Onderzoeksrapport Nr. 8606

EXTENDING A DYNAMIC MODELLING METHOD
USING DATA MODELLING CAPABILITIES: THE CASE OF JSD.

BY

M. MEES AND F. PUT
EXTENDING A DYNAMIC MODELLING METHOD
USING DATA MODELLING CAPABILITIES: THE CASE OF JSD.

M. Mees and F. Put

Department of Applied Economic Sciences,
Catholic University of Leuven,
Dekenstraat 2, B-3000 Leuven (Belgium).

Acknowledgments - This research was partially supported by the Generale Bank, which is gratefully acknowledged. The authors want to thank M. A. Jackson, H. Jansen and M. Verhelst for their stimulating comments on an earlier draft of the paper.
Index Terms - Behaviour modelling, data abstractions, data modelling, extended entity-relationship approach, interaction of structure and dynamics, invariant rules, system specification.
Abstract - The authors investigate how the Entity-Relationship (ER) approach to database design can be used to solve a main problem of Jackson's system development method (JSD); namely the lack of a technique for data modelling. An ER diagram can be transformed into an equivalent diagram, exactly matching the system specification of JSD, but containing more semantics. It will also be shown how some formal concepts can be introduced, using data abstractions, and how this may lead to more flexibility in system development. At the same time JSD contributes its dynamic modelling capabilities, so that integrated tool for system development can be obtained.

1. INTRODUCTION.

Jackson System Development (JSD) is a method for developing information systems in a structured way [1] [2] [3] [4]. The two most important pillars of the method are the separation of model and function on the one hand and the separation of specification and implementation on the other hand.

A first important property of the method is that the real world is modelled dynamically, thus avoiding a number of integrity problems that usually are the consequence of modelling a dynamic reality in a static way, as is done very often in a database environment. The model is dynamic since it is built out of a number of model processes, the states of which at execution time being a mirror of the corresponding entities' states in the real world.

By adding extra processes, functions can be imposed on this model.

A second important property, following from the first one, is the fact that the specification is more or less executable. A JSD specification is not a hierarchy of procedures, but consists of a network of interacting processes, resembling communicating sequential processes [5] [6]. The availability of an adequate programming language would allow to execute the processes directly [7]; otherwise, the processes have to be converted to hierarchically structured procedures during the implementation step.

JSD is considered to score well with respect to process modelling (the dynamic structure), but to lack comparable techniques for data modelling (the static structure) [8] [9].
In this paper the authors will indicate how the entity-relationship approach to database design can be useful to solve this shortcoming. This will be illustrated using an example, which is first explained in section two.

In section three, the example will be modelled according to JSD, after a short explanation of the method, indicating the problems that may be encountered during specification and how they are solved. Not all steps will be worked out: only the modelling of the real world will be handled, functions and timing requirements are not considered.

After a short explanation of the ER approach in section four, it will be shown how an ER model can be adapted to the needs of the system development method. Each form of data abstraction has a counterpart in this development method, as is shown explicitly by the ER approach. This may help to formalize the method for system development, and makes it more flexible as is shown in section five.

2. THE MAIL COURSE COMPANY.

The next paragraphs are a description of the administration of a company offering mailed courses.

People who are interested in one or more courses announce this to the institution, asking for a final subscription or asking for a trial package. According to the rules of the institution, a person can ask for a trial package only once, and a person who asked for a final subscription cannot ask for a trial package any more.

When a person asks for a trial package, his name and address, and the date of sending are recorded. A subscription form is included in each trial package. If the institution does not receive the form in return within 20 days, a kind incentive letter is sent to that person, such that he can still subscribe to one or more courses. As long as there is no reaction, this procedure is repeated every half year (at most 4 times). After that, the person not reacting is removed from the list of prospects. If a person does react he becomes a student and additional data such as birthdate and telephone-number are recorded.
When a person immediately takes a subscription for one or more courses, his name, address, birthdate and telephone-number have to be registered. Each course consists of a number of course parts. The students of the course have to prepare a task per course part. They can determine the speed of the course themselves since the next course part is only sent after they prepared their previous task.

A number of payments must be made for each course subscription. A student who is overdue for a certain course gets a friendly dunletter. If he or she does not react, a second dun follows, less friendly now. When the student stays overdue, even after the second dun, the course is stopped. Each course ends with a test which is graded, unless it has been stopped prematurely.

3. SYSTEM DEVELOPMENT: A DYNAMIC MODELLING APPROACH.

3.1 JACKSON SYSTEM DEVELOPMENT.

JSD aims at methodical system development. There are six main steps in the method.

