in "Object-oriented Databases: Analysis Design and Construction", R.Meersman and W.Kent (eds), North-Holland

PROOF-THEORETIC SEMANTICS OF OBJECT-ORIENTED SPECIFICATION CONSTRUCTS

J.FIADEIRO[1*], C.SERNADAS[2], T.MAIBAUM[1] and G.SAAKE[3]

[1]Dpt Computing, Imperial College of Science, Technology and Medecine, London SW7, UK
[2]Dpt Matemática, Instituto Superior Técnico, Av. Rovisco Pais, 1096 Lisboa, PORTUGAL
[3]Abt Datenbanken, Techn Universität Braunschweig, Postfach 3329, 3300 Braunschweig, FRG

A formal semantics for the kernel constructs of an object-oriented specification language is presented. The formal counterparts of objects as the basic building blocks of information systems are given by theory presentations in a logic that has been developed to support the required object-oriented specification mechanisms. Attributes (structure) and events (behaviour) are integrated in coherent logical units (focused on a logical rôle for signatures) around which the notion of locality (encapsulation) is formalised. Objects can be specified directly through formulae of the logic, describing the effects of the events on the attributes as well as the restrictions and requirements on their occurrence. Aggregation, inheritance and particularisation are formalised as specification constructs acting on a context (a diagram in the category of theory presentations) where previously built specifications are stored, thus allowing to assemble large specifications from existing ones. The derivation of safety and liveness properties from specifications using the inference rules of the logic, and the use of the structure of specifications to direct these proofs is also illustrated. In this way, we hope to contribute towards the necessary formalisation of object-oriented information systems design.

1. INTRODUCTION

One of the (few) consensi that exist in the field of object-oriented software development is the lack of formal foundations [1,3]. However, without a solid theoretical framework, it is impossible to establish a sound understanding of the concepts and techniques involved, something that is essential in order to produce provably correct systems, guiding development from requirements capture, through design down to implementation. Hence, there is some urgency for such a formalisation effort. In this respect, we share the belief that in order to attempt a serious and faithful formalisation of object oriented development we must step back from existing practices, and try to address what is essential in object-orientation as a design paradigm. There seems to be little use in formalising existing techniques that have evolved in an ad-hoc way.

---

* On leave from Departamento de Matemática, Instituto Superior Técnico, Lisboa, as a grantee of the Commission of the European Communities

Fortunately, recent developments [7,8,32,35,54,57] seem to show that the concepts underlying the object-oriented computation model are stabilising to a point where it becomes possible to attempt a formalisation of the concept of object as a structuring unit for information systems and database design. Indeed, as argued in [50], the concept of object as an "entity" that has an identity independent of its state, that encapsulates a collection of attributes (its private memory) which it is able to manipulate according to a well defined set of actions, and that is able to communicate with other objects, either synchronously (event sharing) or asynchronously (message passing), is rich enough to capture the wide variety of phenomena that are normally encountered in design. It then becomes possible to consider each layer of the systems development process to be structured uniformly as a collection of interacting, fully concurrent objects. By incorporating states and behaviour, this view provides us with a way of dealing uniformly with data and processes, allowing us to abstract from the biases dictated by current technological support at the earlier conceptual levels. During refinement, it allows us to regard each design step as the implementation of some abstract object in terms of a collection of concrete ones that are "assembled" into a configuration that provides the functionality required by the abstract object. Naturally, the process ends when a layer of objects that do not need to be implemented is reached, either because they correspond to "real-world" entities, or they are supported by the chosen programming and database environment, or by the existing operating system.

From this point of view, it seems obvious that the required formalisation effort is, at least, twofold. On the one hand, it must be directed towards the definition of a formal notion of object as a specification primitive that is powerful and abstract enough to capture the above mentioned features throughout the whole development cycle. Naturally, this formal notion must be such that the desired levels of modularity may be achieved at each level (ie, it must support horizontal composition [24]). On the other hand, a corresponding notion of refinement must be developed (ie, supporting vertical composition [24]). The goal of this paper is to show how a collection of specification constructs may be formally defined that supports the former effort. We should stress that we shall not attempt to provide a full and practical specification language that can be used for object-oriented design. We shall have to limit ourselves to concentrate on the definition of our main primitive of specification (formalising the notion of object) together with two well known specification constructs: inheritance and aggregation (complex objects). However, we do not see deep problems in extending our results to other useful constructs such as class/type grouping and parameterisation.

With respect to the nature of the formal entities that we shall be manipulating, we shall adopt the so-called "proof-theoretic" (or "axiomatic") approach to systems development, which has been explored both for software engineering in general [37,38], databases [eg, 22] and conceptual modelling in particular [52], and according to which each layer of the design process is viewed as a theory presentation in a certain logic. One of the advantages of this approach is that it provides a uniform view of the development process in terms of

manipulation of syntactical entities (theory presentations), ending with one that is "executable" (is supported by the chosen environment). Moreover, the proof theory of the logic provides us with the necessary mechanisms for querying each such layer in terms of the properties that are actually enforced by the specification. Finally, as argued in [21,51] this view is also well suited for information systems design because conceptual modelling techniques may be formalised as operations on theory presentations, allowing us to view knowledge bases as structured theories.

