships can help reducing the number of common actions, consider the communication diagram for entity types in a JSD model of the UoD of the circulation desk a library, shown in figure 7. We do not show the PSD's for these entity types, but only the two common actions. $borrow_res$ is an action that terminates a $RESERVATION$ and starts a $LOAN$ life cycle, and *lose* is an action that prematurely terminates a $LOAN$ instance, halts the life cycle of a $DOCUMENT$ and creates a $FINE$ instance. Figure 8 shows an ER diagram of the JSD entities involved, where the actions allocated to the different entity– and relationship types have been written next to the boxes and diamonds for these types. (Other conventions could have been used, but this is not the point here.)

The cardinality information 0, 1 at the $DOCUMENT$ side of $LOAN$ means the following: For each existing $DOCUMENT$, there exist at any moment 0 or 1 $LOAN$ relationships. The absence of

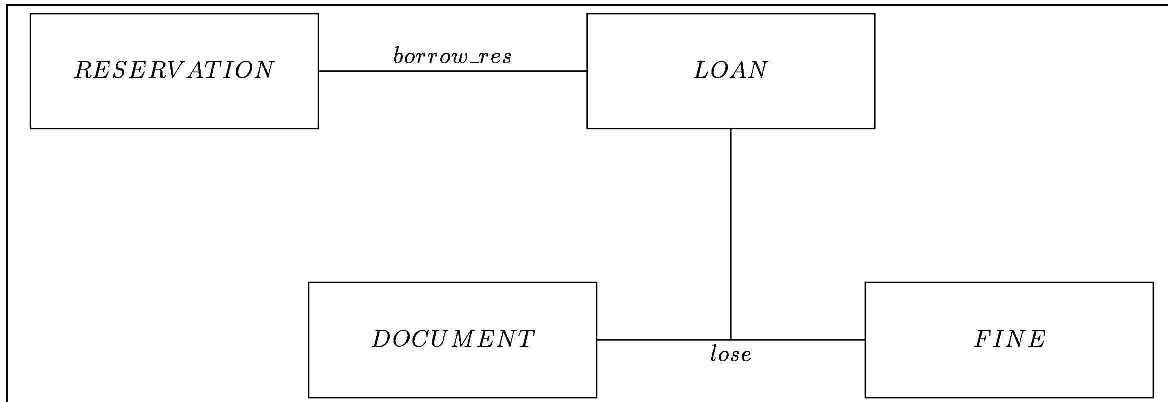

Figure 7: Interaction diagram of the circulation desk UoD.