1. Entity Action and Entity Structure Step.
2. Initial Model Step.
3. Interactive Function Step.
4. Information Function Step.
6. Implementation Step.

In the Entity Action and Entity Structure Step attention is focused first on the subject matter of the system and not on the system itself. This subject matter can be described in terms of actions. An entity then is any object that performs or suffers actions in a given time sequence. Out of lists of candidate actions and of candidate entities relevant actions and entities are selected. The chosen actions are defined properly and their attributes are added. The end-product of this step consists of one or more Sequence Diagrams (Structure Diagrams in Jackson's terminology) that describe the time constraints between the actions.
In the *Initial Model Step* and the *Interactive Function Step*, the description defined in the previous step is realized as a set of communicating sequential processes. The *Initial Model Step* starts by dividing the actions into two types: those that will be generated externally and those that will be generated internally. Next, a connection is defined between the external reality and the model. A level-0 process is used for the description of the real world, while the corresponding level-1 process models this description within the system. Connections between the real world processes and the model processes, and among the model processes themselves are made by data streams and state-vector inspections. All this can be represented in the System Specification Diagram (SSD).

Automation is achieved in the *Interactive Function Step*. In most of the cases this happens by adding new processes to the specification, that use information from the model processes to generate actions internally.

The outputs of the system are finally added to the specification during the *Information Function Step*. New processes are added to extract the necessary information from the model processes by data stream and state-vector connection. This information is used to produce the required outputs. It is worthwhile mentioning that one can use Jackson Structured Programming (JSP) to arrive at a structure for both the interactive and information function processes [10] [11] [12].

A set of informal timing constraints for the implementation is gathered in the *System Timing Step*. One defines the speed of each output stream and one decides on how up-to-date the information in the output has to be. The constraints that must be satisfied to ensure that all input is collected and all necessary orderings are preserved are also defined in this step.

Since the specification will seldomly fit the target hardware and software environment, transformation of the specification is one of the main tasks in the *Implementation Step*, two other main tasks being process scheduling and storage and access of state-vectors.
3.2 THE ENTITY ACTION AND ENTITY STRUCTURE STEP.

Starting from an initial list, we selected the relevant actions. After removal of rejected items, we keep the following actions:

SUBSCRIBE: when someone sends a signed form to the institution he subscribes for the course concerned (one subscription per course).
Attributes: name, address, birthdate, telephone-number, course-name, subscription-date.
Action of TRIAL-PERSON and STUDENT.

TRIAL-SUBSCRIBE: a person can, once during his lifetime (in the system), ask for a trial package.
Attributes: name, address, send-date.
Action of TRIAL-PERSON.

SEND-COURSE-PART: a student gets a number of course parts per course (at least as long as the course is not stopped).
Attributes: name, course-name, part-number.
Action of STUDENT.

PAY: each term of payment, the student is expected to deposit an amount per course.
Attributes: name, course-name, amount, term-number, date-paid.
Action of STUDENT.

DUN: when a student does not pay in time, the institution sends a dun-letter.
Attributes: name, course-name, term-number.
Action of STUDENT.

STOP: a course is stopped when a student, after having received two dun-letters still did not pay.
Attributes: name, course-name.
action of: STUDENT.
PREPARE-TASK: students have to prepare one homework for each course part.
Attributes: name, course-name, part-number, score.
Action of STUDENT.

TEST: normally a course ends with a test.
Attributes: name, course-name, grade.
Action of STUDENT.

INCITE: when someone who has taken a trial subscription does not react, i.e. he does not subscribe, we send him an incentive letter.
Attributes: name.
Action of TRIAL-PERSON.

TRIAL-STOP: when a trial-person has not reacted after four incentive letters, we strike him off.
Attributes: name.
Action of TRIAL-PERSON.

These actions belong to two entities:

TRIAL-PERSON : TRIAL-SUBSCRIBE, INCITE, TRIAL-STOP, SUBSCRIBE;
STUDENT : SUBSCRIBE, PAY, SEND-COURSE-PART,
DUN, PREPARE-TASK, TEST, STOP.

Time has come to specify how the actions of each entity are ordered. For that purpose we use Sequence Diagrams. Such a Sequence Diagram is a tree that consists of a combination of the three classical programming control structures: sequence, iteration and selection. The root of the tree references the entity name and the actions are represented in the leaves: there is always exactly one path from the root to any leaf. A sequence is indicated by leaving the child boxes unmarked; e.g. TRIAL-PERSON in figure 1 is a sequence of TRIAL-SUBSCRIBE, INCENTIVE-BODY and TRIAL-END. When there is an asterisk in the upper
right corner of the child one faces an iteration: INCITE occurs zero or more times within INCENTIVE-BODY. A small circle in the upper right corner of a box indicates the presence of a selection. TRIAL-END is either a TRIAL-STOP or a S-FURTHER.

