Concepts for Real-World Modelling

Andreas L. Opdahl* Guttorm Sindre†

Abstract

Simple, intuitive, and yet powerful languages are needed to model the real-world in the problem analysis phase of information system development. However, contemporary real-world modelling languages are either weak in expression or cluttered with rigorous detail. In the former case, models become to vague to be meaningful, while the latter case makes modelling even of rather simple dynamic systems a complex task and hardly facilitates communication with end users.

Object-orientation is not considered appropriate for this purpose, due to its low emphasis on dynamics. Dataflow diagrams, on the other hand, emphasise dynamics, but unfortunately, some major conceptual deficiencies make DFDs, as well as their various formal extensions, rather useless for real-world modelling.

This paper presents concepts for real-world modelling which rely on some seemingly small, but essential modifications of the DFD language. Hence the well-known, communication-oriented diagrammatic representations of DFDs can be retained. It is indicated how the approach can support a smooth transition into later stages of object-oriented design and implementation.

keywords: dataflow diagrams, real-world modelling, conceptual modelling, object-orientation, problem analysis

1 Introduction

There is an increasing tendency in information system development to consider not only the functionality which is going to be automated, but to give considerable attention to the organisation around the system [11]. This increased scope puts heavier requirements on the modelling languages, which have to be appropriate for modelling any real-world activity, rather than just the information processing activities to be automated. Real-world modelling can also be interesting in cases where no new information system is planned, e.g for general analysis of the organisation's information, document, and/or material resource flows with respect to performance, availability, reliability, or security.

*Andreas Opdahl is at the Dept. of Information Science, University of Bergen, Norway; email: andreas@ifi.uib.no

†Guttorm Sindre is at the Faculty of Electrical Engineering and Computer Science, University of Trondheim, Norway; email: guttorm@idt.unit.no

Concept	Process	Flow	Store
Activity	Transformation	Transportation	Preservation
Aspect	Matter	Location	Time

Table 1: A dataflow diagram taxonomy of real-world dynamics

Object-orientation has been promoted as a kind of panacea in software engineering, also in the analysis phase. However, dynamics are not sufficiently emphasised in object-oriented models, and this creates problems for dynamic analyses [2], as well as resulting in vague, imprecise representations. Section 2 will elaborate on this.

There are lots of languages around for modelling dynamics. Many are variants either of Petri-nets [28] or state-transition diagrams [19]. However, "place" and "transition" based Petri-nets do not include concepts which are closely linked with what goes on in the real-world. Hence they are unintuitive and difficult to understand for untrained personnel, and the resulting models do not support easy communication. State-transition based models go one step even further, disregarding the entities from the which the real-world is composed altogether. An often-used alternative is dataflow diagrams (DFDs) [13, 15], which have been praised for their intuitive appeal [25]. Knowingly or unknowingly, the DFD language embodies a beautiful taxonomy of real-world dynamics in its three major concepts for dynamic modelling. This taxonomy is indicated in table 1, where the three concepts correspond to a straight-forward perception of human activity, and furthermore to three orthogonal aspects of nature: matter, location, and time. This will be explained in sec. 4. However, DFDs have several weaknesses which prevent them from being appropriate for real-world modelling. Most emphasised is the lack of a formal basis [31], especially with respect to representation of *dynamics*. E.g. it is unclear *when* a dataflow process executes, as well as what it receives and sends on its input and output flows per execution [4]. Such issues have been addressed by later dataflow extensions [36, 16, 14]. The lack of support for declarative business rules in a DFD setting is another point that has been addressed [20]. However, none of these improvements have fully addressed the implications of the above taxonomy and as a result, DFD, in spite of all extensions, remains unsuitable for real-world modelling.

The goal of this paper is therefore to arrive at a language which maintains and consistently exploits the above taxonomy, and which provided the appropriate abstraction mechanisms for modelling large systems.

The rest of the paper is structured as follows: Section 2 describes the major shortcomings of object-orientation as a paradigm for real-world modelling, and section 3 similarly describes the major shortcomings of dataflow diagrams. Section 4 presents our extensions to DFD, both in concepts and notation, addressing the previously mentioned weaknesses. Section 5 outlines the relation between the proposed modelling framework and object-oriented modelling before section 6 presents some concluding remarks and paths for further work.

2 Criticism of Object-Oriented Analysis

Object-orientation is becoming increasingly popular. Object-oriented programming (OOP) and design (OOD) are already well established. As a consequence, numerous object-oriented analysis (OOA) methods are being proposed, an overview of which is found in [5]. Some of the most often stated advantages of OOA are:

- object-orientation corresponds very well to the human perception of the real world, i.e. it is "natural"
- choosing an object-oriented structure also for analysis, one achieves a smooth transition to design and implementation

In the following we will address these two issues in separate.

2.1 OOA is not "Natural"

True enough, object-oriented specification languages contain many of the same constructs as semantic data models [27, 29], which are believed to support the way humans think. However, whereas data models only aspire to giving a static portrait of the world, object-oriented analysis is supposed to deal also with dynamics, and must thus be evaluated in a different light. In the dominant approaches to OOA/OOD [10, 33, 38, 34, 7, 37] the focus is on classes: identifying the classes and their relationships and organising class hierarchies. The methods first capture statics, then dynamics. By being subordinate to the static structure, dynamics are generally given less emphasis than statics, and diagrams typically depict only static structure — to find out what goes on in an object-oriented specification one will often have to think through some complicated spaghetti-style message-passing interaction. This creates problems in the analysis phase where *gaining understanding* of the problem at hand is a main issue. Furthermore, intuitive validation of the established models is difficult, and attempts at formal verification would create great solution complexity.

