How to Formalize It?
Formalization Principles for Information System Development Methods

A.H.M. ter Hofstede[1] and H.A.Proper[2]

[1]Faculty of Information Technology
Queensland University of Technology
GPO Box 2434, Brisbane, Q 4001
Australia
arthur@icis.qut.edu.au

[2]Origin B.V.
PO Box 7377
1007 JJ Amsterdam
The Netherlands
e.proper@acm.org

Published as: A.H.M. ter Hofstede and H.A. Proper. How to Formalize It? Formalization Principles for Information Systems Development Methods. Technical report #5/97, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, June 1997.

Abstract

Although the need for formalisation of modelling techniques is generally recognised, not much literature is devoted to the actual process involved. This is comparable to the situation in mathematics where focus is on proofs but not on the process of proving. This paper tries to accomodate for this lacuna and provides essential principles for the process of formalisation in the context of modelling techniques as well as a number of small but realistic formalisation case studies.

Keywords: Formalization, Methodologies, Information Systems

1 Introduction

Traditionally, information systems has been a rather informal field of study and many methods and techniques have been introduced without a formal foundation. The past decade, however, has seen a shift from informal approaches to more formal approaches towards information systems development. On the one hand, a number of new formal approaches have been introduced and on the other hand a number of initially informal techniques have been given a formal foundation. Nowaydays a vast body of literature exists justifying the need for formality (see e.g. [Jon86, Spi88, Coh89, SFMS89, Hal90a, HE90, BH95]) and this paper assumes that the reader is thoroughly convinced thereof.

Although the concept of proof is at the heart of mathematics, relatively few attention is given to the notion of a *correct* and *elegant* proof. Furthermore, hardly any literature at all is devoted to the intuitive process of proving: How to find a proof for a given problem? A notable exception is Polya's "How to Solve It" [Pol57]. In this seminal work, the famous mathematician George Polya addresses the task of finding correct, elegant, and general solutions for mathematical problems. Inspired by this work, we have set ourselves a similar task trying to facilitate the process of formalizing methods and techniques in the field of information systems development. Naturally, our ambition level is far beneath that of Polya's and emphasis is on formalization principles and their application in a number of concrete case studies. In that sense the paper may be considered a sequel to [HW92], where emphasis was on the need for formalization and only a small number of very small case studies was presented.

The main justification of this paper is the observation that the important process of formalization is very complicated and typically requires substantial experience and mathematical maturity. It is also a process that is usually not addressed in university curricula and many formalization experts are basically self-taught.

In this era of emphasis on applied research, this paper makes an indirect contribution only. An empirical proof that a better understanding of the process of formalization leads to improved methods and ultimately to better systems is extremely hard to give. Again, the reader is expected to be convinced that this is true.

The paper is organized as follows. Section 2 provides a definition of important formalization notions and a discussion of five major formalization principles. Sections 3 and 4 provide case studies illustrating these principles. Section 3 focusses on the formalization of a technique for process control specifications, while section 4 provides examples of the formalization of a conceptual data modelling technique. The paper ends with conclusions and topics for further research.

2 On Syntax and Semantics

Syntax defines what well-formed models are, while semantics is concerned with the meaning of well-formed models. In section 2.1 concrete syntax and abstract syntax are contrasted, while section 2.2 discusses four different ways of assigning a formal semantics. Finally, section 2.3 provides a definition of five formalization principles.

2.1 Concrete versus Abstract Syntax

In this section the distinction between concrete and abstract syntax is defined and illustrated. Further, it will be emphasized why it is important to focus on an abstract rather than a concrete syntax in the formalization of modelling techniques.

In the definition of the syntax of programming languages it is customary to use BNF. A definition of a conditional would typically look like:

$$
\begin{array}{ll}
<\mathit{conditional}> \triangleq & \textbf{if } <\mathit{boolean\_expression}> \\
 & \textbf{then } <\mathit{statement}> \\
 & \textbf{else } <\mathit{statement}> \\
 & \textbf{endif}
\end{array}
$$

In this definition, an explicit choice for *keywords* is made as well as an explicit choice for the *order* of the various parts in a conditional. Both confuse the underlying deeper meaning of what a conditional is about. Different programming languages may use different keywords and may even use a different order of the various parts mentioned above, the underlying meaning, however, remains essentially the same: a conditional represents a choice between two program parts depending on the evaluation of a certain boolean expression.

An abstract syntax definition simply gives the components of a language construct and would leave out representational details. For the conditional this could look like:

$$
\begin{array}{ll}
<\mathit{conditional}> \triangleq & \text{thenbranch: } <\mathit{statement}>; \\
 & \text{elsebranch: } <\mathit{statement}>; \\
 & \text{condition: } <\mathit{boolean\_expression}>.
\end{array}
$$

In a sense, the abstract syntax can be compared to what is referred to as the *conceptual level* in the famous three level architecture of [Gri82]. The conceptual level focusses on concepts as they are and not on how they are perceived (*external level*) or implemented (*internal level*). In [Mey90], Betrand Meyer stresses that

> *The use of abstract syntax rather than concrete syntax as a basis for studies of programming languages is representative of an important trend in software engineering: the move towards a higher-level view of software objects, emphasising deep structure rather than surface properties. Concepts such as abstract data types are another example of this trend.*

While leaving the phase of infancy behind in the field of information systems, it is clear that a similar shift from representational issues to semantical issues has to be made. Often discussions are blurred as a result of mixing representation with essence. Questions whether an entity type should be represented as a square or as a circle are conceptually irrelevant. In some cases the way representational issues are interwoven with essential issues are more subtle. In section 4 this is illustrated by the formalization of relationship types in the context of conceptual data modelling.

In the field of information systems, abstract syntax could be defined by the use of grammars, but also, typically, by the use of set theory or graph theory. A model could be defined as a many-sorted algebra with a number of sets, functions, relations and possibly some special constants, as well as a number of axioms that have to be satisfied. These axioms could be specified by means of logic. A state transition diagram for example, could consist of a set of states, say $\mathcal{S}$, a set of transitions, a relation, say $\mathsf{Trig} \subseteq \mathcal{S} \times \mathcal{S}$, and a begin state $s_0 \in \mathcal{S}$. An axiom like

$$(s, s_0) \notin \mathsf{Trig}$$

would prevent transitions to the begin state. Typically, these axioms correspond to what is called *static semantics* in the field of programming languages (see e.g. [Mey90]).

2.2 Assigning a Formal Semantics

Having defined an abstract syntax for a certain modelling technique, the next step is the definition of a formal semantics. Often one will find though that these two steps are not strictly sequential. A choice for a certain semantics may lead to a different design of the abstract syntax, one which leads to more elegant formulations (this cannot always be foreseen).

There are many styles of assigning a formal semantics. The following approaches, taken from [Mey90], are relevant here:

- *Translational semantics*: In translational semantics models specified in a certain modelling technique are given a semantics by the definition of a mapping to models of a simpler language, a language which is better understood.
- *Operational semantics*: If a translational semantics amounts to a compiler for a modelling technique, an operational semantics is like an interpreter. The idea is to express the semantics of a modelling technique by giving a mechanism that allows to determine the effect of any model specified in the technique. Such a mechanism can be seen as an "interpreting automaton".
- *Denotational semantics*: Denotational semantics may be seen as a variant of translational semantics. In this method the syntactic constructs of a language are mapped onto constructs in another language with a well-defined meaning. In contrast with translational semantics, the "target" of denotational semantics is a mathematical domain and not another modelling technique.
- *Axiomatic semantics*: An axiomatic semantics treats a model as a logical theory, it does not try to define what the model means, but what can be proved about the model only. As such, this approach might be seen as more abstract than denotational.

In the context of the formalization of modelling techniques each of these approaches will be illustrated in section 3, for task structures.

Clearly, each of these approaches to defining semantics requires a choice of a suitable target domain. Logic/set theory for example, would not constitute a good target domain for dynamic approaches as they are not really suitable to deal with *change*. However, the choice of a target domain not only depends on the type of modelling technique to be formalized, but also on the goal of the formalization. An operational semantics is perfectly acceptable if one aims at rapid development of a support tool. This way of defining semantics is less suitable, however, if focus is on investigating and proving properties about a modelling technique. In section 4, three different domains for defining the semantics (in a denotational way) of conceptual data modelling are investigated.

2.3 Formalization Principles

In this section five major formalization principles are outlined. Each of these principles is defined and discussed at a fairly high level of abstraction. In the two following sections, they are illustrated by means of some case studies. Principles imply guidelines and heuristics and occasionally some of these are stated. No attempts are made though to reach any form of "completeness" (one may even question whether that would be possible at all).

Principle – *Primary Goal Principle*

> Determine the primary objective of the formalization and choose style and mathematical domain accordingly. □

There may be many reasons for formalizing a method or technique. These reasons include educational purposes, investigation of properties, establishing a sound basis for sophisticated automated support etc. The *Primary Goal Principle* emphasizes that the style and domain is to be chosen on the basis of the primary reason for formalization. An operational semantics is typically desirable if efficient and correct implementations are needed, while a denotational semantics may be of more use for investigating and proving formal properties of models. Similarly, the formalization goal determines the formal domain that is most suitable. For educational purposes, highly abstract mathematical domains (e.g. category theory) may not be very suited. This might also be true when the formalization goal is to arrive at a blueprint of an implementation of a support tool. In general, the theory and notation available in a certain domain deserves a close study as it may considerably facilitate, not only the formalization process, but also the manipulation of the resulting formalization.

To discuss this matter in more detail, let us consider some concrete examples. In case of the formalization of techniques for conceptual data modelling, such as ER and NIAM, possible mathematical domains are logic, set theory, and category theory (each of these will be used in the case study of section 4). Logic is particularly useful for educational purposes, but could also favorably be considered in the context of a PROLOG implementation. Set theory is more suitable in case the formalization is required to be concise and formal properties need to be investigated. Category then could be exploited as an even more abstract system, formalization is intellectually more demanding, but the rewards are reduction in proof complexity (among others because of the principle of duality) and true generality (this will be shown in section 4.5).

For dynamic modelling techniques, domains like logic (exception: specialized logics, such as temporal logic and dynamic logic) and set theory are less suited. Category theory can still be used, but typically, domains like Petri nets and higher-order variants thereof and algebraic approaches such as Process Algebra [BW90], CCS [Mil89], and CSP [Hoa85] are particularly suitable. Petri net-based approaches provide formal verification techniques (such as S-invariants), but a limited number of concepts, while algebraic approaches typically provide a wide range of operators and theories for determining equivalence of process specifications (e.g. bisimulation), but usually less convenient means for process verification.

Principle – *Semantics Priority Principle*
Focus should be on semantics, not on syntax. □

In a formalization it is imperative to focus on formal semantics and to choose those syntactic elements that facilitate the specification of this semantics. It is important not to get carried away by imposing possible syntactic restrictions to exclude models that at first may seem undesirable. The main reason for this is that at the early stages of formalization it is hard to characterize "undesirable" models and to provide a clear justification for this. Often one finds that situations that might seem undesirable at first do make sense from a semantic point of view and in some cases even add flexibility and expressive power. Recursive decomposition in task structures (extensively discussed in section 3) provides an example. For those used to traditional Data Flow Diagrams decomposition should be hierarchical. As it turns out, however, process specifications can be made more elegant by allowing recursive decomposition and in addition to that recursive decomposition increases the expressive power of task structures (this is formally shown in [HO97]).

As a rule, only dismiss those models as not being well formed to which you cannot possibly assign a formal semantics. Models that are initially well formed but "problematic" can then be formally characterized in semantic terms. As an example consider the data modelling technique NIAM [Hal95]. Naturally, conceptual data models that cannot be populated/instantiated are undesirable. As it turns out however the formalization of NIAM in [BHW91] showed that populatability is a complex notion which has several gradations. These could only be expressed after an initial definition of what a population is had been given. Similarly, process control specifications with deadlock or livelock may not be desirable, yet a syntactic characterization of these specifications may be very hard to give (or not even possible).

Principle – *Conceptualisation Principle*
Choose representations that are as general as possible. □

In an interesting prologue in [Sch94] the author argues that the essence of computing might be summed up in the word *generality*. Ultimately, computing has one fundamental objective: to study and to achieve generality. Programs that have been set up in a general way are often easily adaptable to new requirements and specifications that are general are easier to understand than those laced with details and choices which are too specific. In the field of data modelling one may argue that the notion of generality is closely related to the famous Conceptualisation Principle [Gri82] which states that conceptual models should deal only and exclusively with aspects of the problem domain. Any aspects irrelevant to that meaning should be avoided, as this might lead to exclusions of possible solutions in too early a stage.