There is a problem since the interpretation of the action SUBSCRIBE will differ depending on the moment one performs the action. Whether a subscription is the very first one or not for a STUDENT differs a lot. Not only the attributes are different, but also the semantics: a next subscription changes the status of the entity whereas a first subscription creates a new entity.

Therefore SUBSCRIBE will have three aliases [13]: each alias corresponds to a possible interpretation of the action.

S-FIRST: when a student subscribes for the first time and he did not subscribe for a trial-package before, identifying data and the subscription have to be registered first.
- Attributes: name, address, birthdate, telephone-number, course-name, subscription-date.
- Alias of SUBSCRIBE.

S-FURTHER: when a trial-person subscribes for a course the first time, some additional identifying data have to be registered too.
- Attributes: name, birthdate, telephone-number, course-name, subscription-date.
- Alias of SUBSCRIBE.

S-NEXT: a student takes his second, third, ... subscription.
- Attributes: name, course-name, subscription-date.
- Alias of SUBSCRIBE.

In the Sequence Diagrams, leaf nodes show actions in their lower sections and aliases in their upper sections. Sometimes it will not be necessary to distinguish between aliases of an action. In that case the
action is treated like any other action (e.g. SUBSCRIBE in the SUBSCRIPTION structure).

TRIAL-PERSON seems to be a very simple entity: the life-cycle starts when someone TRIAL-SUBSCRIBES, it might be necessary to INCITE him a couple of times. In any case, he ends his TRIAL-PERSON life either when he Subscribes-FURTHER or when he is TRIAL-STOPped (figure 1).

```
TRIAL-PERSON
  /    \
 TRIAL-SUBSCRIBE  INCENTIVE-BODY  TRIAL-END
     |         |        |
    INCITE  TRIAL-STOP  SUBSCRIBE-FURTHER
```

Fig. 1: Sequence Diagram of TRIAL-PERSON.

Our next entity is STUDENT. The structure of STUDENT exhibits an interleaving clash since a particular STUDENT does not have to take his courses sequentially: he can SUBSCRIBE for a course while he is still busy with one or more other courses. The fact that there may exist several course subscriptions concurrently for one STUDENT cannot be expressed adequately in a single Sequence Diagram (figure 2). Therefore a new, marsupial entity is needed: SUBSCRIPTION. The entity SUBSCRIPTION puts some more time ordering in our model (fig. 3).

There is nevertheless still some indeterminacy because PAYing and PREPARing of TASKs do not follow the same rhythm: it happens that someone PREPARES three TASKs before PAYing, but it can also happen that the same person PREPARES one TASK before PAYing twice, and this for the same SUBSCRIPTION.
Fig. 2: Sequence Diagram of STUDENT.

Fig. 3: Sequence Diagram of SUBSCRIPTION.

The indeterminacy in STUDENT was the result of a multithreading or
interleaving clash, in this case however we observe a different kind of structure clash (an ordering clash), leading to the identification of different roles of a single entity rather than two different entity types.

The single entity is SUBSCRIPTION. The indeterminacy is caused by the fact that there are two ways of partitioning a SUBSCRIPTION. The roles are the SENDING of COURSE PARTs and the PREPARING of TASKs on the one hand and everything concerning PAYing and DUNning on the other hand. That is why we are forced to introduce two new processes: SUBSCRIPTION-HOMEWORK (figure 4) and SUBSCRIPTION-PAYMENT (figure 5).

We have come to a point now that we seem to have a fairly robust model. However, our model is still incomplete. Nowhere can we find that somebody who became a STUDENT is not allowed to become a TRIAL-PERSON. Until now we even did not model that it is not allowed to be a STUDENT and a TRIAL-PERSON at the same time because we did not provide a link between these two entities.

The most appropriate solution to solve this boundary clash seems to be the introduction of a new entity: PERSON. PERSON performs or suffers the actions of both TRIAL-PERSON and STUDENT (figure 6). The most important thing is that a PERSON starts his lifetime with a registration, either by TRIAL-SUBSCRIBing or by Subscribing-FIRST, but he cannot do both actions.
Fig. 5: Sequence Diagram of SUBSCRIPTION-PAYMENT.

Fig. 6: Sequence Diagram of PERSON.
3.3 THE INITIAL MODEL STEP.

In the previous steps we have made a description of the real world in terms of entities, the actions they perform and suffer, and the ordering of those actions. In this step we will begin to build our system by specifying a set of sequential processes which model the real world entities and their behaviour. There will be one sequential process in the model for each Sequence Diagram defined in the previous section.