Thus, a claim that object-orientation is a natural way to view the world, implies that statics must somehow be more basic or fundamental than dynamics in the human perception of the real world. However, there are many indications to the contrary:

- Particularly in Eastern philosophy [30, 32] but also in Western [3], it has been stated that everything is change, and that processes are more fundamental than things. These attitudes have also penetrated to some researchers in modern physics [6, 9].
- From a more pragmatical point of view, human organisations (which information systems are, after all, made for) can also be described as ever changing systems, more like processes than things [24].

- The *verb* is the most sophisticated word class of almost all natural languages, with alternative forms on a wide variety of axes: tense, mood, aspect, number, person [23]. The forms of the noun are fewer, and in a modern language such as English, case and gender have become extinct [21]. Moreover, the verb is usually the most prominent word in a sentence, to which all the other sentence elements stand in some relation — cf. the tendency to start with identifying the main verb during syntactic analysis of natural language [23]. It can also be noted that language itself is a process of continuous change [12].

From this, we conclude that a paradigm which emphasises nouns/objects at the cost of verbs/processes, can hardly be called natural if the ambition is to model the real world.

2.2 OOA does not Avoid "Gaps"

The claim that OOA results in small gaps between the phases is often true. However, the small gap is achieved at the cost of a very shallow analysis. The demand that requirements should be formulated declaratively has been increasing in strength since the early eighties [17]. OOA claims to be declarative compared to OOD, i.e. describing *what* the system should provide (in terms of classes and operations), whereas the design tells how this is done. However, to some extent this declarativity is *fake*, i.e. it is achieved by the fact that OOA is merely less detailed than OOD, a sketch of the design.

Compared to using DFDs, object-oriented analysis might be a step forward. However, compared to other recent experimental approaches it is semantically restricted. E.g. it is very difficult to specify declarative business rules within an object-oriented structure. Declarative rules are inherently global in nature, whereas object-orientation means that one can only state rules in a fragmented way within class definitions.

Hence a small gap can be achieved only by pulling analysis closer to design than it should have been. As stated in [18] requirements engineering should be problem-oriented, not target-oriented. Object-orientation is a design decision, not a requirement, and choosing object-oriented analysis just because the next step is object-oriented design is simply not a sufficient argument.

3 Criticism of DFD for Real-World Modelling

To establish DFD-like concepts for real-world modelling, the more fundamental weaknesses of dataflow diagrams in this respect muse be surveyed. In the following we will make some more observations on the weaknesses of DFD for real-world modelling. In particular, we address two issues which emphasise the problems with DFD as a language for modelling the real world:

- the weakness of the flow concept, and

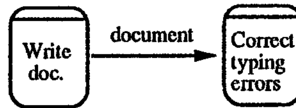

Figure 1: What does a flow mean?

- the dominance of the processes.

Related deficiencies have been touched upon in [8]. In particular, it is pointed out that decomposition is very problematic, particularly with respect to flows. Either, a tremendous amount of flows occur at the higher levels of abstraction, or alternatively, flows obtain very obscure meanings.

3.1 Weaknesses of the Flow Concept

One of the main problems of the flow in DFD is that it is overloaded with alternative meanings. Some of these problems have been addressed, for example distinguishing between *control flow* and *dataflow* [36] or *information flow* and *material flow* [22]. However, these are all flows, and thus, the main problem has not been addressed: namely that the flow is the only possible link between processes, stores and external entities.

Consider the simple diagram of fig. 1, with the first process "Write document" producing a (electronic or paper) document which is the input to the second process "Correct typing errors". What does the flow in this picture mean? In fact, it can have two rather different meanings: either it can mean that the document is in fact transported (or transferred electronically) from one place to another, e.g. because the two tasks are performed by two different persons. But it might also mean simply that the output of the first process is the input of the second, without any input taking place (e.g. if the two tasks are performed by the same person). Intuitively, one would think from the arrow symbol that there is some transportation, but inspecting a number of DFDs, one is likely to find lots of examples where flows are simply used to make an output/input connection between two processes. This ambiguity of the flow concept is a major problem of DFD, and looking at a slightly more complicated example, we will in fact see that DFD completely fails to model a dynamic system in a proper way.

For this example, consideran extension of the previous one, where there are two processes working in parallel correcting the document after it has been written — one dealing with typing errors and one with deeper errors (syntactic, semantic, pragmatic). Of course, one could make one single process "Proofread document" out of **B** and C, but then we would totally miss the fact that there are two different tasks, which might be performed by different persons, or the first even by a computer. Clearly, we must be able to model this as two processes, and if they are performed

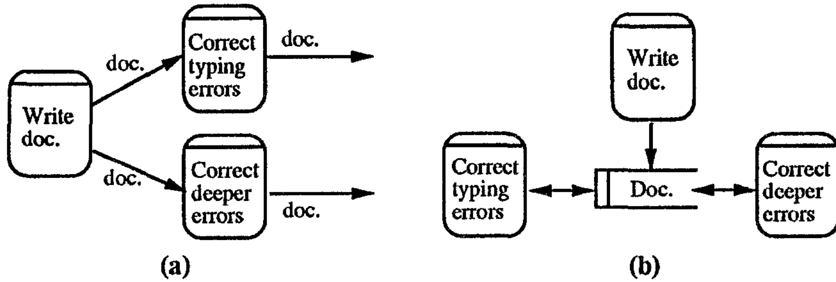

Figure 2: Problems with parallel material flows

in parallel in real life, they must also be modelled as such.