The concepts of conceptualization and generality translate directly to formalizations. In a formalization any choice irrelevant to the modelling technique under consideration should be avoided. In particular, unnecessary representational choices should not obscure the essential meaning of concepts. An example in the context of the formalization of relationship types is given in section 4. True generality can be achieved if the semantics of a modelling technique is "configurable", i.e. if the addition of semantic features does not require a complete redesign of the formalization. For example, the addition of time, uncertainty, or missing values to a data modelling technique. An example of how this can be achieved is discussed in section 4.5.

Another instance of the Conceptualisation Principle is the issue of abstract versus concrete syntax. Although concrete syntax is important and necessary from a communication point of view, it introduces non-essential elements and as such violates the Conceptualisation Principle.

Principle – *Orthogonality Principle*

Choose as few concepts as possible as the basis for the formalization. □

In formalizations it is preferable to keep the number of concepts small. This keeps the formalization concise and facilitates subsequent proofs as it reduces the number of possible case distinctions.

At this point it is important to stress that it is absolutely imperative for formalizations to be readable. Formalizations that are not readable cannot be used for communication nor educational purposes and are difficult to check for correctness. Orthogonality is one of the prerequisites for achieving this goal. Another important prerequisite is the availability of a suitable set of auxiliary operators, functions, relations etc. that allow for compact formulation of complex expressions. This ties in with the next principle which advocates a bottom up approach carefully defining the necessary mathematical base first such that the important rules of the formalization can subsequently be captured in *one line*.

Principle – *Bottom Up Principle*

Work bottom up and start with a base of suitable syntactic elements such that subsequent semantic rules can be formulated easily and do not require more than *one line*. □

Naturally, this principle derives from sound engineering practice as well as structured programming and is perhaps slightly too strict in the sense that in practice often a mix of working top down and bottom up is the most convenient way to go. However, it is very important to invest substantially in a sound base theory and to investigate properties of several of the elementary mathematical objects used. In addition to that, it might in some cases even be useful to introduce proof schemes. Of course, this principle ties in with the Conceptualisation Principle in the sense that well-chosen generalizations tend to shorten specifications considerably too.

3 Case Study I: Formalizing Task Structures

In this section the formalization of a dynamic modelling technique, *task structures*, is studied. The four different ways of assigning a semantics, as presented in the previous section are illustrated:

1. Translational semantics by means of a mapping of task structures to Petri nets (section 3.3);
2. Operational semantics by means of a task structure interpreter specified in Predicate/Transition nets (section 3.4);
3. Denotational semantics by means of a mapping of task structures to Process Algebra (section 3.5);
4. Axiomatic semantics by means of a mapping of task structures to logical derivation rules (section 3.6).

First, however, task structures are explained informally, and a formal syntax in terms of first order predicate logic is presented.

3.1 Introduction to Task Structures

Task structures were introduced in [Bot89] to describe and analyze problem solving processes. In [WH90, WHO92] they were extended and used as a meta-process modelling technique for describing the strategies used by experienced information engineers. In [HN93] they were extended

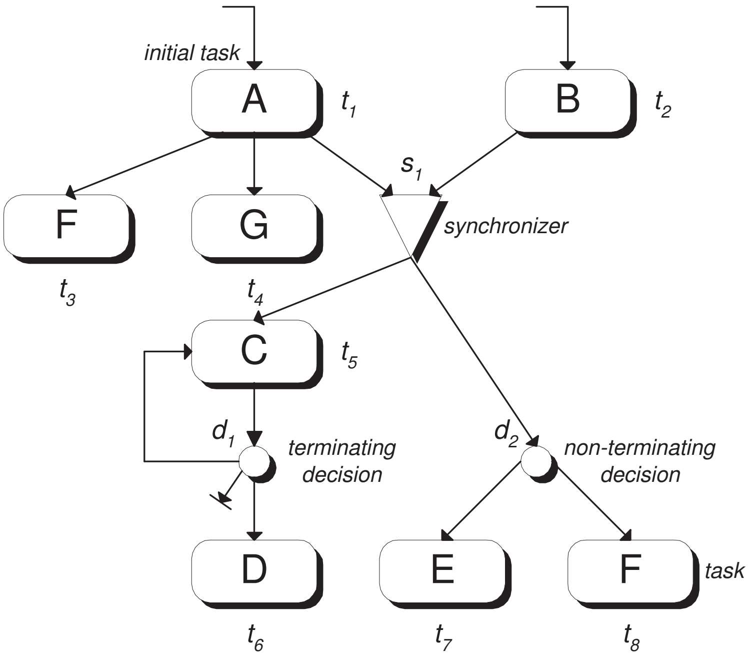

Figure 1: An example task structure

again and a formal semantics in terms of Process Algebra was given. In figure 1, the main concepts of task structures are represented graphically. They are discussed subsequently.

The central notion in task structures is the notion of a *task*. A task is defined as something that has to be performed in order to achieve a certain goal. A task has a name representing its functionality; naturally, different tasks may have the same name. In addition, a task can be defined in terms of other tasks, referred to as its *subtasks*. This *decomposition* may be performed repeatedly until a desired level of detail has been reached. For the purpose of this paper though, decomposition of tasks will not be considered.

Performing a task may involve choices. *Decisions* represent these moments of choice, they coordinate the execution of tasks. Two kinds of decisions are distinguished, *terminating* and *non-terminating* decisions. A decision that is terminating, may lead to termination of the execution path of that decision. If this execution path is the only active execution path, the task as a whole terminates as well.

Triggers, graphically represented as arrows, model sequential order. Multiple output triggers are a means to specify parallelism. After the execution of the task with name A for example, both the task with name F and the task with name G are triggered in parallel. *Initial items* are those tasks or decisions, that have to be performed first. Due to iterative structures, it may not always be clear which task objects are initial. Therefore, this has to be indicated explicitly (in the example, the tasks with names A and B are initial). Finally, *synchronisers* deal with explicit synchronisation. In figure 1 the task named C (and decision d_2 for that matter) can only start when the tasks with names A and B have terminated.

3.2 Syntax of Task Structures

In this section the syntax of task structures (without decomposition) is defined using elementary set theory.

Formally, a task structure consists of the following components:

1. A set $\mathcal{X}$ of task objects. $\mathcal{X}$ is the union of a set of synchronisers $\mathcal{S}$, a set of tasks $\mathcal{T}$ and a set of decisions $\mathcal{D}$. In $\mathcal{D}$ we distinguish a subset $\mathcal{D}_t$ consisting of the terminating decisions. For convenience, we define the set $\mathcal{U}$, the set of non-synchronisers, as $\mathcal{T} \cup \mathcal{D}$.
2. A relation $\mathsf{Trig} \subseteq \mathcal{X} \times \mathcal{X}$ of triggers.
3. A function $\mathsf{TName} : \mathcal{T} \rightarrow \mathcal{N}$ yielding the name of a task, where $\mathcal{N}$ is a set of names.
4. A subset $\mathcal{I}$ of the set of non-synchronisers $\mathcal{U}$, consisting of the initial items.

Although additional well formedness rules could be specified, e.g. excluding non-terminating decisions without output triggers or excluding non-reachable task objects, they are not needed from a semantical point of view (Semantics Priority Principle). As will be shown it is possible to assign a semantics to task structures not satisfying these rules.

Example 3.1

The task structure of figure 1 is formally captured by

$$\begin{aligned}
\mathcal{X} &= \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, s_1, d_1, d_2\},\\
\mathcal{T} &= \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8\},\\
\mathcal{S} &= \{s_1\},\\
\mathcal{D} &= \{d_1, d_2\},\\
\mathcal{D}_t &= \{d_1\}.
\end{aligned}$$

Further, $t_1 \;\mathsf{Trig}\; s_1$, $t_2 \;\mathsf{Trig}\; s_1$, $t_1 \;\mathsf{Trig}\; t_3$ etc, and $\mathsf{TName}(t_1) = A$, $\mathsf{TName}(t_2) = B$, etc. Finally, $\mathcal{I} = \{t_1, t_2\}$.

3.3 Semantics of Task Structures through Petri Nets

Petri nets (see e.g. [Pet81] or [Rei85]) are a technique for modelling communication between parallel processes, developed by C.A. Petri in the sixties [Pet62]. Petri nets consist of places and transitions. Places model states, while transitions model state changes. Applications of Petri nets can be found in many areas such as communication protocols, process synchronisation and resource allocation. Recently, Petri nets are also used for requirements specification (see e.g. [Dav90]).

Formally, a Petri Net (in its most elementary form) is a four tuple $\langle P, T, I, O\rangle$, where P is a set of places, T is a set of transitions, $I : T \rightarrow \wp(P)$ is the input function, a mapping from transitions to sets of places, and $O : T \rightarrow \wp(P)$ is the output function, also a mapping from transitions to sets of places. A marking of the net is an assignment of tokens to the places. Firing rules determine how transitions may change markings. A transition can fire if and only if each of its input places contains at least one token. In case of firing, these tokens are removed and tokens are placed into the output places.

Mapping task structures to elementary Petri nets is relatively straightforward. Each task $t \in \mathcal{T}$ is mapped onto a place $E_t \in P$, and a transition C_t is created which has as input place E_t and as output places all places corresponding to the task objects triggered by that task (if such places exist!). An exception is the treatment of synchronizers. For each synchronizer $s \in \mathcal{S}$ and each task object $x \in \mathcal{X}$ such that $x \;\mathsf{Trig}\; s$, a place with the name $\sigma_{x,s}$ is created. Synchronisation is now achieved by creating a transition H_s which has all these places as input places and has as output places the places corresponding to the task objects triggered by that synchronizer. Each decision $d \in \mathcal{D}$ is mapped to a place E_d and has for each of its choices $e \in \mathcal{X}$ an arrow to a unique transition $G_{d,e}$ which has an outgoing arrow to place E_e. If d is terminating as well, there is an arrow from the place corresponding to that decision to a transition F_d without output places (if that transition fires it will simply consume a token from that place). Finally, the initial marking of the net is a marking with exactly one token in each of the places E_i with i an initial item. The following definition captures this mapping formally.

Definition 3.1

Given a task structure $\mathcal{W}$, the corresponding Petri net $\mathcal{P}_{\mathcal{W}} = \langle P_{\mathcal{W}}, T_{\mathcal{W}}, I_{\mathcal{W}}, O_{\mathcal{W}} \rangle$ is defined by:

$$
\begin{aligned}
P_{\mathcal{W}} &= \{E_x \mid x \in \mathcal{T} \cup \mathcal{D}\} \cup \{\sigma_{x,s} \mid x \,\mathsf{Trig}\, s \wedge s \in \mathcal{S}\} \\
T_{\mathcal{W}} &= \{C_t \mid t \in \mathcal{T}\} \cup \{F_d \mid d \in \mathcal{D}_t\} \cup \{G_{d,e} \mid d \,\mathsf{Trig}\, e \wedge d \in \mathcal{D}\} \cup \{H_s \mid s \in \mathcal{S}\} \\
I_{\mathcal{W}} &= \{(C_t, \{E_t\}) \mid t \in \mathcal{T}\} \cup \{(F_d, \{E_d\}) \mid d \in \mathcal{D}_t\} \cup \\
&\quad \{(G_{d,e}, \{E_d\}) \mid d \in \mathcal{D}\} \cup \{(H_s, \{\sigma_{x,s} \mid x \,\mathsf{Trig}\, s\}) \mid s \in \mathcal{S}\} \\
O_{\mathcal{W}} &= \{(C_t, \{E_x \mid t \,\mathsf{Trig}\, x \wedge x \notin \mathcal{S}\} \cup \{\sigma_{t,s} \mid t \,\mathsf{Trig}\, s \wedge s \in \mathcal{S}\}) \mid t \in \mathcal{T}\} \cup \\
&\quad \{(G_{d,e}, \{E_e \mid e \notin \mathcal{S}\} \cup \{\sigma_{d,e} \mid e \in \mathcal{S}\}) \mid d \in \mathcal{D}\} \cup \\
&\quad \{(H_s, \{E_x \mid s \,\mathsf{Trig}\, x \wedge x \notin \mathcal{S}\} \cup \{\sigma_{s,s'} \mid s \,\mathsf{Trig}\, s' \wedge s' \in \mathcal{S}\}) \mid s \in \mathcal{S}\} \cup \\
&\quad \{(F_d, \varnothing) \mid d \in \mathcal{D}_t\}
\end{aligned}
$$

The initial marking assigns tokens to E_{t_1} and E_{t_2}.