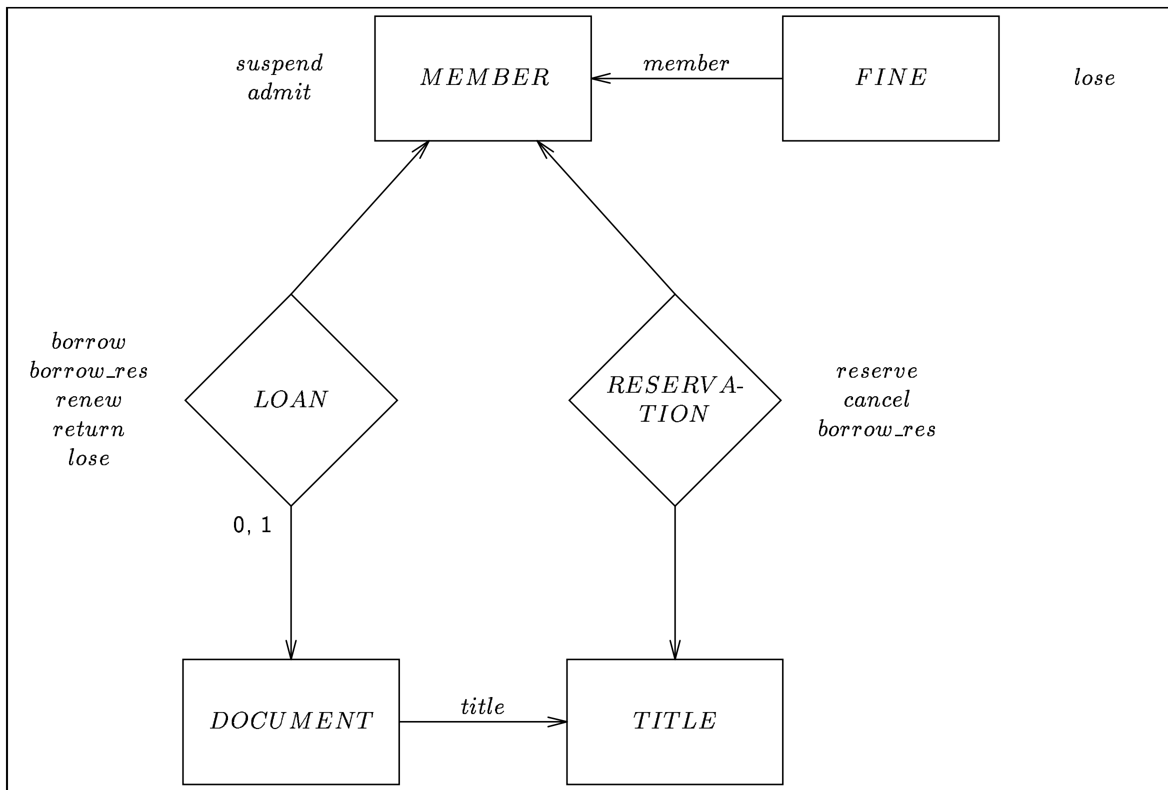

Figure 8: ER diagram of the JSD entity types in the JSD model of the circulation desk UoD. The actions allocated to the JSD entity types have been written next to the ER entities and relationships in the diagram.

a cardinality constraint at the $MEMBER$ side of $LOAN$ means that for each existing $MEMBER$, there can exist at any moment any number of $LOAN$ relationships.

Now, we could argue that $reserve(m,t)$ is an action shared by the member m and the title t, and should therefore be allocated to those to entity types in addition to it being allocated to $RESERVATION$. However, this would make the PSD's for the entity types more complicated, and it would make the communication structure of the diagram very heavy. This is the more so because almost every action in the diagram is subject to the same considerations: $borrow$, $extend$ and $return$ would be shared by $MEMBER$ and $DOCUMENT$, etc. The complexity of the resulting communication structure is a reason to try to reduce the number of communications.

Luckily, there is no reason to assign $reserve$ to $MEMBER$ and $TITLE$. Staying within JSD terminology, if we assign attributes $reserver$ and $reserved_title$ to $RESERVATION$, the $reserve$ action does not need to change any attribute of $MEMBER$ and $TITLE$ instances and we can answer any query about reservations by looking at $RESERVATION$ instances only. The query will just need a state vector connection with $RESERVATION$ instances. However, using now ER terminology, what we are doing in this way is treat $RESERVATION$s as relationships between $MEMBER$s and $TITLE$s. Now, any information concerning a reservation of a title t by member m is shared by m and t and is attached to the $RESERVATION$ instance $\langle m,t\rangle$. We do this with shared attributes, which reduces the number of attributes which need to be assigned to $MEMBER$ and $TITLE$, but we can do this also with shared actions, when then do not need to be assigned to $MEMBER$ and $TITLE$. This argument applies to all actions allocated to relationships. It leaves us with only a few common actions in the circulation desk model, $borrow_res$ and $lose$, as shown in figure 7.

I propose the following quidelines for allocating actions to entities in a combined ER/JSD model:

- We allocate an action to an entity if it needs to change the local state of that entity. For example, if $reserve$ would change the state of a $TITLE$, we should allocate it to $TITLE$ as well as to $RESERVATION$. This way, we respect a principle of *locality* in which an action can only change the state of the entities in whose life it occurs. This is an important principle in object-oriented modeling, where it is often called *encapsulation.*

- We allocate an action to an entity because want to enforce a sequencing by means of a PSD, and we allocate it to several entities if we want to enforce sequencing by means of common actions. We saw an example of this when 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.

- We allocate an action to an entity in such a way that we can answer queries about whether, or how often, the action occurred in the life of an entity. This guideline may tell us to allocate $reserve$ to $RESERVATION$ and not to $TITLE$ or $MEMBER$, just as it tells us to allocate $suspend$ to $MEMBER$.