Since both processes are supposed to change the document in some way, it must be an input and output of both these processes. However, the diagram of fig. 2 does not make much sense — the first process seems to be sending out two documents rather than only one, and the other two processes turn out one corrected document each — one corrected only for typos, and one corrected only for deeper errors. It would of course be possible to achieve the wanted effect (both processes working on the same document) by defining some underlying execution semantics for the diagram, unique labelling conventions etc., but this does not seem very satisfactory, since the strong side of DFDs in the first place is their intuitive appeal [14], the basic dynamics of the system should be apparent from the diagram itself rather than relying on some odd, underlying conventions.

The attempt in fig. 2b, introducing a store to hold the document, does not work either — it would seem contradictory that the two correcting processes should be able to process the same document at the same time, since intuitively an arrow from a store means taking something out of it, and an arrow to a store means putting something into it. One of the major problems is that whereas information can be duplicated for processing purposes and still be the same information, material cannot — there is no "non-destructive get" (i.e. a "read") for a material substance. Thus, whether on file or paper, if the physical document is processed in parallel by several actors, the situation becomes very difficult to model with a DFD. And parallel processing being very common, a language which cannot deal with it properly is of course useless for our purposes.

3.2 The Dominance of Processes

Another observation is that the driving force in dataflow diagrams, as well as in more modern and formalised extensions [36, 16, 14] is the process — it is processes that make data (or material) flow, and it is processes that take data or material in and out of storage — flows and stores are only passive slaves. The process is equipped

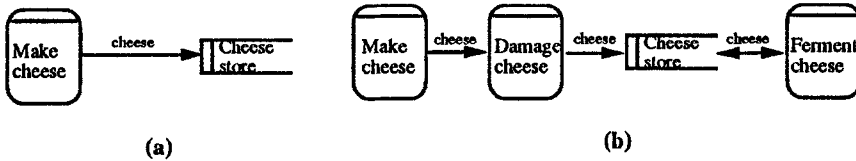

Figure 3: Data or materials being changed inside flows and stores.

with much more complex semantics than the other two, and all decisions are made within processes. Accordingly, only processes can be decomposed to contain full DFDs inside (i.e. both new processes, flows, and stores). In some DFD formalisms, store and flow decompositions are allowed, but stores can only be decomposed into substores, and flows only into subflows. This is a major reason for some of the problems reported in [8] that high-level flows have very little meaning, just grouping together various things which happen to flow between the same high-level processes.

Although seemingly "natural", this dominance of processes reduces the expressive strength of diagrams. Also, it is not consistent with the ways in which real-world systems actually work: Just like there is flow and storage within a process, there should be an ability to represent processing within flows and stores. In the real world artifacts might change during transportation or storage, and such changes are sometimes important when designing models. To accomplish this, we must either:

- allow full decomposability also of flows and stores, or
- model any change by means of a process *outside* the flow or store.

The latter alternative is not very satisfactory, because it creates an abundance of processes, resulting in inferior abstraction, as illustrated in fig. 3. The case here is transportation and storage of cheese. Of course, cheese which was OK when loaded onto the truck/boat/train/plane might be destroyed during transport, and the quality may also be reduced (or improved) during storage. Compared to the simple picture of 3a, fig. 3b has to make two extra processes: "Damage cheese" for the possible change during transportation and "Ferment cheese" for the possible change during storage. Especially since the processes of damage and fermentation are merely results of transportation and storage, rather than something which is done actively by some actor in the organisation, it would seem more natural to hide these details within the flow and store, rather than exposing them as independent processes at the same level as cheese production.

With the former approach, one would yield just the picture of fig. 3a — details about the flow and the store could be hidden within their respective nodes, only to be shown when inspecting their decompositions. It is very difficult to abstract away details such as this without hiding them in the flow or store — the processes of damage and fermentation happen after the "Make cheese" process is finished

and cannot easily be hidden inside this process. It seems particularly unnatural to picture "Ferment cheese" outside the store, virtually taking cheese out of the store and putting it back after some fermentation, when actually, the cheese remains in the store all the time. The modelling of "Damage cheese" is also very unintuitive — from the picture it seems that the cheese is first transported to a particular site where it is damaged, then to be brought on to the store. Hopefully, the damage only applies to a very small ratio of the transported cheese, but the diagram also fails to indicate this — apparently, damage is the rule rather than the exception.

Summing up, our main requirements for a real-world modelling language based on DFD are as follows:

- Processes, flows, and stores should all have well-defined meanings which correspond to the human perception of transformation, transportation and preservation. Specifically, flows should not be used both for transportation and simple output/input connections, but rather be reserved for the former.
- The three concepts should be fully inter-decomposable, to make it easier to abstract away details at the appropriate place in a model.

These requirements will be dealt with in the following.