The above definition provides a *translational* semantics for task structures. One may argue whether this definition really adds anything to the explanation given before, it certainly is not readable without it.

Example 3.2

Figure 2 contains the result of applying the previously described translation to the task structure of figure 1.

3.4 Semantics of Task Structures through Predicate/Transition Nets

In [HL91] two principal problems with Petri nets are stated. The first problem is that Petri net specifications easily become huge, complicated and unstructured. The second problem is their structural inflexibility, making modification extremely difficult. In order to overcome these problems high-level Petri nets have been developed. Examples are *Predicate Transition nets* ([Gen87]) and *Coloured Petri nets* ([Jen87, Jen91]). In this section we will define an *operational* semantics of task structures in terms of Predicate Transition nets. First however, we will give a brief informal explanation of this type of high-level Petri net.

Contrary to Petri nets, tokens in a Predicate Transition net are individualized and as such they may have a structure. Arrows may be labeled with linear combinations of tuple variables (e.g. $2\langle x, y\rangle + \langle y, z\rangle$) and transitions may have associated transition selectors (e.g. $y \le z$). These transition selectors are to be interpreted in terms of a formal structure, referred to as the *support* of the net (usually a first-order structure).

A transition can fire if and only if a combination of tokens can be found in its input places conforming to the arrow annotations, such that the transition selector evaluates to true. Tokens will then be produced for the output places again conforming to the annotations of the arrows.

As a simple example of a Predicate Transition net, consider figure 3. The support of this net consists of the structure $\langle\{a, b, c\}, <\rangle$, where $<$ captures alphabetic ordering. The marking resulting from firing the transition with substitution $(x, y, z) \leftarrow (a, b, c)$ is shown in figure 4.

The expressive power of Predicate Transition nets has been illustrated in [Gen87] by means of a simple example involving tracks and trains. In this example there is a circular track consisting of seven segments. Two trains may move along this track, but they should never be in the same nor in adjacent segments. The Petri net solution consisted of 21 places and 14 transitions, while the Predicate Transition net solution consisted only of 2 places and 1 transition. In addition to that, the Predicate Transition solution could easily be generalized to n segments with m trains (the

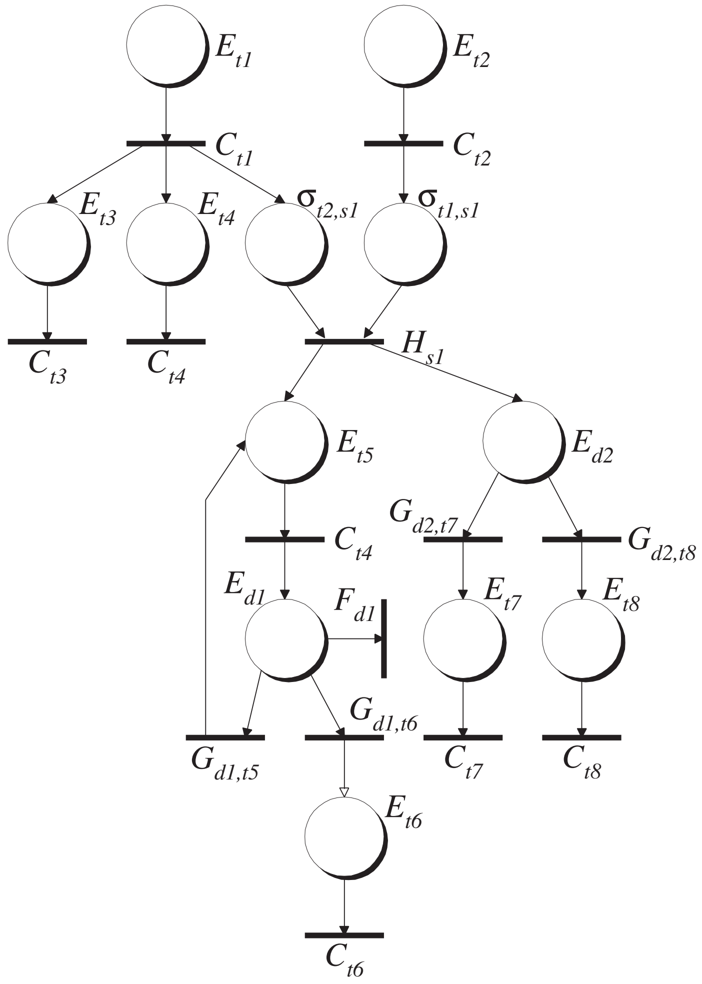

Figure 2: Petri Net translation of Task Structures

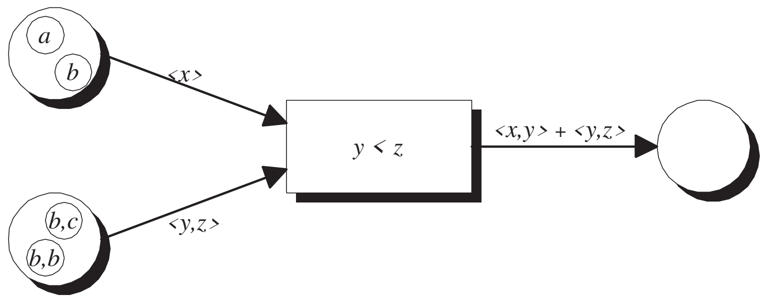

Figure 3: Simple Predicate Transition net with initial marking

number of places and transitions would remain the same). This solution was obtained by applying a number of "folding" transformations to the Petri net solution. Basically, this same idea can be applied in the context of assigning formal semantics. The Petri nets resulting from the translation described in the previous section can be *generalized* into one Predicate Transition net which might be viewed upon as a task structure interpreter. This interpreter is shown in figure 5. It provides an *operational* semantics to task structures.

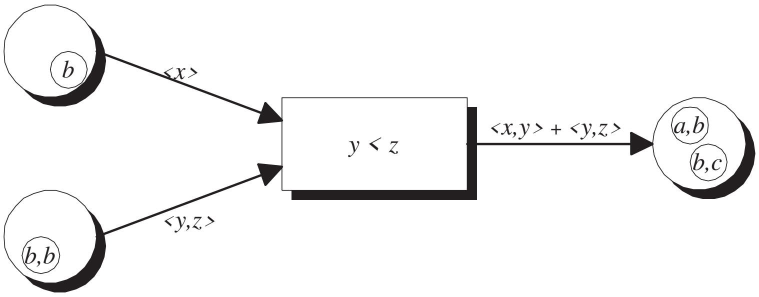

Figure 4: Simple Predicate Transition net with initial marking after firing of transition

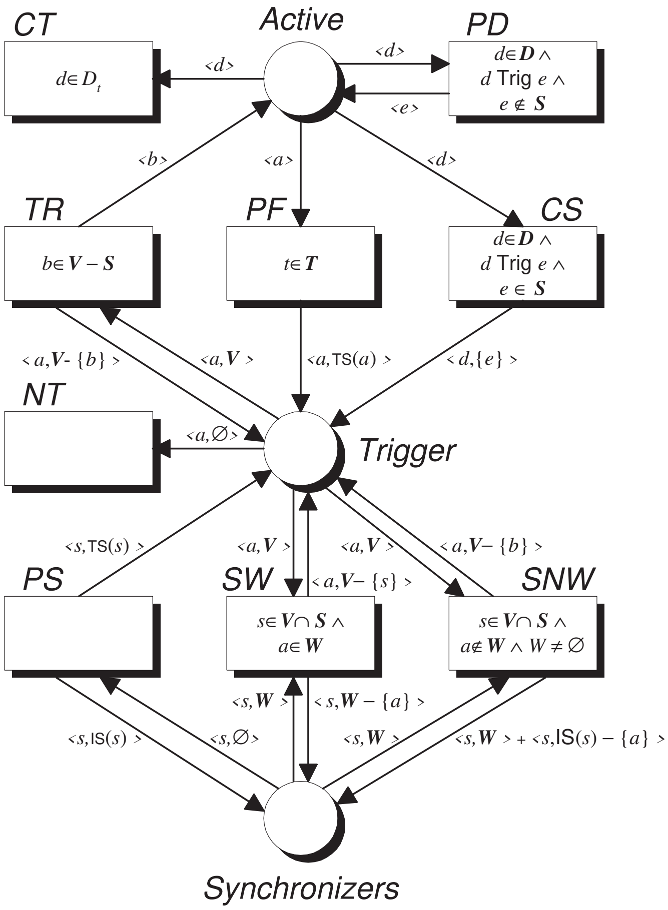

Figure 5: Predicate/Transition net semantics for Task Structures

The Predicate Transition net of figure 5 contains a place *Active* which is to contain tokens corresponding to active instances of tasks and decisions. The place *Synchronizers* is to contain tokens of the form $\langle s, W\rangle$, where s is a synchronizer and W is a set of task objects that s is waiting for. A token corresponding to a decision may be replaced by a token corresponding to a non-synchronizer that is among the possible choices of that decision (firing transition PD). Choosing a synchroniser requires that that decision is removed from the waiting list for that synchronizer (this is achieved through firing transition CS). If the decision is terminating, the token may also be removed altogether (firing transition CT).

An active task must fire, upon termination, all its successors. This is achieved by transition *PF* having output place *Trigger*, which contains tokens of the form $\langle t, V \rangle$, where t is a task and V is a set of task objects (still) to be triggered by t. The annotation of the corresponding output trigger uses the auxiliary function TS, yielding all task objects to be triggered by a task object, which is formally defined by:

$$\mathsf{TS}(x) = \{ y \in \mathcal{X} \mid x \,\mathsf{Trig}\, y \}$$

If a successor of t is a non-synchronizer, this successor becomes an active token in *Active* and is removed from V (by firing transition *TR*). If it is a synchronizer s, then it is either removed from the set W in a token corresponding to that synchronizer (transition SW), or, if there is no such set W, a token of the form $\langle s, W - \{t\} \rangle$ is added to the place *Synchronizers* (transition SNW). Here the auxiliary function IS is needed which yields all the task objects input for a synchroniser:

$$\mathsf{IS}(s) = \{ y \in \mathcal{X} \mid y \,\mathsf{Trig}\, s \}$$

This latter situation corresponds to the situation where t is the first task object to activate synchronizer s, which then has to await termination of its other input task objects.

A synchronizer s can become active if all its input task objects have terminated, and in terms of the Predicate Transition net of figure 5 this means that there is a token $\langle s, \varnothing \rangle$ in the place *Synchronizer*. All output task objects then have to be activated, which can be achieved in the same way as for tasks (transition *PS* is enabled). Note that a task, or synchronizer, x with no output task objects would result in a token $\langle x, \varnothing \rangle$ in the place *Trigger*. Such tokens are simply removed by firing transition *NT*.

The initial marking of the net is the marking which assigns tokens corresponding to the initial items to the place *Active*. Further, in order to evaluate the transition selectors, the syntactic structure has to be provided. This syntactic structure acts as the support of the net and simply corresponds to the syntactic definition of a task structure as defined in section 3.2.

Example 3.3

By providing the Predicate Transition net of figure 5 with the syntax of the task structure of figure 1 and an initial marking which assigns the tokens $\langle t_1 \rangle$ and $\langle t_2 \rangle$ to the place *Active*, the operationale semantics of this task structure is completely defined.

3.5 Semantics of Task Structures through Process Algebra

In this section the semantics of task structures is defined by means of a translation to Process Algebra ([BW90]). Process Algebra has been applied to specify and verify a variety of processes and protocols (see e.g. [Bae90]). One of its virtues is its ability to prove equivalence of process specifications. As an algebraic theory Process Algebra belongs to the same family as CCS [Mil89] and CSP [Hoa85].

First a short introduction to the important ingredients of Process Algebra is given (for an in-depth treatment we refer to [BW90]), then the translation of task structures to Process Algebra is defined (based on [HN93]). This provides an example of *denotational* semantics.