3.6 Functions

We now turn to models of the DBS and ask in what way ER models and JSD models could be combined at this level. First, we note that one type of node is added to the entity type nodes in the system network: nodes that represent function processes. JSD distinguishes *short-running* functions, which are instantiated every time they are called, from *long-running* functions which are instantiated once and exist for as long as they may have to be activated. The $LIST_RESULTS$ function in figure 4 is a short-running function and the $LIST_PARTICIPANTS$ function is long-running. We can assume that the existence of instances short-running functions need not to be remembered, so that it is not useful to introduce an entity type in the ER model corresponding to a short-running function. A long-running function, on the other hand, could be made to correspond to an entity type in the ER

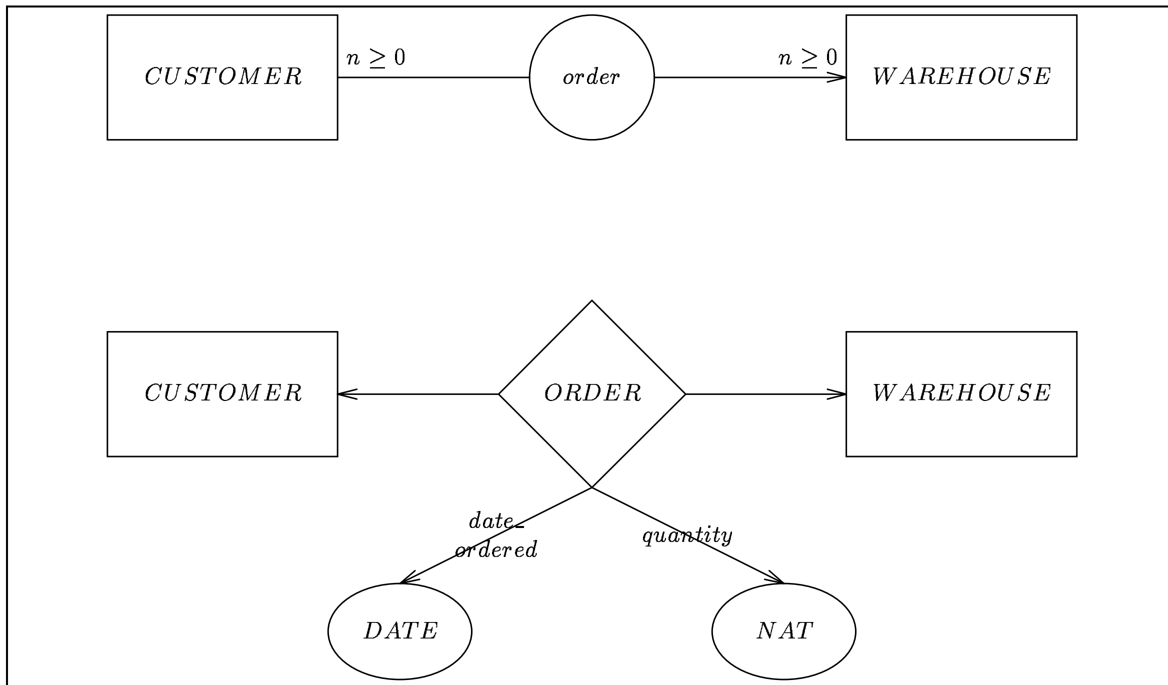

Figure 9: A data stream in a system network corresponds in an ER diagram to a relationship type. The data stream in addition contains the information that the contents are ordered according to the date of arrival in the buffer, and are to be read in that order. If the *order* data stream never contains duplicates, then at any moment, the contents of the data stream is the extension of the relationship type.

model, for we may have to remember the state of the function in between activations. There will then be one instance of this type in the DBS, and this instance does not correspond to an entity in the UoD but is a long-running system function.

3.7 Process connections, relationships and cardinalities

There are three kinds of connections that may connect function nodes with entity type nodes in the system network. Now, *state vector connections* correspond to *read* statements in the function processes and are hence invisible in an ER diagram. *Data stream connections*, on the other hand, may be thought to lead to relationship type in the ER diagram. The following example shows that we must be very careful with this.

Suppose we have an *order* data stream connecting two entity types, as shown in figure 9. (Data streams that connect entity types are not excluded from JSD models, but the argument does not hinge on this.) We translate this data stream into a relationship type as shown in the figure, where we add the attributes of the messages sent to the data stream as relationship attributes to the ER diagram.