In terms of the intended specification constructs, this means that it is necessary that our formalism supports the desired object-oriented techniques by providing a formal counterpart to objects as the basic building blocks. The obvious building blocks in the proof-theoretic approach are the theory presentations of the logic. Hence, the logic to be adopted must be able to provide the necessary mechanisms that allow for formalising objects as theory presentations. That is to say, it must be possible to address through a logical coherent unit both structural and behavioural phenomena, as well as to incorporate into the logic the notion of locality ("data abstraction") as providing the criterion for recognising theory presentations as true specification modules. On the other hand, it must be possible to compose simpler specifications (theory presentations) to obtain more complex ones, thus bringing the desired degree of modularity to the formal level. In order to formalise the required structuring principles, we shall use tools from category theory following the dogma being preached by J.Goguen [eg, 25].

From the point of view of the specification language, each expression will be interpreted as an operation acting on a context that provides us with names for theory presentations and morphisms so that they can be used to build more complex specifications. That is to say, the semantic domains will be given in terms of diagrams in the category of theory presentations. For a model-theoretic semantics of a similar language, ie where the specification constructs are seen to manipulate models instead of theories, please refer to [47]. The proof-theoretic view will be further discussed in section 2, including a motivation for the logic that was adopted. This logic is briefly presented in section 3 as part of the definition of the semantics of the primitive expressions of our language. Finally, the structuring mechanisms, respectively inheritance, particularisation, and aggregation, are studied in section 4 and 5.

In this way, we hope to make a contribution to the referred effort towards a solid theoretical framework for object-oriented specification. In fact, research on formal foundations started as early as [23], but the main thrust has only been given more recently, partially as a result of the progress in the areas of abstract data types and concurrency, and there are still many areas besides the proof-theoretic semantics of specification languages that need attention. Some examples of recent work include the combination between the object-oriented paradigm and other well known paradigms like the functional paradigm [28] and the deductive database paradigm [36]. More specific work is also going on which is related, for instance, to the formalization of inheritance [3,5,12], extensions to accomodate concurrency and distribution

[4], and to giving denotational semantics to object-oriented languages [31,47,55]. Actually, the work reported herein is also part of an integrated research effort on object-oriented information systems design that started with [48], and which is also addressing the algebraic fundamentals of the notion of object [10,11], logical calculi for reasoning about objects [16], as well as methodological aspects of object-orientation [50,53].

2. PRELIMINARIES

A specification language for object-oriented conceptual modeling must at least include constructs for specifying primitive objects, particularisations of predefined objects, inheritance relationships between objects, and aggregation of objects in order to define more complex objects. Other operations, such as parameterisation, are also essential for a workable specification language, but we shall concentrate herein on the above mentioned constructs. We recall that our aim is not to propose a new specification language, but merely to show how certain constructs that are useful for object-oriented specification can be given a proof-theoretic semantics. At the top level of our syntax we shall consider, therefore, four constructs:

```
expression = primitive | particularisation | inheritance | aggregation
```

In this section, we shall define the intended semantic domains, starting by making precise what we mean by a proof-theoretic semantics.

2.1. Proof-theoretic semantics

We have already mentioned that our approach is to adopt theory presentations in a given logic as semantic units. That is to say, the semantics of primitive expressions will be given by theory presentations, and the semantics of other constructs will be given through appropriate operations on the category $\mathcal{THE}$ of theory presentations. But, what category is this?

A concept of logical formalism that has been found abstract enough to formalise specification building, including conceptual modelling and knowledge representation [21], identifies three components in a logic [19]:

- a category of signatures - providing the required structures of non-logical (or extra-logical) symbols,

- a functor from this category to the category of sets - providing the set of formulae over each signature, ie providing the grammar of the language of the logic, including its logical symbols,
- a consequence relation - providing for each set of formulae the set of its consequences.

This view is closely related to institutions [27], except that models and satisfaction were replaced by a consequence relation in order to stress that it is not in the model theory of the logic that we are interested, but rather on the nature of the consequence relation that is supported. A theory presentation for such a logic consists of a pair (θ,Φ) where θ is a signature and Φ is a collection of formulae for that signature (usually referred to as the axioms of the specification). The theory presented by (θ,Φ) consists just of the set of formulae that are consequences of Φ (its theorems). The difference between theories and their presentations is just that theories are closed under consequence and, hence, generally infinite and not necessarily recursive sets. Theory presentations are, on the other hand, generally finite so that they can serve as specification units.

In a category of widgets, morphisms are structure-preserving mappings between widgets ie, morphisms define the notion of structure that the category is intended to capture. Hence, signature morphisms reflect the structures of non-logical symbols to be used. These are generally expressed through functions between indexed-sets. A more interesting case is given by the intended category of theory presentations. In this case, we want morphisms to be mappings that preserve the "meaning" of the presentations. The meaning of a theory presentation is taken herein to be the presented theory, ie the collection of its theorems. (An alternative view for a model-theoretic semantics would take the collection of its models, or some representative, as the ultimate meaning of a presentation.) Hence, a morphism σ between two theory presentations (θ,Φ) and (θ',Φ') is naturally defined as a morphism between the signatures θ and θ' such that, for every formula f for θ, if $(\Phi \vdash_{\theta} f)$ then $(\Phi' \vdash_{\theta'} \sigma(f))$. Notice that by $\sigma(f)$ we are denoting the translation of the formula f that is induced by the functor that defines the language of the logic.