Although the name Process Algebra suggests a single algebra to describe processes, it actually refers to a whole family of algebras based on the same principles. Traditionally, only the family member used is presented. In this section we briefly introduce the Algebra of Communicating Processes with the empty action (ACP_ε) as the semantics of task structures will be defined in this algebra. Again, for an in-depth treatment we refer to [BW90]. The axioms of ACP_ε are listed in table 1.

The units of Process Algebra are atomic actions. The set of all atomic actions is called $\mathcal{A}$. Although they are units of calculation, atomic actions need not be indivisible (see [GW96]). Starting

$X+Y=Y+X$	A1	$X+\delta=X$	A6
$(X+Y)+Z=X+(Y+Z)$	A2	$\delta\cdot X=\delta$	A7
$X+X=X$	A3	$X\cdot\varepsilon=X$	A8
$(X+Y)\cdot Z=X\cdot Z+Y\cdot Z$	A4	$\varepsilon\cdot X=X$	A9
$(X\cdot Y)\cdot Z=X\cdot(Y\cdot Z)$	A5		
$X\parallel Y=X \mathbin{\|\!\lfloor} Y+Y \mathbin{\|\!\lfloor} X+X\mid Y+\surd(X)\surd(Y)$	C1	$aX\mid bY=(a\mid b)(X\parallel Y)$	C7
$\varepsilon \mathbin{\|\!\lfloor} X=\delta$	C2	$(X+Y)\mid Z=X\mid Z+Y\mid Z$	C8
$aX \mathbin{\|\!\lfloor} Y=a(X\parallel Y)$	C3	$X\mid(Y+Z)=X\mid Y+X\mid Z$	C9
$(X+Y) \mathbin{\|\!\lfloor} Z=X \mathbin{\|\!\lfloor} Z+Y \mathbin{\|\!\lfloor} Z$	C4	$a\mid b=\gamma(a,b)$ if $\gamma(a,b)$ defined	C10
$\varepsilon\mid X=\delta$	C5	$a\mid b=\delta$ otherwise	C11
$X\mid\varepsilon=\delta$	C6		
$\surd(\varepsilon)=\varepsilon$	TE1	$\partial_H(a)=a$ if $a\notin H$	D1
$\surd(a)=\delta$	TE2	$\partial_H(a)=\delta$ if $a\in H$	D2
$\surd(X+Y)=\surd(X)+\surd(Y)$	TE3	$\partial_H(X+Y)=\partial_H(X)+\partial_H(Y)$	D3
$\surd(X\cdot Y)=\surd(X)\cdot\surd(Y)$	TE4	$\partial_H(X\cdot Y)=\partial_H(X)\cdot\partial_H(Y)$	D4

Table 1: Algebra of Communicating Processes with the empty action

with atomic actions, new processes can be constructed by applying sequential and alternative composition ("$\cdot$" resp. "+"). The algebra that results (axioms A1-A5) is called basic process algebra (BPA). As a convention, the names of atomic actions are written in lowercase (e.g. a, b, red_nose_reindeer), while process variables are written in uppercase (e.g. A, B, RUDOLPH). Normally, the $\cdot$ will be omitted unless this results in ambiguity. A special constant δ, *deadlock*, denotes the inaction, or impossibility to proceed (axioms A6-A7).

To add parallelism, an additional operator has to be introduced. This operator, called (free) *merge* and denoted as $\parallel$, is defined with the aid of an auxiliary operator $\mathbin{\|\!\lfloor}$, the *left-merge* (axioms C1-C4).

Another special constant ε, the *empty action*, is used to denote the process that does nothing but terminate successfully (axioms A8-A9). After adding ε, processes may terminate directly. The *termination operator* $\surd$ determines whether or not this termination option is present for a given process (axioms TE1-TE4). The inclusion of the expression $\surd(X)\surd(Y)$ in axiom C1 guarantees that $\varepsilon\parallel\varepsilon=\varepsilon$.

The communication merge $\mid$ allows parallel processes to exchange information, i.e. to communicate (axioms C5-C9). Associated with communication is a *communication function* γ defined over pairs of atomic actions (axioms C10-C11). Specific process specifications will have to define the (partial) communication function γ. This function is both commutative and associative. Axioms C10 and C11 assume that communication is binary, but higher order communication is permitted as well.

Finally, one needs a way to prevent the isolated occurrence of atomic actions meant to communicate with other actions. This is achieved through the *encapsulation operator* ∂_H. In fact it is a whole family of operators, one for each $H\subseteq\mathcal{A}$ (axioms D1-D4).

In the translation of task structures to equations in ACP$_\varepsilon$, for every task object $x\in\mathcal{X}$ an object E_x is introduced. Intuitively, E_x should be seen as an entry point for task object x.

For every task $t\in\mathcal{T}$ the corresponding process algebra equation expresses that the associated atomic action (the name of that task) has to be performed and that afterwards all task objects to be triggered are triggered in parallel. The treatment of synchronizers brings a slight complication as in order to become active they may have to await termination of other task objects. This problem is solved through a rather unusual use of the communication operator. Every task object x which triggers synchroniser w starts an atomic action $\sigma_{x,w}$ after termination. A synchroniser is started upon availability of all the atomic actions of its input task objects. Therefore, the communication function has to be defined for the σ's of each synchroniser. As a synchroniser can have more than two input task objects, communication is not necessarily binary, which is quite uncommon indeed. Binary communication could be used at the expense of some additional definitions. The equation for a task $t \in \mathcal{T}$ is:

$$E_t \quad = \quad \mathsf{TName}(t) \;\cdot\; \left(\mathop{\Big\|}_{u \in \mathcal{U}, \mathsf{Trig}(t,u)} E_u \;\Big\|\; \mathop{\Big\|}_{s \in \mathcal{S}, \mathsf{Trig}(t,s)} \sigma_{t,s} \right)$$

As can be seen, tasks now produce a unique atom for all their output synchronisers. The set of atomic actions $\mathcal{A}$ is defined as $\mathsf{ran}(\mathsf{TName})$. To comply with the notational conventions of Process Algebra, we denote the elements of $\mathcal{A}$ in lowercase.

It should be noted that some tasks do not trigger any other task objects. In that case, the merge is specified over the empty set. This exception is dealt with by defining the merge over the empty set to be the empty action ε as this is the neutral element for the merge. Therefore, if no task objects x exist, such that $\mathsf{Trig}(t, x)$, the equation for task t reduces to $E_t = \mathsf{TName}(t)$.

For every terminating decision $d \in \mathcal{D}_t$ we have the following equation in Process Algebra:

$$E_d \quad = \quad \sum_{u \in \mathcal{U}, \mathsf{Trig}(d,u)} E_u + \sum_{s \in \mathcal{S}, \mathsf{Trig}(d,s)} \sigma_{d,s} \;+\; \varepsilon$$

The execution of a terminating decision involves the choice between one of the task objects that are output of that decision and termination. Although not very likely from a practical point of view, in theory it is possible that a decision does not have any output triggers. To comply with the Semantics Priority Principle, we therefore define the sum over the empty set to be δ as this is the neutral element with respect to summation.

The equation for non-terminating decisions is quite similar, except that termination is no option. Hence the substitution of ε by δ:

$$E_d \quad = \quad \sum_{u \in \mathcal{U}, \mathsf{Trig}(d,u)} E_u + \sum_{s \in \mathcal{S}, \mathsf{Trig}(d,s)} \sigma_{d,s} \;+\; \delta$$

Note that if a non-terminating decision does not have any output triggers, it will inevitably result in a deadlock. Otherwise, the δ is removed by axiom A6.

The definition of a synchroniser comprises three parts: an equation, a definition of the communication function and its encapsulation set. The equation for a synchroniser $s \in \mathcal{S}$ is similar to the equation for a task, except that no action is needed:

$$E_s \quad = \quad \mathop{\Big\|}_{u \in \mathcal{U}, \mathsf{Trig}(s,u)} E_u \;\Big\|\; \mathop{\Big\|}_{s' \in \mathcal{S}, \mathsf{Trig}(s,s')} \sigma_{s,s'}$$

The communication function is defined as:

$$\mathop{\Big|}_{x \in \mathcal{X}, \mathsf{Trig}(x,s)} \sigma_{x,s} \quad = \quad E_s$$

From this it is clear that all input task objects have to be ready before the synchroniser is initiated. It should be noted that the above equation is only present for synchronisers that can be triggered by other task objects, i.e. $s \in \mathcal{S}$ such that $\{x \in \mathcal{X} \mid \mathsf{Trig}(x, s)\} \neq \varnothing$.

In order to assure that the σ's are used only to trigger the synchroniser, the encapsulation operator, ∂_H, has to be applied. The set of communication atoms H is $\{\sigma_{x,s} \mid \mathsf{Trig}(x,s) \wedge x \in \mathcal{X} \wedge s \in \mathcal{S}\}$. The whole system $\mathcal{W}$ can now be captured by the equation:

$$\mathcal{W} = \partial_H \left(\mathop{\Big\|}_{i \in \mathcal{I}} E_i \right)$$

Example 3.4

Consider the task structure of figure 1. Application of the previously described translation yields:

$$\begin{array}{lcllcl}
E_{\mathcal{W}} & = & \partial_{\{\sigma_{t_1,s_1},\sigma_{t_2,s_1}\}}\,(E_{t_1} \parallel E_{t_2}) & E_{d_1} & = & E_{t_6} + E_{t_5} + \varepsilon \\
E_{t_1} & = & a \cdot (E_{t_3} \parallel E_{t_4} \parallel \sigma_{t_1,s_1}) & E_{t_6} & = & d \\
E_{t_2} & = & b \cdot \sigma_{t_2,s_1} & E_{s_1} & = & E_{d_2} \parallel E_{t_5} \\
E_{t_3} & = & f & E_{d_2} & = & E_{t_7} + E_{t_8} + \delta \\
E_{t_4} & = & g & E_{t_7} & = & e \\
E_{t_5} & = & c \cdot E_{d_1} & E_{t_8} & = & f
\end{array}$$

where

$$\sigma_{t_1,s_1} \mid \sigma_{t_2,s_1} = \sigma_{t_2,s_1} \mid \sigma_{t_1,s_1} = E_{s_1}$$

3.6 Semantics of Task Structures through Logical Derivation Rules

In this section an *axiomatic* approach to the semantics of task structures is taken. Task structures are mapped onto a set of formal derivation rules, which define that under certain conditions a formula may be derived from another formula. These derivations represent *state transitions.* In this context a formula basically represents a *state* of the task structure. Formally, such a state is captured by the number of active instances of each task object and the number of signals sent to synchronizers by their input task objects, i.e. S is a state implies that

$$S : \mathcal{X} \rightarrowtail \mathbb{N} \cup \mathsf{Trig}[\mathcal{X} \times \mathcal{S}] \rightarrowtail \mathbb{N}$$

The square brackets denote function restriction in this definition. Ignoring traces, the semantics of a task structure simply is the set of states that can be derived from the derivation rules.

To easily manipulate states, we need to introduce some notation (Bottom Up Principle). Firstly, states can change. The notation $S \oplus V$ denotes a function similar to S except that its values that are in the domain of V are redefined:

$$S \oplus V = S[\mathsf{dom}(S) - \mathsf{dom}(V)] \cup V$$

This definition can be used to capture the auxiliary function incr which increases the number of active instances by one in a state S for all task objects satisfying a certain predicate $P(x)$:

$$\mathsf{incr}(S, x, P(x)) = S \oplus \{x \mapsto x + 1 \mid P(x)\}$$

Similarly decr decreases the number of active instances in a state S by one for all task objects satifying a predicate $P(x)$. In some cases the number of active instances has to be increased/decreased for a specific task object c; in that case the notation $\mathsf{incr}(S, c)$ (or $\mathsf{decr}(S, c)$) is used as an abbreviation for $\mathsf{incr}(S, x, x = c)$ (or $\mathsf{decr}(S, x, x = c)$).