This translation is not without assumptions nor without loss of information. First, the relationship does not contain the first-in first-out ordering information that is represented by data streams, which are, after all, queues of messages. This is solved in the figure by adding the *date_ordered* attribute to *ORDER*. Dynamics is not represented by the ER diagram, but with this attribute added we can at least set up the software to retrieve from the relationship the sequencing information that is stored in the corresponding data stream.

More seriously, the relationship contains a different kind of cardinality information than the system

network. What is counted by the cardinality in a data stream connection is the *number of JSD entities* that is connected to a data stream. So the system network in figure 9 says that one $WAREHOUSE$ instance is connected to 0, 1 or more $CUSTOMER$ instances. However, what is counted by a relationship cardinality in an ER diagram is the *number of different existing relationships* that is mapped to one entity. The absence of cardinality constraints for $ORDER$ means that for every existing $CUSTOMER$ we can have any number of $ORDER$ relationships and that similarly for every existing $WAREHOUSE$, we can have any number of $ORDER$ relationships.

Now, this is not what we want: The set of existing $ORDER$ instances can keep only one $order$ occurrence per $CUSTOMER$ and $WAREHOUSE$ instance. This is because the identity of an $ORDER$ instance is composed of the identity of the two component instances. Where an $order$ data stream can hold any number of consecutive occurrences of the $order$ action performed by the same customer for the same warehouse, the $ORDER$ relationship can hold only one at a time. The translation of the data stream into a relationship is meaning-preserving only under the assumption that each $CUSTOMER$ instance sends its next $order$ to the $WAREHOUSE$ instance only if there are no other $order$ instances in the data stream pending to be received by the $WAREHOUSE$ instance.

We can get around this by translating $ORDER$ into an entity type instead of into a relationship type. This requires giving each $ORDER$ instance an identifier that is not composed of $CUSTOMER$ and $WAREHOUSE$ instances, and this allows us to give different identifiers to orders placed by the same $CUSTOMER$ at the same $WAREHOUSE$.

To conclude, when combining an ER model with a JSD model it is best not to transform data streams into relationships but to transform them into entity types. The date and time when a message was sent to the data stream must still be added as an entity attribute. Note that the instances of the entity type now are *messages* that are sent to the data stream. When the data stream connects entity types, as in the $ORDER$ example, then the messages represent real-world messages. When they connect a function process with another process, they represent nothing in the UoD but realise a system function.

Turning to *controlled data stream connections*, finally, there is again no need to remember the fact that a message was sent through such a connection. The receiver is locked during the interaction, and after the interaction has taken place, there is no need to remember that it has taken place. In a combined ER/JSD approach, we therefore do not let any entity– or relationship type in the ER diagram correspond to controlled data stream connections.

To summarise, the combination of JSD models and ER models is feasible, and leads to a meaningful extension of either modelling technique. A combined model consists of an ER model, a process model (consisting of PSD's) and a system network (representing communications). It is advisable to include common actions in the system network, so as to have an overview of all communications. This partitioning of the model into three kinds of models is not unlike the approach found in object-oriented methods such as OMT [19] and Shlaer and Mellor's method [21]. I return to this later.

In a combined ER/JSD model of the DBS, entity types may have surrogates, a long-running function process or messages as instances; surrogates represent UoD entities, long-running functions do not. Messages may represent messages in the UoD or may realise system functions. Relationships can be used to represent common actions whose occurrence must be remembered, and they can also be used to reduce the number of common actions in the model.

4 JSD Models and Data Flow Models

We now turn to the comparison of JSD models with data flow models such as those described by DeMarco [10], Page-Jones [18], Yourdon [31] and others. I do this by transforming the JSD model

of the test registration administration into a data flow model. Since the JSD model consists of two parts, the UoD model and the DBS model, the data flow model will be built up in two stages.

4.1 Transforming PSDs into a data flow model

A data flow model consists of a data flow diagram (DFD), a data dictionary and minispecs. The DFD represents a system as a set of communicating subsystems, which are either data transformations or data stores. The data dictionary specifies the structure of the data passing through the flows and held in the stores, and the minispecs specify the logic of the data transformations. We look at the transformation of a JSD model into a DFD only. To transform a JSD model of the UoD into a level 0 data flow model, we must first extend the JSD model with an ER model as indicated in the previous section. This allows us, by the definition of relationship types that correspond to common actions, to indicate which communications must be represented as data stores.