This condition is not easy to check because it requires that every theorem of the domain presentation be proved as a theorem of the codomain. Depending on the properties of the consequence relation, some logics allow us to simplify the condition by merely requiring that every axiom of Φ be translated to a theorem of (θ',Φ'). This is an important simplification because, specifications being usually built by extending old ones, ie by adding more non-logical symbols and non-logical axioms, it would ensure that extensions provide morphisms. Indeed, the inclusion $\Phi' \supset \Phi$ would trivially satisfy the condition. Unfortunately, not every logic allows for this simplification, in particular the logic that we shall adopt for specifying objects. In these cases, this means that there are assumptions that are not expressed in Φ but "hidden" in $\vdash_{\theta}$ and not captured by $\vdash_{\theta'}$ and that, hence, must also be coded somehow in (θ',Φ') in order to obtain a morphism. A well known case is given in many-sorted first-order logic with non-empty domains [14]: a morphism between two theory presentations exists

only when the target theory ensures that the domains of the relativisation predicates are non-empty. Fortunately, there are ways of dealing with these hidden assumptions, and we shall illustrate them in section 4.

The specific logic that was chosen to support the formalisation of our object-oriented specification constructs will be progressively defined throughout the paper. Naturally, as a starting point, we might wonder whether the logics that have been adopted for formalising the specification of abstract data types (in general, more or less exotic first-order equational formalisms) would do the job. However, the work that has been developed mainly by the so-called ADT school since [26,58] has formalised the principles for specifying "in the large" (structured specifications, modularisation, implementation, …) around the notion of *value* as opposed to *object*. As put forward in [45], although both concepts play an important rôle, they also reflect a fundamental technical distinction. The notion of a data type as a collection of data domains and operations on these domains is purely *applicative*. In contrast, the concept of object involves *imperative* notions (eg the notion of state, of action) that cannot be accommodated easily within this applicative view. We do recognise that it is possible to model the notion of state as a value (as another data domain) so that actions are just operations on that additional data domain, but this does not seem to provide the right level of abstraction at which we want to deal with objects. On the one hand, we do want objects to have an identity independent of their state so that they can be named and manipulated as a coherent unity and, on the other hand, the treatment of the behavioural notions, in particular the notions of interaction, safety and liveness which are intrinsic to objects as dynamic entities, does not seem to be given a satisfactory account in this applicative way.

Hence, although recognising the rôle to be played by values in our formalisation of objects, we would like to adopt a more abstract level for dealing with states and behaviour. Taking into account the great success of formal accounts of change and behaviour based on temporal or dynamic logics, namely for reasoning about *concurrency* (since [43]), and for the specification of information systems and databases (since [46]), it seemed obvious that we should turn to modal formalisms as an alternative to first-order logics. However, it still remained to define a logic that would support the desired degree of modularity, ie such that its theory presentations might be used as true specification modules, thus providing a faithful formalisation of the notion of object.

The chosen logic satisfies these requirements, and follows previous work on action, deontic and temporal logics (eg, [15,17,18,20,33,34]). Basically, a combination of modal (action) operators (as in dynamic logic [29]) and of deontic predicates of permission and obligation on actions is used to describe and prescribe behaviour as suggested in [34], and temporal operators are used to reason about the safety and liveness properties of the described objects following [15]. It will not be possible to give a full account of this logic in the paper, namely of the proof theory that is being developed for deriving properties of specifications. The reader interested in more specific details should consult [15,16,20] for several of its

fragments, but not for its application to the formalisation of the above mentioned object-oriented constructs, which is being presented for the first time in this paper.

2.2. Contexts

The semantics of the object-oriented language constructions that we define is based on the concept of context (in a way similar to what is done, for example, in [6,13] in the area of languages for the specification of abstract data types). Thus, when giving the semantics of a language construct, we assume we have as a context a diagram of theory presentations for indicating the state of the specification so far. We may also think that this context interacts somehow with a library where we store the specifications that we build. The intended view of specification building is to be able to have such a library from which we are able to collect previously defined specifications in order to extend them, or to assemble them into configurations that provide the required functionality. Instead of having to build from scratch, our work may just consist on having to provide the required extensions, or to provide the necessary interfaces between the components that we want to assemble. However, the purpose of the paper is not to discuss the problems of creating or maintaining such a library. A lucid discussion on those matters, including a language for library interconnection, may be found in [24].

Hence, our constructs will be seen to enrich a context that provides us with names for both theory presentations and morphisms so that they can be used to build more complex specifications. In other words, the semantics of each expression in the language is a function that, when applied to a context, returns another context. In this way, we can easily define an attribute grammar for our specification language by introducing the context dependencies as restrictions upon changes of the context. However, and again for simplicity, we shall not detail such an attribute grammar.

This context will be seen as a diagram in the category of theory presentations of our logic, ie a graph whose nodes are labelled by theory presentations and whose edges are labelled by morphisms between theory presentations:

Definition 2.2.1: A context is a pair $(\mathcal{N}odes, \mathcal{E}dges)$ where

- $\mathcal{N}odes$ consists of a set $|\mathcal{N}odes|$ (of nodes), and a labelling function π that associates with every $n \in |\mathcal{N}odes|$ a theory presentation $\pi(n):\mathcal{THE}$. We shall often consider π to be a pair (θ,Φ) where $\theta(n)$ gives the signature of the theory presentation and $\Phi(n)$ gives the respective set of formulae (axioms).
- for each pair (n,n') where $n,n' \in \mathcal{N}odes$, $\mathcal{E}dges(n,n')$ consists of a set $|\mathcal{E}dges(n,n')|$ (of edges between n and n'), and a labelling function μ that associates with every $e \in |\mathcal{E}dges(n,n')|$ a morphism $\mu(e): \pi(n) \rightarrow \pi(n')$. □

Given an expression `expr` in our specification language, we shall denote by ⟦`expr`⟧ the associated operation, so that ⟦`expr`⟧(*Nodes*,*Edges*) denotes the context that results from applying the operation to the current context (*Nodes*,*Edges*). We shall often write (⟦`expr`⟧*Nodes*, ⟦`expr`⟧*Edges*) instead of ⟦`expr`⟧(*Nodes*,*Edges*) in order to give separate definitions of the effects of the constructs on the nodes and edges of the context.