To ensure that the behaviour of the model correctly reflects the events in the real world, we will need to connect the model process, directly or indirectly, to the real world. But first of all we have to decide on which actions will be gathered outside the system and which will be generated by the system itself. In other words, we have to decide which parts will be automated. This automation happens by adding functions to the model.

We could for example generate the following actions internally: INCITE, TRIAL-STOP, DUN, STOP and SEND-COURSE-PART. INCITE and TRIAL-STOP will be generated by a function INCENTIVE, while DUN and STOP will be the possible result of a function process DUNNING and SEND-COURSE-PART last but not least should be expected from SEND.

Now we can connect our model processes with their sources of actions in a System Specification Diagram. Sequential processes are denoted by rectangles, circles represent data streams. PERSON-1, STUDENT-1 and SUBSCRIPTION-1 in figure 7 are examples of sequential processes. Each action of SUBSCRIPTION-1 is signalled by STUDENT-1 via a record of SS. Multiplicity is shown by a double bar on the arrow on the side of the many processes.

The PERSON-0 process represents the real-world behaviour of the entity PERSON. PERSON-1 is the model process of this entity. There is no such thing as a level-0 process for STUDENT because all actions of STUDENT are already actions of PERSON. The same reasoning applies to TRIAL-PERSON, SUBSCRIPTION, SUBSCRIPTION-PAYMENT and SUBSCRIPTION-HOMEWORK (figure 7).
Figure 7: Initial Model: System Specification Diagram.
4. DATA MODELS AND ABSTRACTIONS.

In the previous section, a list of entities (TRIAL-PERSON, STUDENT) was made, more or less in an intuitive way. Each of these entities performs or undergoes some actions, which are bound by the life-cycle constraints of those entities.

During the entity structure step it was sometimes necessary to extend the list of entities, to be able to include all ordering constraints in a Sequence Diagram.

- SUBSCRIPTION, introduced as a marsupial of STUDENT, was necessary to model the ordering constraints within each course a student subscribes.

- For each SUBSCRIPTION, there are actions concerning the financial administration (DUN, STOP, PAY), and actions about the actual behaviour of a course (SEND-COURSE-PART, PREPARE-TASK); therefore two new structures (SUBSCRIPTION-PAYMENT, SUBSCRIPTION-HOMEWORK) were introduced, modelling for a subscription the two roles that can be active at the same moment.

- A STUDENT cannot ask for a trial subscription again; this is reflected in the entity life cycle of PERSON, introduced above STUDENT and TRIAL-PERSON.

New entities are introduced when problems are recognized during the entity structure step: a parallelism may be found that cannot be modelled in a single structure (interleaving clash and ordering clash); or a constraint involving several structures, that cannot be modelled without a higher level structure (boundary clash). Entities are only considered if they are necessary for the modelling of actions.

Database applications however have both structural and behavioural properties. Behaviour is modelled in the Sequence Diagrams representing the entity life-cycles, using actions. The list of entities, one obtains at the end of the initial model step, has to be elaborated further.
before it can be considered as a data model, because entities without a life-cycle are not represented, and relationships between entities are only considered implicitly if there are actions common to those entities. The logical database design is disposed of as a matter of implementation and has to start completely when the JSD specification is ready. In this section we will show that there is no reason to postpone the design of a data model, which in turn can make a considerable contribution to JSD.

4.1 THE ENTITY-RELATIONSHIP APPROACH.

The Entity-Relationship (ER) approach [14] [15] presents a useful notation for data modelling and, also because of its simplicity, received a lot of attention in the literature [16] [17] [18]. The three basic concepts for modelling are entity, relationship and attribute.

An entity is any concrete or abstract thing in the real world, e.g. student John Smith or course Math1A. Entities with similar attributes are classified in an entity-type, e.g STUDENT or COURSE (we will follow the convention that types are written in capital letters), represented graphically as a rectangular box in an Entity-Relationship Diagram (ERD), as is shown in figure 8.

![Entity-Relationship Diagram](image)

Figure 8: Entity-Relationship Diagram.

In a similar way, relationships are classified in relationship-types, represented as a diamond-shaped box (e.g. SUBSCRIBE in figure 8). The functionality M-N indicates that each occurrence of COURSE may be involved in one or more subscriptions, each for one occurrence of STUDENT, and each occurrence of STUDENT must have one or more subscriptions, each for one occurrence of COURSE.
The relationship-type `SUBSCRIBE` is said to be *total* on `STUDENT`, because each student must have a subscription for at least one course, otherwise he would not be a student. Total relationships are indicated by a dot in the graph.