The basic idea of transforming a JSD model in a DFD model is to put all state vectors of instances of entity– or relationship type T into a data store T_s, and turn all actions allocated to T into data transformations that access and update T_s. Furthermore, in the transformation we follow the guidelines given in JSD to transform a system network into a system implementation.

More in detail, the guidelines for transforming the JSD into a data flow model are then 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 DFD. 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* data flows to the data stores corresponding to each entity 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 also necessary to update the state and possibly other attributes, then use a bidirectional *read/write* data flow instead of a *read* data flow.

5. If a transformation corresponds to a common action in the JSD model that itself corresponds to a relationship in the ER model, then connect it to the data store corresponding to that relationship. Use a *read* or a *read/write* connection according to the criteria mentioned above, but choose a *write* connection if the action requires creation of a new record in this data store.

Figure 10 shows the result of applying these rules to the student administration model. The ER diagram represents the structure of the data in the data stores. We ignore the documentation of the DFD by means of a data dictionary and minispecs.

If we compare this DFD with the JSD model of the UoD, then we can observe the following differences.

Behaviour representation versus interface representation. All sequencing information is lost in the DFD. The DFD only shows which actions occur and how these interface with data stores and external entities. The JSD model, by contrast, shows behaviour in its PSDs. Interface information is

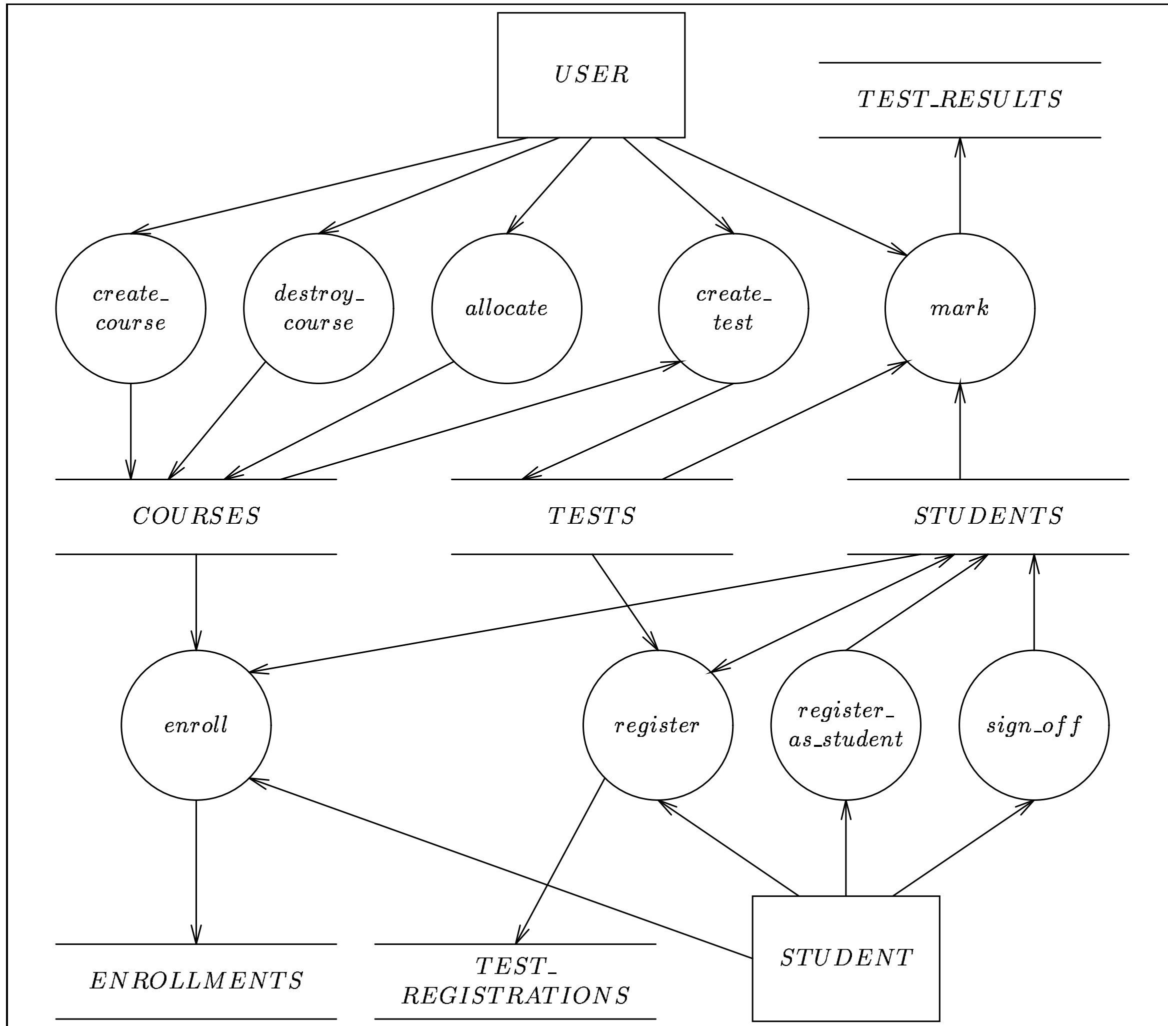

Figure 10: Transformation of the JSD model of the test registration UoD into a DFD. The input data flows carry the parameters of the action to which they are connected. This should be documented in the data dictionary, but this is not shown here.

not absent from JSD, but it is represented differently, viz. by common actions. These interfaces can be represented by a communication diagram.

We can preserve the behavior information in the transformation if we use control flow diagrams (CFDs), by extending the DFD's with control transformations [27]. The behavior information represented by the PSD of a JSD entity type T can then be represented by the state transition diagram of a control transformation C in the CFD. Initially, the data transformation corresponding to the initial action(s) of T would be enabled, and all others would be disabled. Each action in the life of an entity of type T would, in the control transformation C, have the effect to enable the next possible actions and disable all others. This control flow diagram would preserve the behavior information, but would do so in a less perspicuous way than the JSD model.

UoD-orientation versus DBS orientation. The JSD model is a model of the UoD, whereas the level 0 DFD is still a a model of the DBS, despite it being produced from a UoD model. This is visible by the fact that the DFD contains data stores, which are typical for a DBS, and by the fact that some JSD entities are duplicated as data stores and external entities. Conversely, the DFD also includes external entities, such as $USER$, that are not JSD entities because they do no exist in the UoD. They are present in the DFD because they provide the DBS with input.