3. PRIMITIVE EXPRESSIONS

At the most basic level, an object specification may be given through a primitive expression which consists of a name (identifier) and a body where all the information that is relevant to the definition of the target object is given. This structure reflects the idea that an object is to be specified as a dynamic entity which has a state that can be observed through a collection of attributes, and whose overall behaviour is determined by the events that are allowed and required to occur during the object's lifetime. Hence, several components are distinguished in a body:

```
primitive = object id body end
body = data events attributes valuation safety liveness
```

As the running example of this section, we shall use the following specification of products within a stock management universe of discourse:

```
object product
  importing data types natural
  events
    set-price(nat), upgrade(nat)
  attributes
    price, version: nat
  valuation N:nat
    [set-price(N)]price = N                    (V1)
    [upgrade(N)]price = price+10               (V2)
    [set-price(N)]version = version            (V3)
    [upgrade(N)]version = N                    (V4)
  safety N:nat
    Per(set-price(N)) → price≤N                (S1)
    Per(upgrade(N)) → version≤N                (S2)
end
```

Intuitively, we are specifying a product as an object whose state consists of a price and a version (`attributes`), the operations on this state consisting of setting new prices and upgrading the current version (`events`). The effects of the events are described under `valuation`, and the restrictions on their occurrence (ie, when they are permitted to happen)

are specified under `safety`. We shall explain this example more thoroughly throughout the section.

In terms of the intended proof-theoretic semantics, a primitive expression defines a theory presentation in the chosen logic, ie a signature and a collection of formulae. Naturally, a specification language should provide us with the means for defining such signatures and formulae. For simplicity, we shall use only a minimal set of constructions and use directly the syntax of the logic. That is to say, `data`, `event`, `attributes`, `valuation`, `safety`, and `liveness` will be given in terms of the linguistic structures of our logic, which will be defined and illustrated below. But, we insist that this does not mean that we are advocating the language of the logic as a specification vehicle. Rather, our view is that the language of the logic will serve best as a lower-level language to which a specification language may be compiled.

3.1. Signatures and interpretation structures

Signatures are defined under the clauses `importing data types`, `attributes` and `events`. Each of these clauses addresses one of three different components that are essential to the specification of an object: the *data component*, the *attribute component*, and the *event component*. The data component describes the information that is state-independent and which provides the data context in which the object is placed. It provides the data types that are necessary to define the domains and codomains of the attributes, as well as the parameters of the events. The attribute component describes the information that is state dependent, like the program variables of a program or the records in a database. The event component accounts for the (atomic) actions that the object is potentially able to perform. These components are reflected in the definition below:

Definition 3.1.1 (object signature): An *object signature* θ is a triple (Σ,A,Γ) where

- Σ is a signature in the usual algebraic sense [13], ie a pair (S,Ω) where S is a set (of sorts) and Ω is a $S^*\times S$-indexed family (of function symbols). This is the data signature associated with θ.
- A is a $S^*\times S$-indexed family (of attribute symbols).
- Γ is an $S^*\times E$-indexed family (of event symbols). The symbol E stands for the sort of events.

We shall call $U(\theta)=(S\oplus\{E\},\Omega\oplus\Gamma)$ the universe signature associated with θ. □

The families Ω, A, and Γ are assumed to be disjoint and finite (ie, there are only a finite number of function, attribute and event symbols).

In the case of products, and assuming that under `natural` we are importing the data signature

$$\Sigma(\texttt{natural})=(\{nat,bool\},\{true_{bool},false_{bool},zero_{nat},suc_{nat,nat},+_{nat,nat,nat},\leq_{nat,nat,bool}\})$$

we shall have

$$\theta(\texttt{product}) = (\ (\{nat,bool\},\{true_{bool},false_{bool},zero_{nat},suc_{nat,nat},+_{nat,nat,nat},\leq_{nat,nat,bool}\}),\ \{price_{nat},version_{nat}\},\ \{set\text{-}price_{nat,E},upgrade_{nat,E}\}\ )$$

That is to say, our attribute symbols are `price` and `version`. Each of them conveys information that is state dependent: respectively, the current price of the product and its current version. This state information will be updated by events generated in terms of the declared event symbols: `set-price` and `upgrade`. We shall see below how the effects of these events on the attributes is specified. Moreover, as we shall see later on, only these events will be allowed to update prices and versions. This is the idea of encapsulation that is characteristic of object-orientation.

Generalising, a primitive expression

```
object name
  importing data types data
  attributes A
  events Γ
  valuation V
  safety S
  liveness L
end
```

introduces the signature $\theta(\texttt{name})=(\Sigma(\texttt{data}),A,\Gamma)$ where $\Sigma(\texttt{data})$ is the algebraic signature stored under the name `data`. For simplicity, we shall abstain from formalising the specification of the data type component of our specifications. This may be done as in the many successful languages that have been developed for the specification of abstract data types (eg, Clear [6]). Hence, we shall always resort to importing specifications that we have stored in some data type dictionary which, given a name `data`, returns an algebraic signature $\Sigma(\texttt{data})$ and a collection of equations $\Phi(\texttt{data})$.