Attributes are represented by a circle in the graph. Remark that both entities and relationships may have attributes, as is shown in figure 8.

Besides *classification* in types, three additional abstraction mechanisms have been proposed to extend the modelling capabilities of the ER approach: *aggregation*, *generalization* [19], and *association* [20] [21].

- **Aggregation** is a form of abstraction, originally proposed in [22], where a relationship between entity-types is considered as a higher level entity-type; e.g. the relationship between `STUDENT` and `COURSE` can be seen as a higher level entity of type `SUBSCRIPTION` (figure 9).

![Figure 9: Aggregation.](image)

- **Generalization** is an abstraction, originally proposed in [23], where a higher level and more general, generic entity-type is created by extracting from several category entity-types the commonalities, and suppressing the details that differ in the description of those types. For example, `PERSON` is a generalization of the entity-types `TRIAL-PERSON` and `STUDENT` (figure 10).

- **Association** is a database abstraction, originally proposed in [24], based on the mathematical notion of set, where a collection of member entities is considered as a higher level set entity. For example, all parts which belong to one course can be considered as a set `COURSE` (figure 11).
Very often an association is also a *weak relationship*, indicated by a double sided diamond-shaped box and a dot; in this case it means that each occurrence of *COURSE-PART* must belong to a specific occurrence of *COURSE* (it is not sufficient that it belongs to any course whatsoever).

Generalization supports downward inheritance: all properties of the generic entity are inherited by the category entities. Aggregation and association support upward inheritance: properties of the components or members are inherited by the aggregate or set.

### 4.2 A DATA MODEL FOR JSD.

The application of the ER approach may give several equivalent, but structurally different solutions for the same problem. Given the description of the requirements and using the ER approach, a straightforward solution for the Mail Course Company is given in figure 12. *PERSON* is a generic type with specializations *TRIAL-PERSON* and *STUDENT*. Each student takes at least one subscription for a course. For this
subscription he has to make a set of payments and he receives a set of homeworks, each for one part of the course, but always one after the other. That is the reason why the functionality of both PAYMENT and HOMEWORK is 1-N: only one homework and one payment can be active for a particular subscription at a particular moment. (If we consider an additional action GRADE-HOMEWORK one could argue that the functionality of HOMEWORK is M-N, if a new homework may be sent, even before the previous one has been graded. However one occurrence of the relationship is bound by an older one just as in case of functionality 1-N.)

Figure 12: ER Diagram for the Mail Course Company.
The ER Diagram presented in figure 12 does not fit the JSD model found in the previous section, which only recognizes entities with a life-cycle, and certainly not relationships of functionality M-N.

An alternative ER Diagram may be obtained, starting from the solution found in figure 12, and matching the system structure produced by JSD, by the application of a number of transformation rules. A first rule, which is based on the instantiation concept [25], transforms a relationship-type into an entity-type; the second rule does the inverse: it decomposes an entity-type such that it can be considered as an aggregation of several roles of the entity-type.

- **Transformation T1.** A relationship-type R of functionality M-N between entity-types E and F may be replaced by a new entity-type \( R^* \), inheriting the identifiers of E and F and the attributes of R, and E and F each become owner of a weak association on \( R^* \), with \( R^* \) as member.

This means that in the example, the relationship-types SUBSCRIBE and COURSE-PART may be instantiated as entity-types. Remark that for \( R \) being total on E (or F), the association must be weak on \( R^* \) and total on E (or F), as is the case in the example for SUBSCRIPTION. Because PART is completely included in the new entity-type COURSE-PART as an attribute, it can be deleted from the diagram.

- **Transformation T2.** If an entity-type E is involved in two (or more) relationships \( R,S \), which cannot be considered as the result of the application of T1, and at least one of which has either a functionality of 1 on E, or a dependency between its occurrences, it can be decomposed in two (or more) entity-types \( E_R \) and \( E_S \), linked to each other with a weak relationship of functionality 1-1 (one entity-type implies the other).

Application of this rule makes SUBSCRIPTION an aggregation of SUBSCRIPTION-PAYMENT and SUBSCRIPTION-HOMEWORK. The overall result can be presented as in figure 13, and this is exactly the structure corresponding to the model found in section 3.
Figure 13: Transformed ER Diagram for the Mail Course Company.