Object-oriented versus file-oriented modularisation. The JSD model and the DFD 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 DBS are very different. The DFD 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. I call this *file-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 behavior. I call this *object-oriented modularisation*. A consequence of this difference in modularisation criteria is that in the level 0 data flow model, the state of an object is *separated* from its actions, for 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 [6], is *globally accessible* in the DFD. Any data transformation that accesses a data store, has access to all records in the data store. This contrasts with the localisation 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.

Reactive systems and objects. There is a consequence of this for the understandability of reactive systems in JSD models and data flow models. A *reactive system* is a system that, when it is ready to accept inputs (i.e. when it is ready with processing the previous inputs), has a memory of its past. The concept of reactive system was proposed by Manna and Pnueli [16]. A functional primitive in a data flow model is not a reactive system, for after it has transformed its input into output, it has no memory of past inputs. A JSD entity, on the other hand, is a reactive system, for it remembers past inputs (i.e. actions) in its current state.

Data flow models can be leveled, in which case a data transformation may be specified by another data flow diagram. Let us call such a transformation a *compound transformation*. Now, a data flow diagram for a compound transformation can contain data stores, and a compound transformation containing a data store is *reactive*. Input to a reactive data transformation may have a side effect on the local data stores of the transformation. There are no reactive transformation in our example

DFD, but in general such transformations can arise in a DFD because the DFD becomes complex and must be transformed into a leveled DFD. Our simple example already led us to a DFD of maximum complexity. In addition, we will see that reactive transformations can arise in a level 1 DFD, as the result of transforming a long-running function into a DFD.

Now, the problem is that reactive transformations are hard to understand, for the same reason that programming with side-effects may lead to programs that are hard to understand. If a reactive transformation receives an input twice, it may react differently to it, because the second time it receives the input, it may be in a different state from the state it was in the first time it received it.

Having side-effects is in itself not bad or good. In fact, the world we live in could not exist without side-effects, because state changes are nothing but side-effects of events. However, the side-effects of an input on a reactive transformation are generally hard to understand, because we do not know what the transformation stands for. All we know is that the transformation remembers some of its input. 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. We saw that all subsystems of the initial DBS model 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.