Before proceeding to the definition of the language supporting the specification of the valuation, safety and liveness components, it is worthwhile giving a flavour of the intended model theory. A semantic interpretation structure for an object signature is given by an algebra that interprets the data and event parameters, a mapping that gives the values taken by the attributes in traces (finite sequences of events), thus accounting for the state observations,

and two relations between events and traces (permission and obligation) accounting for the underlying process:

Definition 3.1.2 (θ-interpretation structures): A *θ–interpretation structure* for a signature $\theta=(\Sigma,A,\Gamma)$ is a quadruple $(\mathcal{U},\mathcal{I},\mathcal{P},\mathcal{O})$ where:

- $\mathcal{U}$ is a $U(\theta)$-algebra.
- $\mathcal{I}$ maps each $f\in A_{<s_1,\ldots,s_n>,s}$ to $\mathcal{I}(f)$: $s_{1\mathcal{U}}\times\ldots\times s_{n\mathcal{U}}\times E_{\mathcal{U}}^{*}\rightarrow s_{\mathcal{U}}$
- $\mathcal{P}$ and $\mathcal{O}$ are relations in $E_{\mathcal{U}}\times E_{\mathcal{U}}^{*}$. □

We recall that, for each sort symbol $s\in S$, $s_{\mathcal{U}}$ is its interpretation (a set) in the algebra $\mathcal{U}$. Hence, $E_{\mathcal{U}}$ is the set of events of the interpretation structure. As usual, $E_{\mathcal{U}}^{*}$ denotes the set of finite sequences of events (traces). Traces will play the rôle of "possible worlds" in the interpretation of the modal language to be proposed below, so that an interpretation structure provides the necessary Kripke semantics. Hence, attribute symbols are assigned trace-dependent interpretations. This reflects the fact that attributes convey state-dependent information, such as the price of a product. On the other hand, a fixed interpretation (the algebra $\mathcal{U}$) is assigned to the universe component (including the event symbols). For instance, this means that the symbol '+' is interpreted at each trace (state) by the same function. Finally, the two relations $\mathcal{P}$ and $\mathcal{O}$ tell us, for each trace, which events are, respectively, permitted (next) and required to occur (sometime) after that trace.

We should stress that this semantic notion differs from more conventional ones, and in particular from [10,11,12], where sets of life cycles account for the process model of an object. The deontic notions of permission and obligation were preferred because as explained in [34], on the one hand, they are more general, allowing us in particular to formalise other kinds of information, like error recovery through corrective actions, or sanctions, if desired (see also [40,41,56]), and, on the other hand, because they will support a more descriptive account of the behaviour of an object in the sense that we will not be modelling directly the allowed life cycles but describing instead the behavioural properties that traces which are life cycles must satisfy. This has an impact at the formal level by separating the absence of normative behaviours (empty set of life cycles) or the occurrence of non-normative behaviour (ie, having a trace that is not a life cycle) from inconsistency of a specification. We shall illustrate this point in the next section.

Finally, it is important to analyse a restriction to the notion of θ–interpretation structure that will be used to support, at the model-theoretic level, the formalisation of the desired notion of *locality*, which is the "structural" property of objects. As motivated before, objects have an intrinsic notion of locality according to which only the events declared for an object can change the values of its attributes. This notion of locality is incorporated into the notion of θ-interpretation structure through the requirement that non-generated (ie, non-local) events do not interfere with the attributes of the object:

Definition 3.1.3 (θ-loci): Given an object signature $\theta=(\Sigma,A,\Gamma)$, a θ-interpretation structure $(\mathcal{U},\mathcal{I},\mathcal{P},\mathcal{O})$ and a trace ω, let $\mathbf{G}_{\mathcal{U}}=\{e\in \mathbf{E}_{\mathcal{U}} \mid e=g_{\mathcal{U}}(b_1,\ldots,b_n),\ g\in\Gamma\}$. Events belonging to $\mathbf{G}_{\mathcal{U}}$ are said to be *generated*. The θ-interpretation structure is a *θ–locus* iff for every trace ω, for every $e\notin \mathbf{G}_{\mathcal{U}}$, for every $f\in A_{<s_1,\ldots,s_n>,s}$ and $(a_1,\ldots,a_n)\in s_{1\mathcal{U}}\times\ldots\times s_{n\mathcal{U}}$, $\mathcal{I}(f)(a_1,\ldots,a_n,\omega) = \mathcal{I}(f)(a_1,\ldots,a_n,e::\omega)$. □

That is to say, in a θ–locus, the values of the attributes can be changed only by the generated events. In other words, locality declares that all the possible manipulations of the attributes are restricted to the events that were declared in the signature. For instance, in terms of products, this means that we are only allowing interpretation structures where the price of a product can be changed by the events that are generated in terms of `set-price` or `upgrade`. Hence, although we are working with global traces, we are requiring that these global traces be "local" (private) states (identifying, as usual, states with the interpretations of the attribute symbols - observations).