Whether we use the transformed or the original ER Diagram as a basis for logical database design is not important. Both diagrams are equivalent, and can be mapped to the same relational or CODASYL database schema. (Two ER Diagrams are said to be equivalent if they represent the same information)

Theorem. If an Entity-Relationship Diagram ERD' is derived from another Diagram ERD, using the transformation rules T1 and T2, it is equivalent to this ERD,
Proof (only informal). No attributes are dropped, so the collection of attributes remains the same (domain data compatibility). It can easily be shown that all functional dependencies that are present in ERD, although they are derived by different rules, are still present in ERD' (data dependency equivalence) [26]. The existence dependency of a relationship-type \( R \) on its entity-types (a relationship of type \( R \) can only exist if the entities exist) is reflected in the generated weak associations on \( R^* \). If \( R \) is total on an entity-type, e.g. \( E \), the generated association will be total on \( E \) and weak on \( R^* \). As a consequence, deletion of an occurrence of \( R^* \) only implies the deletion of the occurrence of \( E \) if this \( R^* \) was the last member in the association, and this is exactly the same for the deletion of a relationship in the original diagram. Because of the generated weak relationship between both roles of the aggregate, it makes no difference whether relationships have an entity representing a role, or the aggregate itself as participant.

The rules for mapping an ER Diagram into a database schema depend on the particularities of the system in use, and therefore fall outside the scope of this paper. However it can be seen intuitively that ERD and ERD' can be mapped into the same relational database schema.

5. THE INTERACTION OF STRUCTURE AND DYNAMICS.

5.1 DATA ABSTRACTIONS IN THE DYNAMIC MODEL.

The example of the Mail Course Company does not only show the possibility of adapting an ER Diagram to the needs of a JSD model, but it also shows explicitly the presence of data abstractions in this model. JSD starts with a number of entities (TRIAL-PERSON, STUDENT) intuitively chosen from an arbitrary level of abstraction. Because of concurrency problems new entities, with a lower level of abstraction have to be introduced (SUBSCRIPTION, SUBSCRIPTION-PAYMENT, SUBSCRIPTION-HOMEWORK). But also entities with a higher level of abstraction may be needed to model constraints, involving more than one entity (PERSON).
The ER approach explicitly shows the presence of data abstractions from the beginning. Besides there seems to be a separate abstraction mechanism for each type of difficulty that can be encountered in JSD:

- The separation of a marsupial entity corresponds to the presence of an association, that may be obtained by transforming an M:N relationship or aggregation using T1.

- An entity performing several roles can be considered as an aggregation of entities, representing those roles, through the application of T2.

- Generalization corresponds to the introduction of a new entity that is necessary to model the ordering of existence of several existing entities. If we did not start from the specializations TRIAL-PERSON and STUDENT, but from the entity-type PERSON, it would still be possible to recognize the generalization as will be shown below.

The recognition of the need for a new entity-type indicates the presence of a data abstraction, corresponding to the particular type of problem encountered. So, although JSD does not have a notation to represent those data abstractions, it has a way of finding them.

The generalization was detected, starting from the specializations. Now we may ask the question whether or not it is necessary to model each specialization in a Sequence Diagram if we already model the generic type in such a diagram. If we develop a Sequence Diagram for PERSON, do we still need the diagrams for TRIAL-PERSON and STUDENT? (An answer to this question may help to solve the problem of recognizing the presence of a generalization, starting from the generic type).

A first trial to put all constraints in the Sequence Diagram of PERSON is represented in figure 14. However, the diagram does not model all constraints in an appropriate way. A TRIAL-PERSON must not be registered again if he likes to SUBSCRIBE for a course; on the other hand, someone who did not ask for a trial package before, must be registered when he SUBSCRIBES for a course. This restriction is not imposed by the
Fig. 14: Modelling a Generalization without Specializations.

Sequence Diagram: there is no way to indicate that each alternative of the selection POSSIBLE-TRIAL in the left subtree corresponds with a specific alias of SUBSCRIBE in the right subtree. The same reasoning applies to the selection TRIAL-END.

Modelling these restrictions makes the Sequence Diagram of PERSON extremely complex, with a selection on the highest level, and parts of the diagram must be replicated several times, in each branch of the selection (to model each correspondence separately in an explicit way).

Now, we also have a criterion for the detection of the presence of a generalization, starting from the generic type:
If on the highest level of its Sequence Diagram, an entity has a selection or a hidden selection (as in figure 14 where we have a sequence of two selections, each with an empty option), it may be split in several specializations.

There are some additional benefits connected to the separate modelling of a generic type and its specializations.

If we have to model a new action CHANGE-ADDRESS, which may occur any time (without restriction), this action must be added to the Sequence Diagram of PERSON as an option under PERSON-ACTION. If we do not model the specializations separately, this action must be added several times to the Sequence Diagram of PERSON, in every place where it may occur, thus making the Sequence Diagram even more complex.