4 Real-World Modelling Concepts

According to the discussions in the previous section, we will now define a formalism which is more appropriate for real world modelling than DFD, whereas maintaining its good features. As stated earlier, DFD contains a nice taxonomy of real-world dynamics in the three concepts process, flow, and store — denoting transformation, transportation, and preservation. However, as we have seen, DFD itself is not true to this taxonomy, for instance using the flow concept also in cases where there is actually no transportation taking place. In the following we will describe the essentials of such a modelling language.

4.1 Items, Pieces, and Streams

To represent the dynamics of a real-world system (i.e. our *universe of discourse*), the static entities changed by that dynamic system most first be described. According to Wand [35], the real-world can be perceived as consisting of "things", which will be called *items* in this paper. Following Wand, things have properties, which will be called *attributes* by us. Hence an item $\iota = \{a_i\}$ can be formally defined as a set of attributes a_i, $v(a_i, t) \in D(a_i)$, where $v(a_i, t)$ is the *value* of property a_i at time t and $D(a_i)$ is its *domain* (or "type"). Of course, items can be described in terms of *item classes*, but this is not an issue in this paper.

Although items represent matter, they may at the same time represent the data

associated with matter. In general, three aspects of an item can be perceived as relevant when creating a model [26]:

- The *substance* of an item corresponds to material presence, e.g. the presence of a paper form in a worker's input basket which triggers a certain activity related to that form.
- The *properties* of an item are the data that can be extracted from the item's substance, e.g. the colour of the paper form, its size and thickness.
- the *data* carried by an item is the information purposely coded onto the substance, e.g. the letters written on the above form.

Both properties and data are represented as attributes of items, while substance is represented by the item's presence.

Items can be either *discrete* or *continuous*. The attributes of a discrete or continuous item defines its state. Hence $S(\iota, t) = \{v(a_i, t)\}$ is the state of item ι at time t. Changes of attributes change the states of items.

The states of discrete items change instantaneously, corresponding to *events*. An event e occurring at time t can therefore be formally defined as a 4-tuple $e = (\iota, S, S', t)$, where ι is the item whose state has changed, and S and S' are its old and new states respectively. The states of continuous items change continuously, corresponding to *alterations*. The effect of an alteration during interval dt can therefore be formally defined as a 4-tuple $e = (\iota, S, S', dt)$, where ι, S, and S' are as in the above.

Furthermore, items are located in space. Items with a single well-defined location at the level of which the real-world system is perceived are called *pieces*. Items without such a well-defined single location are *streams*. Hence streams can simultaneously occupy several locations, but not all of them are of interest in a real-world model. E.g. when representing the flow of water through a drainpipe, only its end-point are of interest. Pieces and streams may both either have discrete or continuous events, hence they are called *discrete* or *continuous pieces* or *streams* respectively. In the above example, the form is probably most appropriately modelled as a discrete piece.

Let $p(\iota, t)$ be the *location* of piece ι at time t, and let $\mathcal{P}(\iota, t) = \{p(x, t) | x \in \iota\}$ be the *extension* of stream ι. Hence an extension is defined as the set of locations of all its parts, these part locations will be called *location points*. The locations and location points of all items in the universe of discourse can be described as a set Π. Elements $\pi \in \Pi$ of this set will be called *ports* for reasons that will soon become apparent.

4.2 Ideal Processes, Flows, and Stores

The processes, flows, and stores of conventional dataflow diagrams are now interpreted as transformations, transportations, and preservations of items respectively. These divide the representation of real-world dynamics in three orthogonal aspects: matter, location, and time:

- An *ideal process*, p, represents a change of matter (and hence potentially also of the associated data) while keeping location and time constant (i.e. performing the modification in zero time and without any changes of location).

  Formally, an ideal process $p = (\Pi_I, \Pi_O, \Phi)$ consumes items, ι, through a set of *input ports*, $\pi_i \in \Pi_I$, and produces items through a set of *output ports*, $\pi_o \in \Pi_O$. The *attributes* of items produced are determined by a set of functions $\Phi = \{\phi_o\}$. Each function ϕ_o in this set is a function of attributes of input items, determining the attributes of the items output through each output port. All input and output ports $\pi \in \Pi_I \cup \Pi_O$ must represent the same spatial location.

- An *ideal flow*, f, changes the location of items, while keeping matter and time constant (i.e. performing the transportation in zero time and without any changes of matter).

  Formally, an ideal flow $f = (\pi_i, \pi_o)$ consumes an item from input port π_i and immediately produces an item with exactly the same attributes to output port π_o. The output ports must represent locations which are different from the input ports. (The concept can of course easily be extended to account for multiple input and output ports.)

- An *ideal store*, s, changes time, while keeping the matter of items and their locations constant.

  Formally, an ideal store $s = (\Pi_I, \Pi_O)$ consumes items from input ports and produces items with the same attributes to its output ports after some time. The input and output ports $\pi \in \Pi_I \cup \Pi_O$ must represent the same spatial location.

Hence ports are the spatial locations from which items are consumed and to which items are consumed by processes, flows, and stores. These must correspond to the locations of pieces which are perceived as relevant when modelling a real-world system, as well as to the location points of streams. An *ideal model* is a composition of ideal processes, flows, and stores, as will be defined in sec. 4.5.