Naturally, this notion of θ–locus does not formalise locality per se: it must be understood in conjunction with the proof-theoretic account. Rather, our ultimate purpose is to provide a proof-theoretic formalisation of locality, for which the notion of θ–locus given above provides a correct model-theoretic basis. A full understanding of our formalisation cannot be given without referring to the notion of signature, and to the notion of morphism of specifications to be introduced in section 4. Signatures define the boundaries of our objects, ie locality has to be understood relative to a signature. On the other hand, intuitively, morphisms convey our notion of structure, which in the case of object-orientation we take to be locality-preservation. We shall see later on that the notion of morphism to be proposed will indeed capture the desired notion of encapsulation so that an object cannot refer to the attributes of another object (for "read" or "write" purposes) without incorporating it as a component.

Hence, the components (objects) that we specify are "context" independent in the sense that state information is localised rather than shared between components, a strategy for achieving high levels of modularity that can be traced back to [42]. However, notice that locality concerns only the attributes of an object: permissions and obligations are global notions because they are properties of events, which have to be global in order to capture interaction between objects through event sharing.

3.2. Valuation, safety and liveness

The axioms of a specification may be divided into four components: the data specification, the valuation specification, the safety specification and the liveness specification. Again, we

shall assume that the data specification (eg, the theory of natural numbers) is imported from previously defined data types.

VALUATION

Intuitively, the valuation formulae indicate the effects of the events upon the attributes. For example the valuation equation

```
[set-price(N)]price = N
```

indicates that after the occurrence of an event generated from `set-price(N)` the attribute `price` will have the value N. This is a formula of the logic of positional terms defined in [20]. Although, for simplicity, we shall not provide a full account of this logic, it is worthwhile discussing more carefully some of its features.

The most primitive syntactical category is that of a term. There are several categories of terms. Terms built from the universe signature (function and event symbols) are rigid, ie they denote the same value in every trace. On the other hand, terms that are built using attribute symbols are non-rigid, ie their value may change from trace to trace. For instance, the values of the terms `price`, `version`, `price+10` will change in the light of the events that occur. In order to refer to the value that the attributes take after the events occur we use *positional (modal) operators*: given a term t and an event term e, ([e]t) is also a term and denotes the value taken by t after the event denoted by e occurs (in the current state). Summarising, we have the following syntax for positional terms:

```
pos-term = simple | '['event-term']'pos-term | '[]'pos-term
simple = data-term | event-term
event-term = event-symbol '('seq-args')'
data-term = attribute-symbol '('args')' | function-symbol '('args')'
args = empty | data-term args
```

We shall illustrate the use of the operator [] in the next section: it will be used to denote the values taken by attributes in the empty trace, ie in the inital state. A more formal account of the syntax and interpretation of positional terms can be found in [16,20].

Terms are used to form equations such as the one above, which can be used to form more complex formulae using the usual propositional and first-order connectives:

```
positional-formula = atomic-pos-formula | complex-pos-formula
atomic-pos-formula = pos-equation | 'Per''('event-term')'
                     | 'Obl''('event-term')'
pos-equation = pos-term '=' pos-term
complex-pos-formula = ... /* the usual first-order connectives */
```

Given an interpretation structure, a trace, and an assignment of values to variables, these connectives are interpreted as usual in the first-order model defined for the trace by the interpretation structure, the predicate symbols `Per` and `Obl` being interpreted, respectively, by the relations $\mathscr{P}$ (permission) and $\mathscr{O}$ (obligation). An interpretation structure makes a formula *true* when the formula holds for every possible trace and assignment of values to variables. Hence, in a specification, a formula is always implicitly universally quantified and asserts something that holds in every possible trace.

Hence, the remaining valuation formulae

```
[upgrade(N)]price = price+10
[set-price(N)]version = version
[upgrade(N)]version = N
```

specify that upgrades increase prices by (say) 10 units and update the current version to the argument number, and that setting new prices does not update the current version of the product.

It could be argued that specifying the effects of every event over each attribute is a titanic task that cannot be fulfilled by a specifier/programmer, and that some mechanism (frame rule) would be welcome that would "guess" what the specifier means when he does not state what the effects of an event over an attribute are. We shall not attempt to give a complete answer to this question right now. We shall come back to it in the next sections. However, we should point out that one of the characteristics of object-orientation is the "compositional" or "bottom-up" direction of specification by which we build complex specifications by putting together smaller ones. The principle of locality (encapsulation) assures us that by building on top of smaller objects we do not violate their locality so that we do not have to worry about interference of events of one object over the attributes of another object. Hence, the specifier may concentrate on the valuation specification of primitive "small" objects, whose complexity he may control, and rely on the constructs of the language to build more complex objects.

SAFETY

Safety rules indicate conditions that must be fulfilled so that an event is permitted to happen. They are expressed through formulae involving the predicate symbol `Per` which is interpreted by the relation $\mathscr{P}$ (giving the events that are permitted to occur immediately after each trace). For instance, the safety rule

```
Per(set-price(N)) → price≤N
```

states that events generated from `set-price(N)` are only permitted to happen provided that the value of the attribute `price` is less than or equal to N. That is to say, such events are only permitted to be used for increasing prices. This means that we regard price decreases to

be "possible" but not permitted. That is, the occurrence of a price decrease is regarded as a non-normative state-transition, but is not inconsistent with the specification. We shall see in section 3.3 how the "deontic" flavour of this predicate is formalised.