Triggering a task or a synchroniser x in a state T increases the number of active instances of all its output task objects by one:

$$\mathsf{Fire}(T, y) = \mathsf{incr}(T, x, y \;\mathsf{Trig}\; x \wedge x \notin \mathcal{S}) \cup \mathsf{incr}(T, (y, s), y \;\mathsf{Trig}\; s \wedge s \in \mathcal{S})$$

First one instance of every initial item becomes active:

$$\frac{}{\{(i,1) \mid i \in \mathcal{I}\}}\Big[$$

Firing a task or synchroniser $x \in \mathcal{T} \cup \mathcal{S}$ corresponds to increasing the number of its active output task objects and decreasing the number of active instances of x:

$$\frac{S}{\mathsf{Fire}(\mathsf{decr}(S,x),x)}\Big[\, S(x) > 0$$

Firing a decision $d \in \mathcal{D}$ corresponds to making a choice for one of its possible output task objects e ($d\,\mathsf{Trig}\,e$) which is not a synchroniser ($e \notin \mathcal{S}$):

$$\frac{S}{\mathsf{decr}(\mathsf{incr}(S,e),d)}\Big[\, S(d) > 0$$

Choosing a synchroniser s ($d\,\mathsf{Trig}\,s \wedge s \in \mathcal{S}$) has to be dealt with in a separate rule:

$$\frac{S}{\mathsf{incr}(S,(d,s))}\Big[\, S(d) > 0$$

Choosing termination ($d \in \mathcal{D}_t$) corresponds to:

$$\frac{S}{\mathsf{decr}(S,d)}\Big[\, S(d) > 0$$

For a synchroniser $s \in \mathcal{S}$ to fire, all its input task objects need to have fired at least once:

$$\frac{S}{\mathsf{incr}(\mathsf{decr}(S,(x,s),x\,\mathsf{Trig}\,s),s)}\Big[\, x\,\mathsf{Trig}\,s \Rightarrow S(x,s) > 0$$

In an earlier version of this paper, this latter rule was formulated as:

$$\frac{S}{(S \oplus \{s \mapsto S(s)+1\}) \oplus \{(x,s) \mapsto S(x,s)-1 \mid x\,\mathsf{Trig}\,s\}}\Big[\, x\,\mathsf{Trig}\,s \Rightarrow S(x,s) > 0$$

This again illustrates the importance of defining suitable auxiliary functions and relations (Bottom Up Principle).

3.7 Comparison

Thi section provided, among others, an extensive illustration of the Primary Goal Principle: the goal of formalization determines to a large extent the formalization formalism. In case of the formalization of task structures, the formalization in terms of Petri nets has the advantage of being relatively easy (Petri nets themselves are also easily explained and understood). In addition to that, a lot of theory is available for the formal analysis of Petri nets and this theory can be applied to analyze the Petri nets resulting from the translation to determine properties of the original task structures (e.g. whether they are free of deadlock). While a substantial amount of theory is available for the formal analysis of Predicate Transition nets, the Predicate Transition net specifying the formal semantics of task structures is much less suited for such formal analysis as analysis of high-level Petri nets is much more complex (from a computational point of view) than elementary Petri nets.

The main advantage of the Predicate Transition net formalization is its operational nature, which allows for a straighforward implementation (analysis can then be performed by means of simulation). In [Wij91] a PROLOG implementation of task structures is described which has been formally derived from a Predicate Transition net formalization (very similar to the one described in section 3.4, except that decomposition and an abort mechanism were formally specified as well, while on the other hand there were no synchronizers).

The Process Algebra formalization has as advantage that it is relatively compact and may exploit equivalence theory developed for Process Algebra. In [HN93] the theory of bisimulation is used to demonstrate how task structures can be proven to be (in)equivalent. When working on [HOR96], the first author experienced great difficulties in using the Process Algebra formalization for proving properties (in particular deadlock freedom) of task structures that satisfied certain requirements. It seems very likely that the Process Algebra formalization is not useful in a context where proofs are to found for certain *classes* of task structures as this requires reasoning over equation sets of a specific form. For this purpose, the axiomatic semantics proves to be much more suitable (see also [HOR96]). However, if one considers natural extensions to task structures, such as task decomposition, messaging, and time constructs, the axiomatic semantics becomes very hard to define properly and it is best to exploit the wide range of operators supplied by Process Algebra (see e.g. [HH96]).

4 Case Study II: Formalizing ORM

In this section a formalization of a conceptual data modelling technique is studied. The aim of this study is twofold. The first aim is to provide a further illustration of the use of denotational semantics to define the semantics of a modelling technique (contrary to the previous section it concerns a *static* modelling technique). The second aim is to show the importance of the Conceptualization Principle when formalizing this semantics, both on the "microlevel" (the formalization of relationship types) and the "macrolevel" (the use of category theory to achieve a configurable semantics).

The subject of our case study is Object Role Modelling (ORM) [Hal95]. The reason for choosing ORM over the more popular ER family of data modelling techniques is that ORM is a richer modelling language. Although formalizations of the syntax of an information modelling technique are important, they tend to be relatively simple. Most information modelling techniques have a fairly straightforward syntax. Formalization of semantics, if done at all, usually is a much more complicated issue and thus provides us with a more interesting case. Nonetheless, to study the definition of ORM's semantics, a formalization of the syntax is needed. Therefore, only a brief, and standardized, formalization of the relevant parts of the syntax is provided.

In this section solely *denotational* semantics is studied, but with three different domains: logic, set theory, and category theory.

4.1 Informal explanation of ORM

Before formalizing the semantics, a brief discussion of the technique itself is required. For that, focus is on some of the key modelling concepts in ORM.

Two key concepts in data modelling are *object types* and *relationship types*. Generally, a relationship type is considered to represent an association between object types. A relationship type may have any arity (> 0). In figure 6, the graphical representation of a binary relationship type R between object types A and B is shown in the both the ER and ORM style.

A relationship type (in an ORM context also referred to as a fact type), consists of a number of *roles*, represented by the boxes, determining the way those object types participate in that relationship type. Relationship types may be *objectified*, which means that they may play roles in other relationship types.

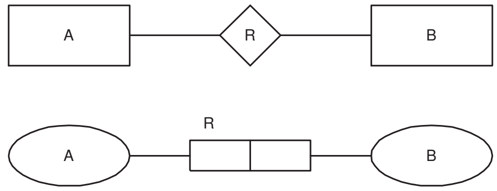

Figure 6: A binary relationship type in ER and ORM

Many conceptual data modelling techniques offer concepts for expressing inheritance of properties [HP95]. In the literature many kinds of inheritance relations exist and terminology is far from being standardized. In this section we only use subtyping.

Subtyping is used when specific facts are to be recorded for specific instances of an object type only. For example, in a data model with persons, one may wish to express that only for Adults, i.e. persons at least 18 years old, the Cars they own are to be recorded. In that case *Adult* becomes a subtype of object type *Person*, and only instances of *Adult* can participate in the relationship registering Car ownership. This is captured by the schema in figure 7.

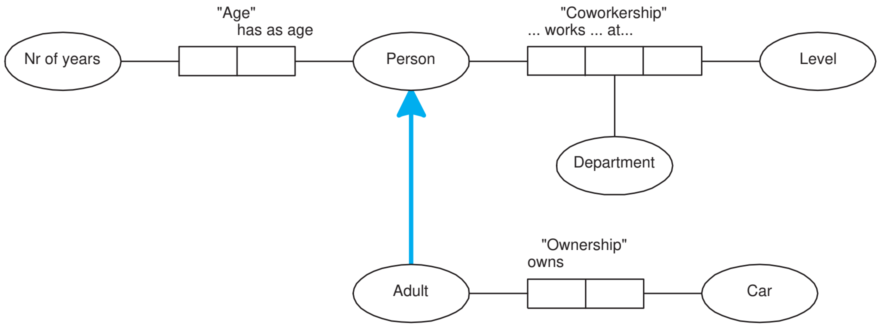

Figure 7: An example of a domain with subtyping

A subtype inherits the properties of its supertype(s), but may have additional properties. *Subtype defining rules* serve as decision criteria for determining which instances of supertypes participate in subtypes. Therefore, they are to be considered population derivation rules. The availability of such rules allows subtypes to be viewed as derivable object types.

Although ORM features numerous types of constraints that restrict the set of allowed database populations [Hal95], they will not be part of this case study neither will be concrete object types (sometimes also referred to as label types or value types) and identification schemes.

4.2 Syntax of ORM

An ORM information structure syntactically consists of the following elements:

1. A finite set $\mathcal{R}$ of *roles*.
2. A nonempty finite set $\mathcal{O}$ of *object types*.
3. A set of relationship types $\mathcal{F} \subseteq \mathcal{O}$.
4. A function $\mathsf{Roles} : \mathcal{F} \to \mathcal{R}^+$, which should provide a partition of roles over the relationship types.

5. A function $\mathsf{Player} : \mathcal{R} \rightarrow \mathcal{O}$, which yields the object type playing a certain role.
6. A binary relation $\mathsf{SubOf} \subseteq (\mathcal{O} - \mathcal{F}) \times \mathcal{O}$ capturing subtyping. $x \,\mathsf{SubOf}\, y$ is to be interpreted as "x is subtype of y". This relation is required to be acyclic.

Two auxiliary functions will prove to be useful:

1. The function $\_[\_] : \mathcal{F} \times \mathbb{N} \rightarrow \mathcal{R}$ yields the role on the given position in a relationship type, and is identified by: $f[i] \triangleq \mathsf{Roles}(f)_i$.
2. $\mathsf{Rel} : \mathcal{R} \rightarrow \mathcal{F}$ yields the relationship type in which a given role is contained, and is defined by: $\mathsf{Rel}(p) = f \Leftrightarrow \exists_{i \in \mathbb{N}}[p = f[i]]$.

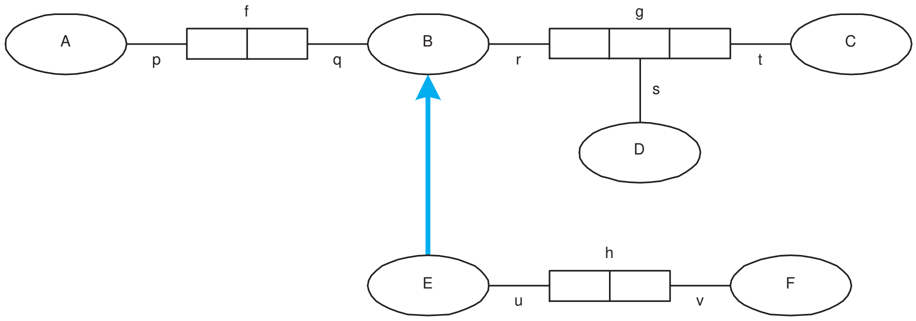

Figure 8: Abstract schema example

Example 4.1

Figure 8 shows the graphical representation of an information structure, i.e. an information structure diagram*. This diagram is actually the same as the one from figure 7, where we now focus on the mathematical constructs of the schema.*

This schema is defined by:

$$\begin{aligned} \mathcal{R} &= \{p, q, r, s, t, u, v\} \\ \mathcal{O} &= \{A, B, C, D, E, F, f, g, h\} \\ \mathcal{F} &= \{f, g, h\} \end{aligned}$$

Where we have:

$$\mathsf{Roles}(f) = \langle q, p \rangle, \mathsf{Roles}(g) = \langle r, s, t \rangle, \mathsf{Roles}(h) = \langle u, v \rangle$$

For players of roles we have:

$$\mathsf{Player}(p) = A, \mathsf{Player}(q) = B, \mathsf{Player}(r) = B,$$

etc. Furthermore, $E \,\mathsf{SubOf}\, B$.

One question that one may already ask is if there is really a need to represent the roles involved in a relationship type as a *sequence*. Isn't this just a set? In the remainder of this paper, a discussion of this will be provided.

4.3 A logic based formalization of ORM

Examples of using a logic based approach to express the semantics of ORM can be found in [Hal89] and [De 93, De 96]. Using a logic based formalization has a direct advantage in that it, for example, allows for automatic reasoning to prove the equivalence of conceptual schemas [Hal91, Blo93] using a theorem prover. Furthermore, experience with teaching shows that this is a very intuitive way for students to see the relationship between logic and NIAM, thus strengthening their understanding of both formalisms.

The first formalization of ORM, using the style of [Hal89, Hal90b], closely follows the syntax of ORM. The basic idea is to interpret an ORM schema as a first order logic theory. This means that a schema should be transformed to a set of formulae; a database population then is a set of additional (atomic) formulae that is consistent with the schema. Any constraints formulated on the conceptual schema would also be translated to appropriate formulae in the theory.