4.3 Real Processes, Flows, and Stores

Obviously, such ideal concepts are not realistic. However, they might be useful in many contexts — formalisms such as Petri-nets [28] and state-transition diagrams [19] have provided powerful mechanisms for dynamic modelling with zero-time transitions only. In the framework, the ideal concepts are the basic building blocks through which the real-world is perceived. However, at higher levels of abstraction — when perceiving more complex systems — it does not seem appropriate, nor necessary, to restrict expressiveness that much. Therefore we introduce, correspondingly, what we call the real variants of process, flow, and store:

- a *real process*, P, is an activity which might change both matter and location of items, as well as time. However, being modelled as a process, the activity

is ***mainly considered*** a transformation activity, the change of matter being perceived as more important than the change of location or time.

- a *real flow*, F, is an activity which might change both matter and location of items, as well as time. However, being modelled as a flow, the activity is mainly considered a transportation activity, the change of location being perceived as more important than the change of matter or time.

- a *real store*, S, is an activity which might change both matter and location of items, as well as time. However, being modelled as a store, the activity is mainly considered a preservation activity, the change of time being perceived as more important than the change of matter or location.

An *real model* is a composition of real ***or ideal*** processes, flows, and stores, as will be defined in subsection 4.5.

Formally, real processes, flows, and stores are essentially the same ***dynamic entity***, $E = (\Pi_I, \Pi_O, \Phi)$, consuming items from input ports $\pi_i \in \Pi_I$ and producing items to output ports $\pi_o \in \Pi_O$. The ***attributes*** of items produced are still determined by a set of functions $\Phi = \{\phi_o\}$ — one for each output port — of attributes of items consumed. Ports $\pi \in \Pi_I \cup \Pi_O$ may or may not have the same location, and output items may or may not be produced immediately upon input item consumption.

Hence the distinction between the three real concepts becomes entirely conceptual, allowing exactly what was wanted: full inter-decomposability among processes, flows, and stores — a process can contain stores and flows in addition to subprocesses, a flow can contain processes and stores in addition to subflows, and a store can contain processes and flows in addition to substores.

Hence real concepts are ***inter-decomposable***. From the definitions of the previous subsection 4.2 however, it can be seen that the corresponding ideal concepts presented there are ***not***: A flow containing a substore would not produce output items immediately upon input item consumption, and a flow or store containing a subprocess would not necessarily produce output items with the same attribute values as its input items.

Nevertheless, the possibility exists for decomposing an ideal process into subprocesses only, an ideal flow into subflows only, or an ideal store into substores only. Hence ideal concepts are ***intra-decomposable*** which is a restricted form of decomposability.

4.4 Links

The previous sections 4.2 and 4.3 explained and defined ideal and real concepts for transformation, transportation, and preservation. However, to represent a real-world *system* with them, their *interactions* must also be defined. Obviously, a concept is needed to relate the output ports and input ports of different processes, flows, and stores to one another. We call this concept a ***link***:

- a *link* l binds the input and output ports of several dynamic entities. The meaning of a link is that the bound ports are actually *identical* in space-time. As a consequence, output items that are produced through the output ports bound by the link are immediately consumed from the input ports. Hence the link has a direction from its output ports to its input port. There is no transformation, transportation, or preservation involved across the link. It is no change, zero time, zero distance. Of course, a link comprising multiple input ports initiate parallel dynamic threads in the model, while links with multiple output ports synchronise those threads.

  Formally, a link $l = (\Pi_o, \Pi_i)$ is a pair of sets of output $\pi_o \in \Pi_o$ and input $\pi_i \in \Pi_i$ ports.

Conceptually, processes, flows, and stores cannot be directly connected to each other — all such connections have to be made through links.

The previous section 3.1 stated that a major problem with flows was that sometimes they mean transportation, other times only an output/input connection. We now realise that this problem has disappeared through the introduction of links: The concept of flows has been reserved for the former phenomenon, and links account for the latter one. Links represent e.g. that the output of one process is the input of another one (or of *several* other ones) without any actual transportation taking place.

4.5 Ideal and Real Models

Summing up our conceptual basis, we have laid down three basic principles for reinterpreting dataflow diagrams:

1. orthogonality of ideal concepts;
2. full inter-decomposability of real concepts;
3. distinguishing between flow (ideal or real transportation) and link (zero-time, zero-distance communication);

We have identified six concepts: *process*, *flow*, *store*, *item*, *link*, and *port*. While processes, flows, and stores were either *ideal* or *real*, no such distinction was made for items, links and ports. However, items were classified as either *discrete* or *continuous* with regard to state changes, and as either *pieces* or *streams* with regard to spatial location.

Formally, an *ideal model*, $D_I = (\mathcal{P}, \mathcal{F}, \mathcal{S}, L)$, is a 4-tuple of sets of ideal processes, flows, and stores, as well as a set of links. A *real model*, D_R, accordingly is a 4-tuple of sets of real or ideal processes, flows, and stores, as well as a set of links. Hence ideal models can contain only ideal concepts, while real models may contain both real and ideal ones.

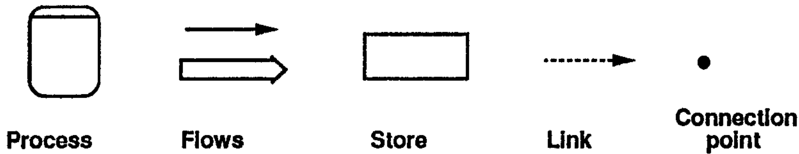

Figure 4: Graphical symbols for the real-world modelling concepts.