In general, we give only necessary conditions for an event to be permitted to occur. This is a consequence of the fact that events may be shared by several objects: it is hard to know exactly what are the conditions under which an event is permitted because, within each object, we have only partial (local) information on the events. We want to be able to give a specification of an object that will hold in any possible environment in which the object is placed. Requiring an event to be permitted in certain circunstances is a very strong condition that may lead to insconsistencies when the object is put in a society where that event is shared with other objects. For instance, consider the case of events that interpret `set-price(N)`. If we said that these events were permitted when N was greater than `price(N)`, then we would not be able to integrate this object in an organisation where price-increases were restricted to certain seasons within the year. But these are only methodological points as the specifier will have the freedom to do as he pleases. However, it is interesting to note that specification languages such as [50] take this approach and allow only for necessary conditions on permissions. Moreover, if we want to consider the object in isolation, we can always ask for the completion of such necessary conditions into an equivalence.

LIVENESS

Finally, liveness specifications cater for the conditions under which we require that events must occur. Not every object requires such liveness conditions. This is the case for products. Such objects are said to be *passive*. Objects that do have liveness requirements are said to be *active*. Such liveness conditions are specified using the predicate symbol `Obl` (interpreted by the relation $\mathcal{O}$), and are usually written in the form:

$$\text{condition} \rightarrow \texttt{Obl}(\text{event})$$

ie, we tend to give sufficient conditions under which an event is obligatory. The motivation is as for permissions except that, as a result of sharing, more obligations (and not less) usually result. We shall see examples of such liveness specification in the next section.

SUMMARY

Summarising, a primitive expression defines a theory presentation, ie a signature and a collection of formulae, the (non-logical) axioms of the specification. The semantics of this construct can be given in terms of its effects on the context:

```
⟦object name
  importing data types data
  attributes A
```

events Γ
valuation V
safety S
liveness L]]$\mathcal{N}odes$ = $\mathcal{N}odes$ ∪ (name,(θ(name),Φ(name)))

where θ(name)=(Σ(data),A,Γ) and Φ(name)=Φ(data)∪V∪S∪L. Naturally, $\mathcal{E}dges$ remains unchanged.

3.3. Reasoning from specifications

Our proof-theoretic semantics assigns to each node (name) of the context a pair (θ,F) where θ is a signature and F is a collection of formulae. Such a pair is a theory presentation in the logic. The theory presented by this pair consists of the set of all formulae that can be derived from F using the proof-system of the logic, and gives us the properties of the specified object (its meaning). In order to reason about such properties of a specification, it is convenient to introduce a syntactical counterpart of the "in a model" notion of consequence which can be given in terms of sequents of formulae over the same signature:

Definition 3.3.1 (assertions): If F is a (finite) set of formulae over a signature θ, and f is also a formula over θ, ($F \Rightarrow_\theta f$) is an *assertion*. Such an assertion is said to be *valid* iff every θ–locus that makes all the formulae in F true also makes f true. □

We shall often write (name ⇒ f) instead of (Φ(name) $\Rightarrow_{\theta(name)}$ f) to refer to the properties of a specification introduced under a certain name.

Assertions allow us to reason about properties that hold in any trace of an object. However, we usually attach an operational meaning to the notion of permission and obligation in terms of which we can speak about the normative behaviours of an object: those where events occur only when permitted and where obligatory events are eventually performed. These normative behaviours characterise the safety and the liveness properties of an object:

Definition 3.3.2: Given a θ-interpretation structure $\mathbb{S}$=($\mathcal{U}$,$\mathcal{I}$,$\mathcal{P}$,$\mathcal{O}$), a *trajectory* $\mathcal{T}$ for $\mathbb{S}$ and θ is an infinite sequence over E_θ. A *safe* trajectory $\mathcal{T}$ is such that, for every i, ($\mathcal{T}$(i+1),$\mathcal{T}_i$)∈ $\mathcal{P}$. A *live* trajectory $\mathcal{T}$ is such that, for every i, (e,$\mathcal{T}_i$)∈ $\mathcal{O}$ implies that there is j>i such that e=$\mathcal{T}$(j). A safe and live trajectory is said to be *normative*. □

That is to say, a safe trajectory is one where each event in the sequence is permitted in the trace consisting of the events that have already occurred, and a live trajectory is one where every event that is obligatory after a prefix of the trajectory will occur later on in the trajectory.

As is well known, temporal logic has been found to be quite suitable for reasoning about safety ("something bad will never happen") and liveness properties ("something good will eventually happen") [eg 39]. In [15] we have already shown how it is possible to reason about such properties of systems from a specification of their behaviour given in terms of permissions and obligations as above. Hence, herein, we shall limit ourselves to a brief overview of the main notions and techniques involved. The previous language (which we shall call positional) may be extended by introducing temporal operators such as **X**, **F** and **G** with the usual flavours: respectively, "next", "sometime in the future", and "always in the future". The temporal operator **X** is also used for constructing terms so that, in every instant, the term (**X**t) denotes the value of t at the subsequent instant. The interpretation of these categories in a trajectory is straightforward (see [15] for details). Naturally, it is also necessary to relate this temporal language with the positional language in which we formulated the valuation, safety and liveness requirements. This can be done by extending the notion of assertion:

Definition 3.3.3: Given a signature θ, a finite set F of positional formulae and a temporal formula f over θ, $(F \overset{\square}{\Rightarrow}_{\theta} f)$ is an assertion. Such an assertion is said to be valid iff f is true in every normative trajectory of every θ–locus where all the formulae of F are also true. □

The properties of the normative behaviours of an object specified through `name` are given by the temporal formulae f for which the assertion $(\texttt{name} \overset{\square}{\Rightarrow} f)$ can be proved to be valid. Because of space limitations, we cannot present an axiomatisation for such a calculus of assertions. The reader interested interested should consult [16]. However, it seems worthwhile to give a flavour of the underlying proof mechanisms.