As explained below, the only other kind of reactive system in a JSD model is a long-running function, 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 long-running function processes have the maximal degree of cohesion, called *functional cohesion* [18, 23]. According to structured design guidelines, JSD models thus have the best kind of modularity.

In addition to having a clear intuitive meaning and good modularity of reactive systems in a JSD model, we also have an explicit representation of their behaviour by means of PSD's. This makes the meaning of the actions in the life of a reactive system easier to understand, because it give the dynamic context in which an action is to be understood.

4.2 Transforming a system network into a data flow model

To transform a system network of the DBS into a level 1 data flow model, we extend the DFD obtained so far with system functions as represented by the system network of the DBS. In the resulting DFD, there should be no data stores that have no outgoing data flows, such as *TEST_RESULTS* in figure 10. We use the following guidelines for the transformation.

1. Transform each **function process** into a data transformation.
    - If the function process is *long-running*, it may become a reactive transformation that is specified further by a DFD with its own data store. This data store holds one record, representing the state of the long-running function between activations.
    - If the function process is *short-running*, it has no memory between instantiations and it becomes a functional primitive.
2. **Data streams**.
    - If a data stream never holds more than one record at a time, transform it into a data flow.
    - If a data stream may contain more than one record at a time, transform it into a data store connected with a *write* to the sending process and a *read/write* to the receiving process. In the minispec of the sending process, we take care to send a time-stamp along with the written record. In the minispec of he receiving process, the *read/write* events are removals of records from the data store, that take the time-stamps into account.

3. Transform a **state vector** into a *read* data flow from the data store that holds the state vector of the observed entity, to the data transformation corresponding to the observing process.

4. Transform a **controlled data stream** into a data stream together with a state vector connection. The logic of *lock* and *release* must be encoded in the minispecs of the processes that communicate through the controlled data stream.

These guidelines have been motivated by the meaning of the constructs in both kinds of models. There is also some similarity with the guidelines for implementing JSD models [15] and with the guidelines for transforming a data flow model into a structured design [18].

Application of these guidelines to the system network for the test registration administration (figure 4) leads to an extension of figure 10. This figure is not shown, because it adds to the observation of only a few minor extra differences between JSD models and data flow models:

- **Process connections**. A system network contains three different kinds of process connections: state vector, data stream, and controlled data stream connections, a DFD contains only one: the data flow.
- **Cardinality of connections**. A system network contains cardinality information about process connections, a DFD does not.
- **Process structure specification**. A function process in a JSD model is specified by giving its process structure in a PSD. In a data flow model, it is specified by a minispec if it is primitive, or by another DFD if it is compound.

These differences are not big, because, in principle, the system network of JSD has roughly the same purpose as a DFD: show interfaces between processes.

We can conclude that the major differences between JSD models and data flow models are the explicit representation of behaviour in JSD, the UoD-orientation and the object-oriented modularisation of JSD, and its restriction of reactive systems to surrogates for real-world objects and to long-running function processes. This contrasts with the orientation on system interfaces, the orientation on the DBS instead of on the UoD, file-based modularisation, and the possibility of arbitrary reactive systems in data flow models. The conclusion is that nothing is to be gained by combining JSD models with data flow models. As shown by the transformation guidelines, a JSD model contains all information that a DFD model contains. By contrast, a transformation of a data flow model into a JSD model cannot be done without doing some extra analysis of the UoD, so as to get extra information about the objects represented by the system and about the behaviour of these objects [28].

5 Discussion and Conclusions

If we combine ER models with JSD models, we get a system model that consists of three kinds of models.

1. The ER model represents entity– and relationship types and cardinality constraints. Some relationship types may correspond to common actions and some relationships may correspond to data streams.

2. A UoD model represents the UoD as a system of communicating entities, and represents the life cycle of each entity by a PSD.

3. A system network represents the DBS as a system of communicating processes, some of which are entity life cycles and others of which are function processes. The structure of function processes is represented by PSD's. In order to represent *all* communications between processes, we could extend this network with a representation of the common actions between life cycles.

A combined JSD/ER model has advantages over each of these models separately. It allows us to specify which communications must be rembered (as relationships), to specify cardinality constraints, and to make the relationships present in the JSD model explicit.