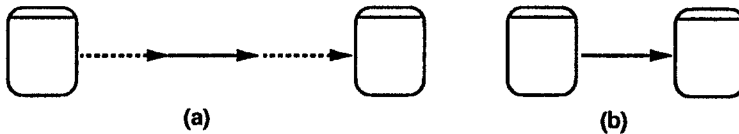

Figure 5: A notational convention to reduce the number of arrows

The only structural consistency requirement imposed on a model, is that all output and input ports π_o and π_i bound by links $l = (\Pi_o, \Pi_i) \in L$ must in fact be output or input ports of some dynamic entity. However, the converse does not have to be true, i.e. there may be input and output ports of dynamic entities in the model which are *not* bound by links. Let the sets $\Pi_{D,I}$ and $\Pi_{D,O}$ of *free* (i.e. not bound) input and output ports be the *input* and *output ports of model D*, respectively. In this way, every model can be represented as a dynamic entity at the next higher level of abstraction. Whether to use a process, flow, or store representation at that level is again a matter of what the modeller sees as most important.

4.6 Graphical Conventions

We retain the traditional DFD notation [13, 15] for processes and flows — however, to facilitate the visualisation of decomposition, it must also be possible to depict the flow as an enlarged kind of box-arrow. Similarly, to facilitate the illustration of decomposed stores, full rectangles are more convenient than the open-ended ones used in the conventional notations. Links are shown as dotted arrows. Connection points are normally not shown in the diagram (since they will generally be implicit from the contact between a link and a flow/process/store. If necessary, though, the connection points can be shown as tiny, black dots. The symbols are shown in fig. 4.

Introducing links for all connections between flows, processes, and stores, there would easily be an abundance of arrows in the diagrams. To avoid this, we introduce the notational convention that links are not necessary when at least one of the connections is to one of the end-points of a flow. Thus, the conceptual situation of fig. 5a, can be portrayed diagrammatically as in fig. 5b.

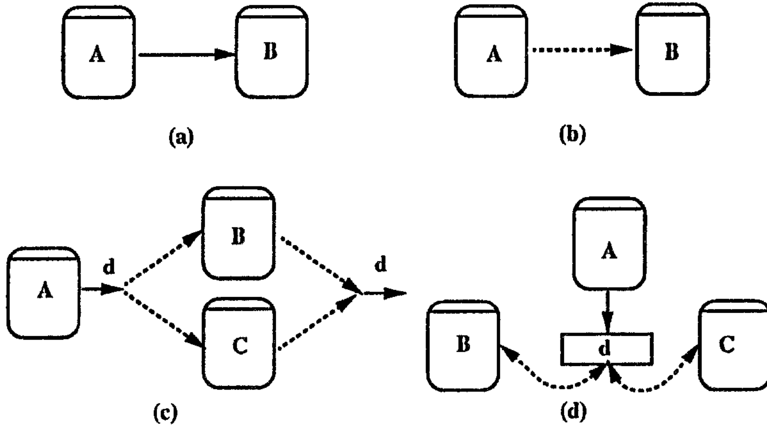

Figure 6: Modelling with links and flows

4.7 Examples

The usefulness of the link concept is indicated in fig. 6, where flows are modelled with full lines and links with dotted lines. Then, the two diagrams of figs. 6a and 6b have different meanings: in 6a the document is actually transported between the two processes, whereas 6b simply says that it is the output of the first process and the input of the next, without any transportation taking place. The real usefulness of the link can be appreciated reconsidering the example with parallel processing of fig. 2. With the introduction of links, both the previous attempts at solution suddenly work. In 2a, instead of directing the flow from "Write document" to any of the two processes, it goes to a free-standing connection port which is again connected to the two processes by means of two links. This tells us that the two inputs of the processes are actually happening at the same point in space-time, i.e. they both get the same input item. Similarly, the output is linked so that this must also necessarily be the same item. This diagram still indicates some transportation between the first process and the other two. If there were no transportation, we could have connected the two links to a point at the surface of "Write document". In 2b, we have simply replaced the flows between the two correction processes and the store with links, both connecting to the same port at the surface of the store — again indicating that the item worked on is not transported to/from the store but resides there all the time, and that both processes work on the same item at the same time.

Examples on the use of these options are indicated by the diagrams in fig. 7: 7a shows the transportation of a document (either electronically or on paper) which has to be coded and decoded. Clearly, it might be convenient to view the coding and decoding as details within the transportation activity, since these activities are logically more a necessity of the communication than of the mere production and reading of the document. This kind of decomposition is shown in fig. 7b. With the traditional

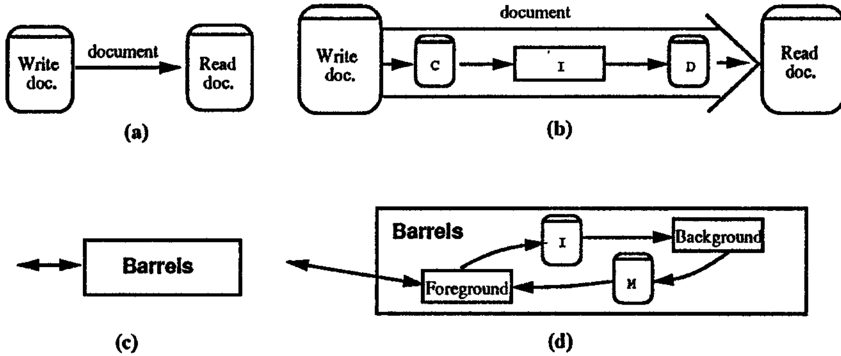

Figure 7: Full inter-decomposability

process dominance, the coding C and decoding D would rather have to be hidden within the "Write" and "Read" processes, respectively, and the intermediate storage I might have been pulled up to a higher level of abstraction. The modeller still has the choice to do this in our formalism, but now there is a wider variety of choices concerning how to do abstractions — one can always choose the way which feels most natural.