The obvious thing to do would be to let each n-ary relationship type in a ORM schema result in one n-ary predicate, and each object type (except for relationship types) result in a unary predicate. For figure 7 this would yield the following predicates:

$$\textsf{Person-has-as-age-Nr-of-years}(x,y), \textsf{Person-works-for-Dept-at-Level}(x,y,z), \textsf{Person-owns-Car}(x,y)$$

In figure 7 we would have the following unary predicates for object types:

$$\textsf{Nr-of-years}(x), \textsf{Person}(x), \textsf{Adult}(x), \textsf{Department}(x), \textsf{Level}(x), \textsf{Car}(x)$$

The semantics of the schema could then be expressed by the following first order theory:

$$\begin{aligned}
\textsf{Person-has-as-age-Nr-of-years}(x,y) &\Rightarrow \textsf{Person}(x) \wedge \textsf{Nr-of-years}(y) && (1)\\
\textsf{Person-works-for-Dept-at-Level}(x,y,z) &\Rightarrow \textsf{Person}(x) \wedge \textsf{Department}(y) \wedge \textsf{Level}(z) && (2)\\
\textsf{Adult-owns-Car}(x,y) &\Rightarrow \textsf{Adult}(x) \wedge \textsf{Car}(y) && (3)
\end{aligned}$$

$$\begin{aligned}
\textsf{Adult}(x) &\Rightarrow \textsf{Person}(x) && (4)\\
\textsf{Person-has-as-age-Nr-of-years}(x,y) \wedge y \geq 21 &\Rightarrow \textsf{Adult}(x) && (5)
\end{aligned}$$

The first block of rules (1-3) are conformity rules. For each relationship it is required that the instances used are valid instances of the object types playing a role in the relationship type. Therefore, in general we have for each $f \in \mathcal{F}$ with $\textsf{Roles}(f) = \langle p_1, \ldots, p_n \rangle$ a rule of the format:

$$f(x_1, \ldots, x_n) \Rightarrow \textsf{Player}(p_1)(x_1) \wedge \ldots \wedge \textsf{Player}(p_n)(x_n)$$

Here we also see how this particular formalization *needs* the predefined order on the roles in the relationship types. The order of the variables in the predicate corresponds to the order chosen for the roles.

Rules 4 and 5 deal with subtyping. Each adult is by definition a person, while as a rule each person older than 21 is to be considered an adult. In general, for each a SubOf b the following rule holds: $a(x) \Rightarrow b(x)$. If subtype a has as subtype defining rule L, then it should also be the case that: $L(x) \Rightarrow a(x)$.

Example 4.2

An example population would be:

Person(Erik), Person(Mark),
Department(Information Systems), Department(Computing Services),
Level(2), Level(1),

Nr-of-Years(19), Nr-of-Years(29),
Adult(Erik), Car(925DJX),

Person-has-as-age-Nr-of-Years(Erik,29),
Person-has-as-age-Nr-of-Years(Mark,19),
Person-works-for-Dept-at-Level(Erik,Information Systems,1),
Person-works-for-Dept-at-Level(Mark,Computing Services,2),
Adult-owns-Car(Erik,925DJX)

This style of formalizing has the advantage that it closely follows the syntactical structure of ORM. From an educational point of view this seems to be a wise strategy. However, looking at the conceptualization principle, this formalization can be viewed as having two major drawbacks. A first drawback is that it presumes there is always a pre-determined order of the roles in a relationship type. For example:

Person x works for Department y at Level z

versus

At Department y works Person x at Level z

Depending on the aim of the formalization (conceptualization vs. educational purposes), this may be acceptable or not. A decoupling between the surface structure (concrete syntax!) of information and deep structure of information seems to be a good strategy to follow when expressing the semantics more conceptually.

The second drawback of the above formalization style is the way in which objectification has to be treated. ORM allows instances of relationship types to be treated as instances that may play a role in other relationship types. This is illustrated in figure 9. An objectified (n-ary) relationship type behaves, by its very nature, both like an object type and a relationship type. Therefore, it must yield both an n-ary predicate and a unary predicate when mapping to a logic theory. For instance, the schema in figure 9 yields both a binary predicate f and a unary predicate D. In [Hal89] this situation is resolved by introducing the rule:

$$D(x) \Leftrightarrow x = \langle x_1, x_2 \rangle \ \wedge f(x_1, x_2)$$

Even more so than in the case of using a pre-defined order on the roles in predicates, the above rule is a violation of the conceptualization principle. The rule unnecessarily reveals the underlying physical structure of instances (tuples). There is no need to state that instances of an objectified relationship type are in fact tuples.

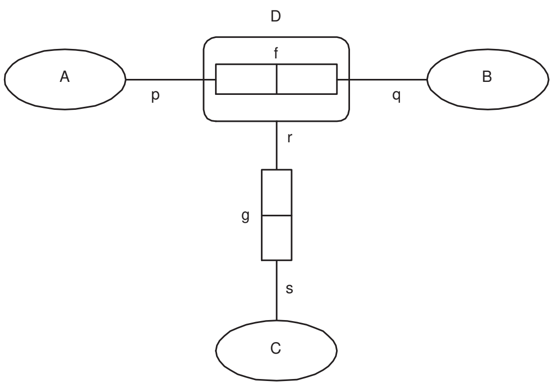

Figure 9: Objectified relationship type

When further abstracting away from the syntax, some of the above drawbacks can be removed. However, due to the higher level of abstraction, the connection between the schema as an expression in a graphical language, and the formalization becomes less apparent. This may impede badly on educational goals.

Consider again the objectified relationship from figure 9. We can translate a schema by producing for each type (both object types and relationship types) a unary predicate, and for each role a binary predicate. For this example, this leads to predicates:

$$A(x), B(x), C(x), f(x), g(x), p(x,y), q(x,y), r(x,y), s(x,y)$$

If we have $p(x,y)$, then we can interpret this as: x plays role p in relationship y.

An important difference with the previous approach is that instances of relationships and roles are treated explicitly as instances, which will have an advantage for objectifications. The downside of this approach is that the connection to the graphical presentation becomes less obvious. On the upside, the order in which roles occur in a relationship type is no longer of importance.

As a concrete example, for figure 7 the translation now becomes:

$$\begin{array}{rcl}
\textsf{Nr-of-Years-is-Age}(x,y) & \Rightarrow & \textsf{Nr-of-years}(x) \wedge \textsf{Age}(y) \\
\textsf{Person-has-Age}(x,y) & \Rightarrow & \textsf{Person}(x) \wedge \textsf{Age}(y) \\
\textsf{Person-involved-in-Coworkership}(x,y) & \Rightarrow & \textsf{Person}(x) \wedge \textsf{Coworkership}(y) \\
\textsf{Department-has-Coworkership}(x,y) & \Rightarrow & \textsf{Department}(x) \wedge \textsf{Coworkership}(y) \\
\textsf{Level-of-Coworkership}(x,y) & \Rightarrow & \textsf{Level}(x) \wedge \textsf{Coworkership}(y) \\
\textsf{Adult-has-Ownership}(x,y) & \Rightarrow & \textsf{Adult}(x) \wedge \textsf{Ownership}(y) \\
\textsf{Car-with-Ownership}(x,y) & \Rightarrow & \textsf{Car}(x) \wedge \textsf{Ownership}(y) \\
\\
\textsf{Age}(y) & \Rightarrow & \exists_x \left[\textsf{Nr-of-Years-is-Age}(x,y)\right] \\
 & \wedge & \exists_x \left[\textsf{Person-has-Age}(x,y)\right] \\
\textsf{Coworkership}(y) & \Rightarrow & \exists_x \left[\textsf{Person-involved-in-Coworkership}(x,y)\right] \\
 & \wedge & \exists_x \left[\textsf{Department-has-Coworkership}(x,y)\right] \\
 & \wedge & \exists_x \left[\textsf{Level-of-Coworkership}(x,y)\right] \\
\textsf{Ownership}(y) & \Rightarrow & \exists_x \left[\textsf{Adult-has-Ownership}(x,y)\right] \\
 & \wedge & \exists_x \left[\textsf{Car-with-Ownership}(x,y)\right] \\
\\
\textsf{Adult}(x) & \Rightarrow & \textsf{Person}(x) \\
\textsf{Person-has-as-age-Nr-of-years}(x,y) \wedge y \geq 21 & \Rightarrow & \textsf{Adult}(x)
\end{array}$$

Although, the actual theory corresponding to a conceptual schema may have become more complicated we now have a cleaner way in which to deal with objectifications. Moreover, the order in which roles appear in relationship types has become irrelevant.

The logic theory for a given schema can be derived as follows. For any role $p \in \mathcal{R}$ the following holds:

$$p(x,y) \Rightarrow \textsf{Player}(p)(x) \wedge \textsf{Rel}(p)(y)$$

Furthermore, any instance of a relationship type $r \in \mathcal{F}$ with $\textsf{Roles}(r) = \{p_1, \ldots, p_n\}$ (note that we dropped the order of the roles) should have some associated role instance:

$$r(y) \Rightarrow \exists_x \left[p_1(x,y)\right] \wedge \ldots \wedge \exists_x \left[p_n(x,y)\right]$$

In this formalization style, an example population would be:

Person(Erik), Person(Mark),
Department(Information Systems), Department(Computing Services),

Level(2), Level(1),
Nr-of-Years(19), Nr-of-Years(29),
Adult(Erik), Car(925DJX),

Age(#01), Age(#02), Coworkership(#03), Coworkership(#04),
Ownership(#05), Ownership(#06)

Person-has-Age(Erik,#01), Nr-of-Years-is-Age(29,#01),
Person-has-Age(Mark,#02), Nr-of-Years-is-Age(19,#02),
Person-involved-in-Coworkership(Erik,#03),
Department-with-Coworkership(Information Systems,#03),
Level-of-Coworkership(1,#03),
Person-involved-in-Coworkership(Mark,#04),
Department-with-Coworkership(Computing Services,#04),
Level-of-Coworkership(2,#04),
Adult-has-Ownership(Erik,#05), Car-with-Ownership(925DJX,#05)

Here the instances #01, ..., #05 are placeholders for some abstract instance that represents the underlying relationship instance.

4.4 A set theoretic formalization of ORM

The set-theoretic approach views a data model as consisting of an information structure describing the structure of information elements, and a set of constraints. A population is a mapping from object types to sets of instances (conforming to the structural requirements and the constraints). The semantics of a data model is the set of all possible populations. This style of formalization can be found in e.g. [BHW91, HW93].

The set theoretic style of formalization also closely follows the syntax of a schema. However, unlike the initial logic based approach, this formalization does not rely on the order of roles in a relationship type. A schema is treated as a set of abstract concepts and the semantics are expressed without explicit reference to names or orderings of roles in relationship types. Therefore, the formalization below will not be using the fact that Roles returns a sequence. All that is needed is a set of roles: $\mathsf{Roles} : \mathcal{F} \to \wp(\mathcal{R})$. This approach is clearly better with regards to the conceptualization principle.

Formally, a population Pop of a data model assigns to each object type a finite set of instances, such that all constraints are satisfied. The instances originate from an underlying universal domain Ω. This universe of instances is inductively defined as the smallest set satisfying:

1. A set $\Theta \subseteq \Omega$ of atomic instances.
2. If $x_1, \ldots, x_n$ are instances from Ω, and $r_1, \ldots, r_n$ are roles, then the mapping

$$\{r_1 \mapsto x_1, \ldots, r_n \mapsto x_n\}$$

   is also an instance (note that a mapping is a set of tuples, the tuple $\langle r_i, x_i \rangle$ is denoted as $r_i \mapsto x_i$). These mappings are intended for the population of relationship types.

The population of a relationship type is a set of tuples. A tuple t in the population of a relationship type h is a mapping of all its roles to values, i.e. $t : \mathsf{Roles}(h) \to \Omega$. For example, for the fact that Adult 'Erik' owns car '925DJX', this leads to:

$$t(u) = \mathsf{Erik} \text{ and } t(v) = \mathsf{925DJX}$$

where $\mathsf{Roles}(h) = \{u, v\}$, and h is the relationship type $\mathsf{Ownership}$. Note that in this approach the treatment of objectified relationships is transparent. An instance of a relationship type is a mapping, and may in itself (see definition of Ω) occur as an object in another mapping for another relationship.