Finally, not all actions may arrive from all sources in a random order if we consider the single entity PERSON, although it is indicated in this way in the SSD (figure 7). A notation for the exact merging of incoming actions is not available for this case. When also the specializations are modelled explicitly, the notation of the SSD completely matches the actual merging of incoming actions.

5.2 ADDING NEW REQUIREMENTS.

Until now, all illustrations were based on the requirements given in section two, describing the student administration of the Mail Course Company. Suppose that later on a new system must be developed for course administration (course scheduling, teacher administration,...), based on new requirements.

Probably we will need a new entity COURSE. Remark that this entity was not modelled until now in JSD, because it had no life-cycle in the system for student administration. This new entity COURSE may be found to need a marsupial entity, representing the presence of a student in a course. Using JSD there is no way to recognize that this is actually the same entity SUBSCRIPTION as was found before for student administration.
Using the ER approach for building conceptual schemas may avoid these difficulties, because entities without a life-cycle are also modelled, and a correspondence may be detected easily. If a new system is developed, the ER Diagram for this system can be integrated in the existing conceptual schema, as is shown in [27].

Also the adjustment of existing specifications to new conditions is much easier using the ER approach. Changed requirements, e.g. course parts (bills) are sent on predefined time instances and homeworks (payments) can be returned, even after the following course part (bill) has been sent, makes it necessary to introduce a new marsupial, HOMEWORK (PAYMENT), as may be recognized immediately in the ER Diagram for student administration.

5.3 THE INTERACTION.

A perfect correspondence between the Entity-Relationship Diagram (ERD) and the System Specification Diagram (SSD) was found earlier: all entities that are modelled in the latter can be found in the ERD, and the abstractions of the ERD are present in the SSD. The ER approach provides the development method of Jackson some useful guidelines in the detection of a minimal set of entities that is needed to model the dynamic constraints. At the same time the ER Diagram is a much stronger starting point for database design, in fact it is database design. All entities, also those without a life-cycle, and all relationships that are necessary are represented. In this way the ER approach brings about a nullification of a criticism mentioned earlier, namely that JSD lacks a data modelling counterpart. On the other hand, JSD may add to the static ER approach its dynamic modelling capability, so that we can talk of a real interaction between both approaches. The modelling of dynamics is one of the fundamental research topics for the future in the ER approach [28], which until now received only scattered attention, e.g. [29] [30].

The ER Diagram however contains more semantics than the SSD, reflected in a number of existence rules that must always be satisfied: the in-
variant properties [19], associated with abstractions and relationships. Because of the perfect correspondence between ERD and SSD, these invariant rules can be imposed on the SSD to, thus adding additional semantics. The consequence is that these invariant rules can be used to check the correctness of the Sequence Diagrams of objects involved in a data abstraction, or even more, to derive mechanically the Sequence Diagram of an entity.

For example, the generalization PERSON can only exist if one of the specializations TRIAL-PERSON or STUDENT exists. On the other hand, if one of the specializations exists, also the generic type must exist.

As a consequence, the generic type PERSON is created either by a TRIAL-SUBSCRIBE, the action that creates a trial-person (figure 1), or by Subscribe-FIRST, the initial action of STUDENT (figure 2). (Remark that Subscribe-FURTHER cannot be the initial action, because it is also the final action for a trial-person.) TRIAL-STOP on the other hand causes the end of PERSON. The other actions of the specializations can happen in any order once the entity is created, giving a structure that is exactly the Sequence Diagram of PERSON (figure 6).

Each student must follow at least one course (the association is total on STUDENT). However, if the last occurrence of SUBSCRIPTION takes an end, either by a final TEST or a STOP, there is no action in the Sequence Diagram of STUDENT (figure 2) to restore this constraint. This means that the specification as found in section 3 is incomplete and that the requirements of section 2 must be elaborated before this situation can be handled, because several solutions are possible: it may be required to remove the person, to create a new specialization EX-STUDENT or, if the invariant rule is considered a prescriptive rule, to send the student a question for a new subscription.

In the same way, an aggregation can only exist if all participating components exist: for example, the creation of an occurrence of SUBSCRIPTION requires the existence of a corresponding occurrence of SUBSCRIPTION-HOMEWORK and of SUBSCRIPTION-PAYMENT. This means that the
Sequence Diagrams of the latter two (figures 4 and 5) are not complete: both should start with an action SUBSCRIBE, and terminate with either a STOP or a TEST (because of the weak relationship, the components cannot exist individually), just like SUBSCRIPTION itself. In that case, the semantics of SUBSCRIPTION is completely replicated in its two components, making the Sequence Diagram of the aggregate redundant. Furthermore, because it is not known in advance whether an active homework will be interrupted by a STOP action or not, SUBSCRIPTION-HOMEWORK must be extended with a mechanism to solve this recognition difficulty (back-tracking), thus making it needlessly complicated. Instead of using an artificially constructed weak aggregation of functionality 1-1, it would be more elegant to model the roles of an entity separately, without replication of the initial and terminating actions.