The next example, fig. 7c, shows a store which contains barrels. Imagine now that within this store there are actually two stores: one which contains the tip-top barrels ready for use, and another one where damaged barrels are stored, awaiting maintenance. If a barrel in the foreground store is found to be inferior, it is sent to the background store for maintenance, and when a barrel has been fixed, it is sent back to the foreground store. Clearly, this interaction between the two parts of the barrel store should be hidden at high levels of abstraction. With traditional process dominance, such abstraction could only be done using a process at the higher level, since the lower level also contains processes (for instance barrel inspection and maintenance). However, the barrel store is first and foremost a store, even if there is transformation and transportation taking place within it, and thus, it would be most intuitive to model it as such. This is achieved with our approach, the result shown in 7d — the process I performing barrel inspection, and the process M performing barrel maintenance (possibly, these two processes could be further hidden within the Foreground and Background stores, respectively).

A typical real-world example, illustrating the benefit of increasing the power of flows and stores, is that of fig. 8. Here, we have got a system of three lakes, with a river connection between them. The river connection has several side-branches, all running top to bottom in the map in fig. 8a. In 8b and 8c we show two models of this system. The first one is on a high level of abstraction and has considered the smallest lake, and all but the two main branches of the river to be too insignificant for presentation. The second one shows all the detail. It should not be difficult to

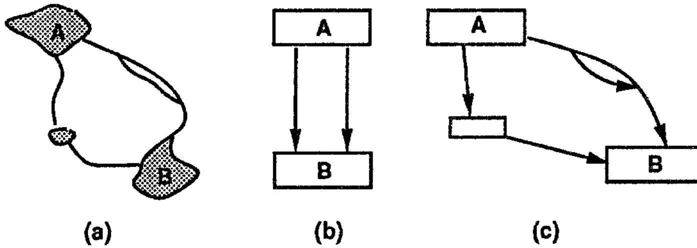

Figure 8: A lake-river system

imagine how complicated the model of this system would seem with the traditional DFD, where one would need processes to get the water in and out of the stores, and where a flow connected to another flow, without any intermediate process or store, would be illegal (and meaningless).

5 Relation to Object-Oriented Design

As pointed out in sec. 4.3, the difference between real processes, flows, and stores is purely conceptual — they can all be considered dynamic entities, with emphasis on transformation, transportation, and preservation, respectively. Taken to the extreme, one could introduce a conceptual "store" for every type of item in the domain modelled, thus portraying everything which would be static in a traditional data model as a process. These stores would then have links to every dynamic entity in the model where the item could occur. An interesting observation is that structurally, such a specification comes very close to object-orientation — especially the OMG object model, level 2 [1], where free-standing operation-objects are allowed, i.e. operations not encapsulated in any objects. At the lowest level of decomposition, our approach and the level 2 approach of OMG might end up with exactly the same dynamic entities in a model, although the models at higher levels could have been completely different, due to an emphasis of objects by OMG and processes by us.

From this it can also be concluded that choosing a highly process-oriented (or for that sake object-oriented) modelling paradigm during analysis, does not restrict us to choosing a similar structuring principle for design. When things are broken down far enough, one can choose whether to arrange processes around objects (OO) or the other way around (PO). This corresponds to the principle of top-down analysis and bottom-up design promoted by Bubenko in [8]. Notice that our criticism of object-orientation in section 2 considered only its restrictions upon the analysis phase, with particular attention to systems where dynamics are important. We do recognise many advantages of object-orientation when it comes to the design and implementation of such systems.

6 Conclusions and Further Work

The need for simple, intuitive, and powerful real-world modelling languages was pointed out. However, both object-oriented and dataflow diagram modelling approaches were found insufficient. Object-oriented analysis was weakened by its focus on static — as opposed to dynamic — aspects of the real-world system. In addition, analysis was to dependent on the later design phase. Dataflow diagrams were problematic with respect to the unclear "flow" concept, in addition to being to focused on "processes."

This discussion lead to the formulation of requirements on a real-world modelling framework, for which concepts were then proposed. A reinterpretation of the three basic dataflow diagram constructs made them consistent with basic human activities transformation, transportation, and preservation, in addition to three orthogonal aspects of nature: matter, location, and time. Interactions between the three types of constructs in a real-world model were expressed through the concepts of links, immediately solving the previous problem of DFD "flow" semantics. A distinction was made between ideal and real processes, flows, and stores, allowing full inter-decomposability between the three. It was shown how the conventional diagrammatic dataflow representation could still be used with the new concepts.