Further enhancements to JSD could be made by choosing more expressive representations for process structures, such as statecharts [13] or process graphs [2, 22]. Another consideration for improvement is to use a declarative specification of output functions instead of the imperative specification by means of a PSD. These enhancements would take us beyond the scope of the present paper, though.

It was argued in section 4 that JSD models and data flow models have an opposite philosophy. The differences between the two approaches are the representation of behavior as well as interfaces in JSD versus the representation of interfaces only in DFD's, UoD orientation versus DBS orientation, and object-oriented versus file-oriented modularization. It was argued that these contrasts are too large to combine the two kinds of models in one modelling effort. In addition, it was argued that reactive systems are in general more easily understandable in JSD than they are in data flow modeling.

These results can be applied to object-oriented methods like OMT [19] and the Shlaer/Mellor life cycle modelling method [21]. OMT divides a conceptual model of a DBS into three components, an object model, a dynamic model, and a functional model. The *object model* is an extension of ER models with semantic constructs like a taxonomic hierarchy. The *dynamic model* is a representation of the life cycles of object by means of state charts. The *functional model* is a representation of the processing that a system does in order to transform its input into its outputs by means of a data flow model; it is used to define the meaning of the actions (called operations in OMT) that can occur in the life of objects. The object model and dynamic model together have, very roughly, the same structure as a combined ER/JSD model, but give more information on taxonomic structure, constraints, derived attributes, etc. As argued in this paper, the functional model is the odd man out here, because it has a philosophy that goes counter to that of object-orientation. The discrepancy between the functional model and the other two models is made even larger by relaxing the connection between the models: One object class may correspond to several data stores, that each contain some attributes of the class; one action in the dynamic model may correspond to several data transformations in the functional model; and one data transformation may correspond to several actions. All of this makes the relationship between the functional model and the rest of an OMT model hard to understand. One conclusion from the argument in this paper is that OMT could benefit from replacing the data flow technique in the functional model by some technique like that used for system networks in JSD. Similar observations can be made of the Shlaer/Mellor life cycle modeling method. For more details on the combination of object-orientation and data flow modeling in OMT, see the discussion by Wieringa et al. [30].

As a final remark, it is interesting to note that JSD, Structured Analysis and object-oriented modeling all have their motivation in an observation made in structured programming in the later 1960s and early 1970s [9]:

1. Software should be given a modular structure, and
2. this structure should have a clear and simple correspondence with the structure of the real-world problem that the software is intended to solve (and so should be independent from implementation structures).

The first observation found its way to structured analysis and structured design in the 1970s and 1980s, but the second observation was not given enough attention. By contrast, both observations

were retained in such languages as Simula and SmallTalk and through them, became basic principles of object-oriented modeling techniques. They also were retained inJackson Structured Programming [14] and later in JSD. It is this common basis which makes JSD a sound partner for some of the object-oriented methods [5].

Acknowledgement. This paper benefited from comments on an earlier version made by Remco Feenstra.

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] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

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

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

[5] 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.

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

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

[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] O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming.* Academic Press, 1972.

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

[11] E. Downs, P. Clare, and I. Coe. *Structured Systems Analysis and Design Method: Application and Context.* Prentice-Hall, second edition, 1992.

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

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

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

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

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

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

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

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

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

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

[22] P.A. Spruit and R.J. Wieringa. Some finite-graph models for process algebra. In J.C.M. Baeten and J.F. Groote, editors, *2nd International Conference on Concurrency Theory (CONCUR'91)*, pages 495–509, 1991.

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

[24] 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.

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

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

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

[28] 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.

[29] R.J. Wieringa and W. de Jonge. The identification of objects and roles. Technical Report IR-267, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1991.

[30] R.J. Wieringa, R. Jungclaus, P. Hartel, and G. Saake. 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.

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

Contents

1 Introduction **1**

2 Jackson System Development **2**
2.1 UoD models 2
2.2 DBS models 3

3 Combining JSD Models with ER Models **6**
3.1 JSD Entities and ER Entities 6
3.2 Relationships 7
3.3 Attributes 8
3.4 Using relationships to remember common actions 8
3.5 Using relationships to reduce action sharing 10
3.6 Functions 12
3.7 Process connections, relationships and cardinalities 13

4 JSD Models and Data Flow Models **14**
4.1 Transforming PSDs into a data flow model 15
4.2 Transforming a system network into a data flow model 18

5 Discussion and Conclusions **19**