The requirement that a value assigned to a role should occur in the population of the object type playing that role is captured by the *conformity rule*:

$$x \in \mathcal{F} \Rightarrow \forall_{y \in \mathsf{Pop}(x), p \in x} \left[y(p) \in \mathsf{Pop}(\mathsf{Player}(p)) \right]$$

Therefore, as $t(u) = \mathsf{Erik}$, we should have $\mathsf{Erik} \in \mathsf{Pop}(E)$, as $\mathsf{Player}(u) = \mathsf{E}$. The *Specialization Rule* expresses which instances participate in a subtype:

$$x \,\mathsf{SubOf}\, y \Rightarrow \mathsf{Pop}(x) = \left\{ v \in \mathsf{Pop}(y) \;\middle|\; R_x(v) \right\}$$

if R_x is the subtype defining rule for subtype x.

This style of formalization is quite elegant and simple in its setup and is basically derived from what is referred to as the *mapping oriented approach* in [Mai88] towards the formalization of the Relational Model [Cod70]. This approach guarantees nice properties of algebraic operators, such as associativity and commutativity of joins.

However, what can be identified as a drawback of this style is the fact that it reveals that instances of relationship types are "implemented" by means of mappings: $t : \mathsf{Roles}(f) \to \Omega$. This is, of course, already a direct consequence of the style used in he relational model, where tables are modeled as sets (not sequences!) of attributes, and each row in the population of a table is a mapping of these attributes to values. In the second logic based formalization style, we have seen that there is absolutely no need to make such assumptions; leading to a more conceptual formalization. In the next formalization style, the category theoretic approach, we will see an even cleaner approach to the problem of hiding unnecessary detail when formalizing.

4.5 A category theoretic formalization of ORM

A more abstract formalization approach usually requires readers to be able to understand the abstractions used, as well as the formalism. In the case of logic we saw how a more conceptual formalization could be obtained by dropping the ordering of roles in a relationship as well as removing the dependence on a tuple representation for objectified relationships. In the set theoretic approach we have essentially seen a repeat of this. This abstraction led to a cleaner formalization of relationship types with regards to the Conceptualization Principle. One could view this as an application of this principle on the "micro" level. As will be shown, the use of category theory provides another, far more substantial, advantage. The semantics becomes "configurable", a choice for desired semantic features (e.g. time stamps, null values, uncertainty, etc), simply becomes a choice for an appropriate instance category and does not require a complete redesign of the formalization (as it would in case of the previously presented formalizations). The idea of configurable semantics provides an example of generalization on the "macro" level.

Category theory is a relatively young branch of mathematics designed to describe various *structural* concepts from different mathematical fields in a *uniform* and highly conceptual way. Category theory offers a number of concepts, and theorems about those concepts, that form an abstraction of many concrete concepts in diverse branches of mathematics. As pointed out by Hoare [Hoa89]: "Category theory is quite the most general and abstract branch of pure mathematics". The resulting level of genericity and abstraction may, however, make category theory formalizations harder to understand for an "average" reader. If the goal of the formalization is not the communication of a formal definition to a wide audience, and there is a strong preference for a highly conceptual formalization, then category theory will be the most ideal instrument. However, if a wider audience needs to be addressed, a more mundane formalization may be more appropriate.

Goguen [Gog91] argued that category theory can provide help with *dealing with abstraction and representation independence*: in computing science, more abstract viewpoints are often more useful, because of the need to achieve independence from the overwhelmingly complex details of how things are represented or implemented. This is particularly relevant in the context of this case study. Category theory allows the study of the essence of certain concepts as it focuses on the *properties* of mathematical structures instead of on their *representation*. This point will be illustrated later when the formalization of relationship types is discussed.

Category theory, therefore, is an interesting approach when defining the semantics of a data modelling technique. Category theory allows us to define these semantics in a highly abstract way, focussing on the essential properties. The example formalization as discussed here, is based on the work reported in [FHL97, LH96, HLF96, HLW97]. For this case study, however, only a fraction of this work is needed and consequently only a few categorical notions are required.

A *category* $\mathcal{C}$ is a directed multigraph whose nodes are called *objects* and whose edges are called *arrows*. For each pair of arrows $f : A \rightarrow B$ and $g : B \rightarrow C$ there is an associated arrow $g \circ f : A \rightarrow C$, the *composition* of f with g. Furthermore, $(h \circ g) \circ f = h \circ (g \circ f)$ whenever either side is defined. For each object A there is an arrow $\mathsf{Id}_A : A \rightarrow A$, the *identity* arrow. If $f : A \rightarrow B$, then $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$.

$$\begin{tikzcd} A \arrow[loop left, "\mathsf{Id}_A"] \arrow[r, "f"] & B \arrow[loop right, "\mathsf{Id}_B"] \end{tikzcd}$$

Figure 10: A simple example of a category

In figure 10 a simple example of a category is shown. It is an abstract example, no assumptions about the meaning of the objects and the arrows have been made (and indeed, have to be made!). In this category the choice of composites is forced: $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$. The objects and arrows of a category may also have a concrete interpretation. For example, objects may be mathematical structures such as sets, partially ordered sets, graphs, trees etc. Arrows can denote functions, relations, paths in a graph, etc.

In the context of this paper, some set-oriented categories are important. The most elementary and frequently used category is the category **Set**, where the objects are sets and the arrows are total functions. The objects of **Set** are not necessarily finite. The category whose objects are *finite* sets and whose arrows are total functions is called **FinSet**. The category **PartSet** concerns sets with *partial* functions, while the category **Rel** has sets as objects and binary *relations* as arrows.

For a categorical formalization, it is necessary to define a uniform syntax of conceptual data models that is as general as possible. Conceptual data models are defined by means of *type graphs* (see also [Sie90] and [Tui94]). Type graphs provide a highly conceptualized view on data models and as will be shown in definition 2, they enable a very elegant formal definition of populations (as such, this again illustrates the Semantics Priority Principle).

The various object types in the data model correspond to nodes in the graph, while the various constructions can be discerned by labeling the arrows. Relationship types, for example, correspond to nodes. An object type participating via a role in a relationship type is target of an arrow labeled with role, which has as source that relationship type. As an object type may participate via several roles in a relationship type a type graph has to be a *multi*graph.

Definition 4.1

A *type graph* $\mathcal{G}$ is a directed multigraph over a label set $\{\mathsf{role}, \mathsf{spe}\}$. Edges with label spe are called *subtype* edges, The type graph may not contain cycles consisting solely of subtype edges. The function type yields the label of an edge.

An edge e, labeled with role, from a node A to a node B indicates that A is a relationship type in which B plays a role. If e is labeled with spe, then A is a specialization of B.

An important observation with respect to the definition of the type graph is that it is very liberal. It allows a binary relationship type to be a subtype of a ternary fact type for example. As will be shown assigning a semantics to type graphs containing such relations is not a problem and from an OO perspective they are even very useful (Semantics Priority Principle),

Example 4.3

As an example of how data models can be represented as type graphs, consider figure 11, which shows the type graph of the ORM data model in figure 7.

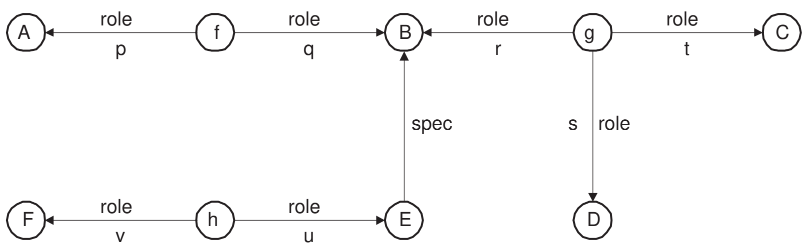

Figure 11: An example typegraph

In this approach, a population is defined as a *model* from the type graph to a category. A model is a graph homomorphism from a graph to a category (interpreted as a graph).

Definition 4.2

Given a category F, a *type model* for a given type graph $\mathcal{G}$ in F, is a model $M : \mathcal{G} \to F$. F is referred to as the *instance category* of the model.

At this point no requirements on the mapping of edges in relation to their labels is imposed. These requirements will be discussed in the remainder of this section. The above definition implies that the semantics of a data model depends on the instance category chosen. A choice for a category is a choice for desired semantic features, and as such the semantics has become configurable. This leads to an elegant way to introduce notions such as time (choose TimeSet), null values (choose PartSet), and uncertainty (choose FuzzySet) as extra dimensions. No redesign of the formalization is needed. Not all categories provide a meaningful semantics for data models. In [LH96] a number of requirements are imposed on categories to act as instance category and for a number of categories it is proven that they satisfy these requirements.

When modelling instances of relationship types, category theory also allows for a highly conceptual approach. Contrary to a tuple oriented approach, or a mapping oriented approach, category theory does not (have to) make any assumptions about an underlying way to represent instances. The categorical approach follows the observation that for relationship instances it is sufficient to have access to their various parts. As examples we will consider populations in three different instance categories.

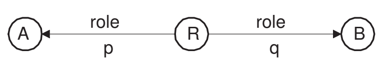

Figure 12: Type graph of figure 6

The type graph of the schema of figure 6 is shown in figure 12. Category theoretically, a population corresponds to a mapping from the type graph to an instance category. An example population

Pop in **FinSet** therefore, could be represented as:

$$\begin{aligned} \mathsf{Pop}(p) &= \{r_1 \mapsto a_1, r_2 \mapsto a_2\}, \\ \mathsf{Pop}(q) &= \{r_1 \mapsto b_1, r_2 \mapsto b_1\}. \end{aligned}$$

In this approach, the two relationship instances, r_1 and r_2, have an identity of their own, and the functions p and q can be applied to retrieve their respective components. Note that in this approach it is possible that two different relationship instances consist of exactly the same components, (this conforms to OO). This is not possible in the set-theoretic formalization.

Apart from **FinSet** it is also possible to choose other instance categories. As remarked before, the category **PartSet** allows certain components of relationship instances to be undefined, hence allowing a natural treatment of null values:

$$\begin{aligned} \mathsf{Pop}(p) &= \{r_2 \mapsto a_2\}, \\ \mathsf{Pop}(q) &= \{r_1 \mapsto b_1, r_2 \mapsto b_1\}. \end{aligned}$$

In this population, relationship instance r_1 does not have a corresponding object playing role p.

Another possible choice of instance category is the category **Rel**. In **Rel** the components of relationship instances correspond to sets, as roles are mapped on relations. A relationship instance may be related to one or more objects in one of its components (this is comparable to the notion of multi-valued attributes in OO). A sample population could be:

$$\begin{aligned} \mathsf{Pop}(p) &= \{r_2 \mapsto a_1, r_2 \mapsto a_2\}, \\ \mathsf{Pop}(q) &= \{r_1 \mapsto b_1, r_2 \mapsto b_1, r_2 \mapsto b_2\}. \end{aligned}$$

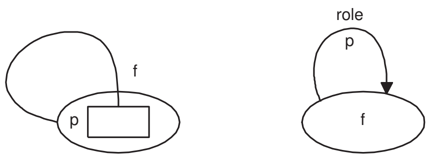

Figure 13: A cyclic unary fact type

As an example of another important difference between the set-theoretic formalization and the categorical formalization of relationship types consider figure 13. This fact type, represented both as an information structure diagram and a type graph, is populatable in the categorical formalization. A sample population in the category **FinSet** could be:

$$\mathsf{Pop}(p) = \{f_1 \mapsto f_1\}$$

This schema however is not populatable in the set-theoretic formalization. This can easily be seen as follows. Every instance in Ω can be assigned a natural number representing the shortest proof of its membership of Ω. This natural number can be referred to as the *depth* of the instance. If fact type f were to be populatable, then a population assigning a nonempty set of instances to f would contain at least one instance with a minimal depth, say t. Application of the conformity rule then gives that $t(p)$ should also be a member of the population of f, which gives a contradiction as the depth of $t(p)$ is less than the depth of t.

As mentioned before, *specialization* is used when specific facts are to be recorded for specific instances of an object type only. A specialized object type inherits the properties of its supertype(s), but may have additional properties.