Although having evolved over several years of work on the PPP modelling approaches [16], this work is still in its initial phases. The proposed framework must be further refined and validated through more comprehensive examples. The formal framework must be extended correspondingly and tool-support be provided. More work is also needed to fully understand its relation to object-oriented design and implementation.

References

[1] T. Atwood et al. The OMG Object Model. Technical report, September 1991. draft 0.9.

[2] Sidney C. Bailin. An object-oriented requirements specification method. *Communications of the ACM*, 32(5), May 1989.

[3] Heraclitus (500 B.C.). Homeric questions. In *Early Greek Philosophy*. Penguin Books, London, 1987.

[4] S. Berdal, S. Carlsen, A. Sølvberg, and R. Andersen. Information system behaviour expressed through process port analysis. Technical report, Division of Computer Science, The Norwegian Institute of Technology, 1986.

[5] Arne-Jørgen Berre. Object-oriented analysis and design — an overview of some existing methods and techniques. Technical report, Center for Industrial Research, Oslo, Norway, 1990.

[6] D. Bohm. *Wholeness and the Implicate Order*. Ark, London, 1980.

[7] Grady Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[8] Janis A. Bubenko jr. Problems and Unclear Issues with Hierarchical Business Acitivity and Data Flow Modelling. Technical report, SYSLAB, 1988. Working paper no. 134.

[9] F. Capra. *The Tao of Physics.* London, 2nd edition, 1983.

[10] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Prentice Hall, Englewood Cliffs, first edition, 1990.

[11] D. W. Conrath, V. De Antonellis, and C. Simone. A comprehensive approach to modeling office organization and support technology. In *Proc. IFIP WG 8.4 WC on office information systems: the design process, Linz,* August 1988.

[12] D. Crystal. *The Cambridge Encyclopedia of Language.* Cambridge University Press, 1987.

[13] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Inc., New York, 1978.

[14] Robert France. Semantically extended data flow diagrams: A formal specification tool. *IEEE Transactions on Software Engineering,* 18(4):329–346, April 1992.

[15] C. Gane and T. Sarson. *Structured Systems Analysis: tools and techniques.* Prentice-Hall International, 1979.

[16] Jon Atle Gulla, Odd Ivar Lindland, and Geir Willumsen. PPP — An integrated CASE environment. *Proceedings of "CAiSE91, Trondheim, Norway",* May 1991.

[17] M. R. Gustafsson et al. A declarative approach to conceptual information modelling. In T. W. Olle et al., editor, *Information Systems Design Methodologies: A Comparative Review.* North-Holland, 1982.

[18] Jacques Hagelstein. Problem-oriented requirements engineering. In G. Shoenmakers, editor, *Colloquium Software Specificatie Technieken.* Academic Services, Schoonhoven, 1987.

[19] J. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[20] J. Krogstie et al. Information systems development using a combination of process and rule based approaches. In *Proc. CAiSE'91, Trondheim.* Springer Verlag, June 1991.

[21] W. P. Lehmann. *Historical Linguistics.* Holt, Rinehart & Winston, New York, 2nd edition, 1973.

[22] O. I. Lindland et al. PPP — the Process & Phenomenon Model. In *Proc. DnD/Infotech'88,* March 1988.

[23] John Lyons. *Introduction to Theoretical Linguistics.* Cambridge University Press, 1968.

[24] Gareth Morgan. *Images of Organization.* Sage Publications, Inc., 1986.

[25] P. Naur. Intuition in software development. In *Formal Methods and Software Development.* Springer Verlag (LNCS 186), 1985.

[26] Andreas L. Opdahl. A formal definition of diagrammatic systems specifications. Technical report, Diploma Thesis, Division of Computer Science, The Norwegian Institute of Technology, 1988.

[27] J. Peckham and F. Maryanski. Semantic data models. *ACM Computing Surveys*, 20(3), September 1988.

[28] C. A. Petri. Kommunikation mit Automaten. *Schriften des Rheinisch-Westfalischen Institut für Instrumentelle Mathematik an der Universität Bonn*, (2), 1962.

[29] W. D. Potter and R. P. Trueblood. Traditional, semantic and hyper-semantic approaches to data modeling. *IEEE Computer*, 21(6):??, June 1988.

[30] S. Radhakrishnan. *Indian philosophy.* Allen & Unwin, London, 1951.

[31] C. A. Richter. An assessment of structured analysis and structured design. *SIGSOFT Software Engineering Notes*, 11(4), 1986.

[32] N. W. Ross. *Three Ways of Asian Wisdom.* Simon & Schuster, New York, 1966.

[33] James Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[34] S. Shlaer and S. J. Mellor. *Object-Oriented System Analysis: Modeling the World in Data.* Prentice Hall, Englewood Cliffs, first edition, 1988.

[35] Yair Wand. An ontological foundation for information systems design theory. In B. Pernici and A.A. Verrijn-Stuart, editors, *Office Information Systems: The Design Process.* Elsevier Science Publishers B.V. (North-Holland), May 1989.

[36] P. T. Ward. The transformation schema: an extension of the data flow diagram to represent control and timing. *IEEE Transactions on Software Engineering*, 12(1):22–32, January 1986.

[37] A.I. Wasserman, P.A. Pircher, and R.J. Muller. The Object-Oriented Structured Design Notation for Software Design Representation. *IEEE Computer*, 23(3), March 1990.

[38] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990. 341 p.