A new ER modelling concept is needed, to represent a role of an entity, together with appropriate invariant rules, in order to be able to check the correctness of the Sequence Diagrams. Such a role is only necessary to discuss dynamic properties of entities (it does not represent the entity itself), and is therefore not absolutely required in the ER diagram representing a data model (it has no separate database counterpart). A role is connected in an unbreakable way to the entity, just as if there exists a weak relationship. If the relationship involved in a role is total or weak on the entity, then the role must be created and deleted with the same actions as the entity itself.

Transformation T2 can now be reformulated as follows:

* Transformation T2'. If an entity-type E is involved in two (or more) relationships R,S, which cannot be considered as the result of the application of T1, each relationship-type with a functionality of 1 on E, or a dependency between its N occurrences for E, requires a role of that entity-type to be modelled separately.

Remark that the Sequence Diagrams of SUBSCRIPTION-HOMEWORK and SUBSCRIPTION-PAYMENT (figures 4 and 5) are now acceptable dynamic models
of the two roles of SUBSCRIPTION. No additional backtracking is necessary: after a STOP, there can be no further actions, as is indicated by the life-cycle of SUBSCRIPTION.

If this role concept is shown graphically as a rectangular box inside the box of an entity, a new representation for the transformed ER Diagram (figure 13) is given (partly) in figure 15.

Before we can talk of a real integration of static and dynamic concepts into one powerful toolbox, a number of problems have to be solved first, especially in connection with the role concept, e.g. how the additional semantic constraints in the ER Diagram may restrict or drive the propagation of actions for the different abstraction mechanisms, and whether dynamic constraints can be inherited (imminent if a role itself is the owner in an association); how historical information must be handled and derived, when each object has an action to end its active life; which rules are necessary for the mapping of the role concept to an implementation level; also it must be investigated whether the JSD notation provides sufficient power to model (all) dynamic constraints, and eventually, what additional concepts are necessary, and so on.
In this paper we started from an ER Diagram and investigated how it could be transformed in order to obtain a minimal set of Sequence Diagrams, as would be the result of the application of JSD: relationships of functionality M-N are treated as entities (transformation T1), different roles of an entity are modelled separately (transformation T2').

It is also possible to work the other way round: starting from an ER Diagram, simply consider all entities, relationships and abstractions as objects that have to be modelled dynamically. In this way there will be a lot of redundancy in the Sequence Diagrams, because very often the dynamic structure of a relationship simply is a replication of the Sequence Diagram of the single-valued entity involved. However, if we adopt this second approach, transformation T1 is no longer necessary: all relationships are treated separately. Transformation T2', applied directly to the original ER model, identifies the entity roles which require explicit modelling. There are a few additional advantages:

• If there are several aliases of the same action causing some indeterminacy in a Sequence Diagram, the new "redundant" structures can be used sometimes to derive the predicates that are needed to solve this indeterminacy.

• The problem of the placement of primitive update operations becomes trivial: each operation is placed in the appropriate Sequence Diagram; invariant rules can be used to check if all required operations are present in a particular action under consideration.

• Completely redundant structures can be removed afterwards anyway (if necessary).

This may provide some additional opportunities in the development of a knowledge-based system design tool.
6. CONCLUSION.

In this article we indicated how the dynamically oriented models generated by the Jackson System Development method and the static models that are provided by the Entity-Relationship approach may be used in a complementary way.

With the help of a small illustrative problem, we have shown that the three types of obstacles encountered during the specification of a model in JSD (multithreading clash, multiple roles and constraints involving several entities) have a formal counterpart in data modelling in the form of the three data abstractions association, aggregation and generalization. Using two transformation rules it is possible to transform an arbitrary ER Diagram into an equivalent diagram, matching the JSD specifications.

Starting from the different constructs found in the ER model, seems to give JSD more flexibility in the modelling of behaviour: structures are more straightforward, maintenance becomes easier. Besides, the ER approach provides a strong basis for further database design, a point that was neglected completely in JSD. JSD on the other hand may add to the static ER approach its dynamic modelling capabilities, but a new role concept is desired to avoid some inelegant constructs, which may occur if one tries to model dynamic constraints in terms of the existing ER abstractions.

Before we can talk of a really integrated tool however, a number of unsolved problems, as well as some new opportunities must be investigated further.
REFERENCES


[28] P.P. Chen, ER - a Historical Perspective and Future Directions, in [18], pp. 71-77.