As an example of specialization consider the ORM schema of figure 14. In this diagram the object type Teaching-Assistant is a subtype of both Student and Employee. In set-theoretic terms, the

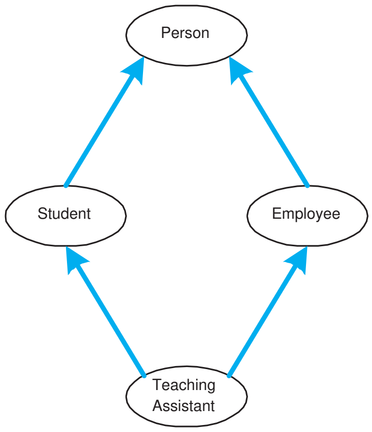

Figure 14: An example of a subtype hierarchy in ORM

most general formalization of a subtype relation would be to treat it as an injective function. This is more general than requiring that $\mathsf{Pop}(A) \subseteq \mathsf{Pop}(B)$ in the case that A is a subtype of B, as instances may have a different representation in both object types (this is particularly so in object-oriented data models). Therefore, category theoretically a subtype relation has to correspond with a monomorphism (in the category **Set** a monomorphism corresponds to an injective function). The definition of a monomorphism is:

Definition 4.3

An arrow $f : A \to B$ is a *monomorphism* if for any object X of the category and any arrows $x, y : X \to A$, if $f \circ x = f \circ y$, then $x = y$.

The requirement that specialization relations have to be mapped onto monomorphisms is not sufficient, however, for an adequate formalization of specialization relations. Consider for example the following population (in **FinSet**) of the schema of figure 14:

$$\begin{array}{rcl} \mathsf{Pop}(\mathsf{Person}) & = & \{\mathit{Jones}, \mathit{Richards}\}, \\ \mathsf{Pop}(\mathsf{Student}) & = & \{\mathsf{ST1943}\}, \\ \mathsf{Pop}(\mathsf{Employee}) & = & \{\mathsf{EM237}\}, \\ \mathsf{Pop}(\mathsf{Teaching\text{-}Assistant}) & = & \{\mathsf{TA999}\}. \end{array}$$

and the following subtype relations (see also figure 15):

$$\begin{array}{rcl} I_1 & = & \{\mathsf{TA999} \mapsto \mathsf{EM237}\}, \\ I_2 & = & \{\mathsf{TA999} \mapsto \mathsf{ST1943}\}, \\ I_3 & = & \{\mathsf{EM237} \mapsto \mathit{Jones}\}, \\ I_4 & = & \{\mathsf{ST1943} \mapsto \mathit{Richards}\}. \end{array}$$

In this sample population each specialization relation is mapped onto an injective function, but the instance TA999 of object type Teaching-Assistant corresponds to two instances of Person: *Richards* as well as *Jones*. Clearly, this is undesirable.

To avoid such problems, subtype diagrams, i.e. diagrams consisting solely of subtype edges, are required to commute. In terms of the presented subtype diagram this would imply that the function composition of I_2 with I_4 should be identical to the function composition of I_1 with I_3 and therefore: $I_4(I_2(\mathsf{TA999})) = I_3(I_1(\mathsf{TA999}))$.

Since the subtype diagram is required to commute, subtypes inherit properties from their supertypes in a unique way. In the example, every teaching assistant inherits the name from its supertype person.

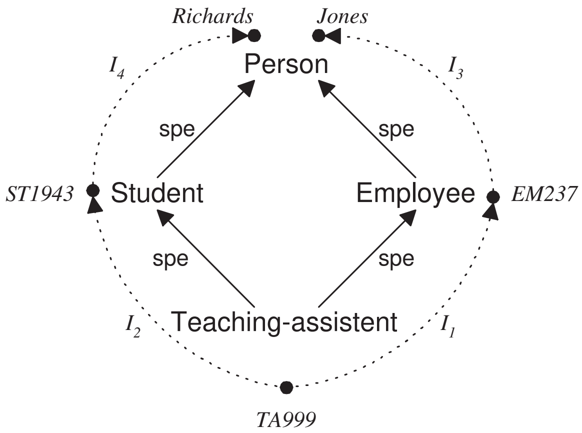

Figure 15: A non-commutative diagram

5 Conclusions

This paper presented a discussion on the *how* of formalization. Taking the need for elegant and concise formalizations as a starting point, it recognized the importance for guidance of the process involved. This guidance was provided in the form of a discussion of four different styles of assigning formal semantics and five formalization principles. Two elaborate case studies illustrated these different styles as well as these principles. Task structures, as an example of a dynamic modelling technique, were formalised in four different styles of assigning semantics, while ORM, as an example of a static modelling technique, was formalized using denotational semantics, with three different target domains.

Naturally, the ambition level of this paper has been modest and there is a definite need to further investigate formalization principles and to provide more in-depth guidelines and heuristics. This is a subject for further research.

References

[Bae90] J.C.M. Baeten, editor. *Applications of Process Algebra.* Cambridge University Press, Cambridge, United Kingdom, 1990.

[BH95] J.P. Bowen and M.G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, pages 34–41, July 1995.

[BHW91] P. van Bommel, A.H.M. ter Hofstede, and Th.P. van der Weide. Semantics and verification of object-role models. *Information Systems*, 16(5):471–495, October 1991.

[Blo93] A. Bloesch. *Signed Tableaux – a Basis for Automated Theorem Proving in Nonclassical Logics.* PhD thesis, University of Queensland, Brisbane, Australia, 1993.

[Bot89] P.W.G. Bots. *An Environment to Support Problem Solving.* PhD thesis, Delft University of Technology, Delft, The Netherlands, 1989.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, Cambridge, United Kingdom, 1990.

[Cod70] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Coh89] B. Cohen. Justification of formal methods for system specification. *Software Engineering Journal*, 4(1):26–35, January 1989.

[Dav90] A.M. Davis. *Software Requirements: Analysis & Specification.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[De 93] O.M.F. De Troyer. *On Data Schema Transformations.* PhD thesis, University of Tilburg (K.U.B.), Tilburg, The Netherlands, 1993.

[De 96] O. De Troyer. A formalization of the Binary Object-Role Model based on logic. *Data & Knowledge Engineering*, 19(1):1–37, May 1996.

[FHL97] P.J.M. Frederiks, A.H.M. ter Hofstede, and E. Lippe. A Unifying Framework for Conceptual Data Modelling Concepts. *Information and Software Technology*, 39(1):15–25, January 1997.

[Gen87] H. Genrich. Predicate/Transition Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986 Part I*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, Berlin, Germany, 1987.

[Gog91] J.A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.

[Gri82] J.J. van Griethuysen, editor. *Concepts and Terminology for the Conceptual Schema and the Information Base.* Publ. nr. ISO/TC97/SC5-N695, 1982.

[GW96] R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, May 1996.

[Hal89] T.A. Halpin. *A logical analysis of information systems: static aspects of the data-oriented perspective.* PhD thesis, University of Queensland, Brisbane, Australia, 1989.

[Hal90a] J.A. Hall. Seven Myths of Formal Methods. *IEEE Software*, pages 11–19, September 1990.

[Hal90b] T.A. Halpin. Conceptual schema optimization. *Australian Computer Science Communications*, 12(1):136–145, 1990.

[Hal91] T.A. Halpin. A Fact-Oriented Approach to Schema Transformation. In B. Thalheim, J. Demetrovics, and H.-D. Gerhardt, editors, *MFDBS 91*, volume 495 of *Lecture Notes in Computer Science*, pages 342–356, Rostock, Germany, 1991. Springer-Verlag.

[Hal95] T.A. Halpin. *Conceptual Schema and Relational Database Design.* Prentice-Hall, Sydney, Australia, 2nd edition, 1995.

[HE90] U. Hohenstein and G. Engels. Formal Semantics of an Entity-Relationship-Based Query Language. In *Proceedings of the Ninth International Conference on the Entity-Relationship Approach*, Lausanne, Switzerland, October 1990.

[HH96] J.W.G.M. Hubbers and A.H.M. ter Hofstede. An Algebraic Semantics for Object-Oriented Behaviour Modeling. In Paul A. Bailes, editor, *Proceedings 1996 Australian Software Engineering Conference*, pages 4–15, Melbourne, Australia, July 1996. IEEE Computer Society Press.

[HL91] X. He and J.A.N. Lee. A Methodology for Constructing Predicate Transition Net Specifications. *Software Practice & Experience*, 21(8):845–875, August 1991.

[HLF96] A.H.M. ter Hofstede, E. Lippe, and P.J.M. Frederiks. Conceptual Data Modeling from a Categorical Perspective. *The Computer Journal*, 39(3):215–231, August 1996.

[HLW97] A.H.M. ter Hofstede, E. Lippe, and Th.P. van der Weide. Applications of a Categorical Framework for Conceptual Data Modeling. *Acta Informatica (to appear)*, 1997.

[HN93] A.H.M. ter Hofstede and E.R. Nieuwland. Task structure semantics through process algebra. *Software Engineering Journal*, 8(1):14–20, January 1993.

[HO97] A.H.M. ter Hofstede and M.E. Orlowska. On the Complexity of Some Verification Problems in Process Control Specifications. Technical report 2/97, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, April 1997.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Hoa89] C.A.R. Hoare. Notes on an Approach to Category Theory for Computer Scientists. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 55 of *NATO Advanced Science Institute Series*, pages 245–305. Springer-Verlag, 1989.

[HOR96] A.H.M. ter Hofstede, M.E. Orlowska, and J. Rajapakse. Verification Problems in Conceptual Workflow Specifications. In B. Thalheim, editor, *Proceedings of the 15th International Conference on Conceptual Modeling (ER'96)*, volume 1157 of *Lecture Notes in Computer Science*, pages 73–88, Cottbus, Germany, October 1996. Springer-Verlag.

[HP95] T.A. Halpin and H.A. Proper. Subtyping and Polymorphism in Object-Role Modelling. *Data & Knowledge Engineering*, 15:251–281, 1995.

[HW92] A.H.M. ter Hofstede and Th.P. van der Weide. Formalisation of techniques: chopping down the methodology jungle. *Information and Software Technology*, 34(1):57–65, January 1992.

[HW93] A.H.M. ter Hofstede and Th.P. van der Weide. Expressiveness in conceptual data modelling. *Data & Knowledge Engineering*, 10(1):65–100, February 1993.

[Jen87] K. Jensen. Coloured Petri Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986 Part I*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299, Berlin, Germany, 1987. Springer-Verlag.

[Jen91] K. Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416, Berlin, Germany, 1991. Springer-Verlag.

[Jon86] C.B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[LH96] E. Lippe and A.H.M. ter Hofstede. A Category Theory Approach to Conceptual Data Modeling. *RAIRO Theoretical Informatics and Applications*, 30(1):31–79, 1996.

[Mai88] D. Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland, 1988.

[Mey90] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Pet62] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, Bonn, Germany, 1962. (In German).

[Pet81] J.L. Peterson. *Petri Net Theory and the Modelling of Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Pol57] G. Polya. *How to Solve It: A New Aspect of Mathematical Method*. Princeton University Press, Princeton, New Jersey, 1957. Second Edition.

[Rei85] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.

[Sch94] Th. Scheurer. *Foundations of Computing: Systems Development with Set Theory and Logic*. Addison-Wesley, Wokingham, England, 1994.

[SFMS89] C. Sernadas, J. Fiadeiro, R. Meersman, and A. Sernadas. Proof-theoretic Conceptual Modelling: the NIAM Case Study. In E.D. Falkenberg and P. Lindgreen, editors, *Information System Concepts: An In-depth Analysis*, pages 1–30, Amsterdam, The Netherlands, 1989. North-Holland/IFIP.

[Sie90] A. Siebes. *On Complex Objects*. PhD thesis, University of Twente, Enschede, The Netherlands, 1990.

[Spi88] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, Cambridge, United Kingdom, 1988.

[Tui94] C. Tuijn. *Data Modeling from a Categorical Perspective*. PhD thesis, University of Antwerp, Antwerp, Belgium, 1994.

[WH90] G.M. Wijers and H. Heijes. Automated Support of the Modelling Process: A view based on experiments with expert information engineers. In B. Steinholz, A. Sølvberg, and L. Bergman, editors, *Proceedings of the Second Nordic Conference CAiSE'90 on Advanced Information Systems Engineering*, volume 436 of *Lecture Notes in Computer Science*, pages 88–108, Stockholm, Sweden, 1990. Springer-Verlag.

[WHO92] G.M. Wijers, A.H.M. ter Hofstede, and N.E. van Oosterom. Representation of Information Modelling Knowledge. In V.-P. Tahvanainen and K. Lyytinen, editors, *Next Generation CASE Tools*, volume 3 of *Studies in Computer and Communication Systems*, pages 167–223. IOS Press, 1992.

[Wij91] G.M. Wijers. *Modelling Support in Information Systems Development*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 1991.

Contents