Consider, for instance, the property "prices never decrease". This is a "safety" property (something "bad" will never happen), maybe not from the point of view of a costumer, which can be expressed by the formula

$$(\texttt{price} \leq \mathbf{X}\texttt{price})$$

That is, at each instant, the value of `price` is less than or equal to the next value of `price` (recalling that **X** is the "nexttime" operator). In fact, this is an example of a *dynamic* integrity constraint because it involves more than one state. Such dynamic integrity constraints are usually proved by showing that they are properties of each possible state transition (event) ie, in our case, that each possible event does not decrease the price. In principle, we should check that *any* possible event does not decrease the price. But, locality (3.1.3) allows us to restrict the analysis of the effects of the events to those that are interpretations of the event symbols that are given in the signature. Moreover, because we are only interested in proving this property for the normative behaviours of the object, we only have to check it when the events are permitted to occur. Hence, we end up with the following inference rule

$$\frac{\texttt{product} \Rightarrow (\texttt{Per}(\text{set-price(N)}) \rightarrow (\text{price} \leq [\text{set-price(N)}]\text{price})) \quad \texttt{product} \Rightarrow (\texttt{Per}(\text{upgrade(N)}) \rightarrow (\text{price} \leq [\text{upgrade(N)}]\text{price}))}{\texttt{product} \stackrel{n}{\Rightarrow} (\text{price} \leq \mathbf{X}\text{price})}$$

This rule tells us that in order to prove that "prices never decrease" is a property of normative behaviours of products we just have to prove each of the antecedants of the rule. These proofs are straightforward. For the case of `set-price`:

1. (price≤N) → (price≤N)	(tautology)
2. [set-price(N)]price = N	(V1)
3. (price≤N) → price≤[set-price(N)]price)	1, 2, substitution
4. Per(set-price(N)) → price≤N	(S1)
5. Per(set-price(N)) → (price ≤ [set-price(N)]price)	3, 4, ✂ ("cut")

And for the case of `upgrade`:

1. (price≤price+10)	(theorem of natural numbers)
2. [upgrade(N)]price = price+10	(V2)
3. (price≤[upgrade(N)]price)	1, 2, substitution
4. Per(upgrade(N)) → (price≤[upgrade(N)]price)	3, monotonicity

Hence, we may infer the conclusion, ie that "prices never decrease" is, indeed, a property of normative behaviours of products.

The rule above is an instance of a schema for proving dynamic safety properties that can be found in [16]. These rules provide us, in part, with the "semantics" of permission in the sense that they show how the permission predicate is to be used. Other rules for proving so-called "static" safety properties can also be found therein. We shall see an example of such a static safety property in the next section. We shall also defer the discussion of liveness properties to the next section, where we shall see how the obligation predicate is to be used.

4. INHERITANCE

Inheritance is one of the main mechanisms for specification that one associates immediately with object-orientation. However, it is difficult to give a precise definition of inheritance that encompasses all that has been and is being done under that banner. Hence, we shall start by putting forward a formal notion of inheritance which allows us to build a specification by extending a given one with more attributes and events but requiring that the locality associated with the given specification be preserved. Then, we shall examine a different

construction (particularisation) where we do not require that locality be preserved, and show how it differs from inheritance from a formal point of view. Basically, inheritance allows us to inherit all the properties of the specified object, whereas particularisation does not. We shall also see how this calls for formulae reflecting this notion of structure preservation. These formulae will end up as axioms of specifications built by inheritance, but not in specifications built by particularisation.

In line with what we have done in the previous section, the problem of inheritance for the data component of specifications will not be addressed. Many formal accounts of inheritance for abstract data types may be found in the literature, eg [5]. We shall concentrate herein on the inheritance of state (attributes) and behaviour (events).

4.1. The inheritance construct

The syntax for inheritance expressions is as follows

```
inheritance = object id inheriting list-arguments body end
list-arguments = argument | argument list-arguments
argument = from id through id
```

As an example, we shall consider the specification of stocks. In order to illustrate immediately the use of multiple inheritance, we shall consider first the specification of an order service that registers the arrival of orders:

```
object order-service
  importing data types order
  events
    arrival(ord),deliver(ord)  /* we assume that the sort ord belongs to order*/
  attributes
    pending(ord):bool          /* tells which orders are currently pending */
  valuation O:ord
    [deliver(O)]pending(O) = false            (V1)
    [arrival(O)]pending(O) = true             (V2)
  safety O:ord
    Per(deliver(O)) → pending(O)=true          (S)
  liveness O:ord
    pending(O)=true → Obl(deliver(O))          (L)
end
```

The axioms of this specification seem self-explanatory. Notice that we have included a liveness requirement, making order-services active objects: an order-service has an obligation to deliver each order that is currently pending. We should point out that we are assuming a "deferred" notion of obligation in the sense that an event that is obligatory in a certain state is not required to occur "next" but sometime in the future. In fact, our logic supports more general obligation predicates with which we are able to impose boundaries for this delayed fulfilment of the obligation (see [16] for more details).

Our specification of a stock can now be given inheriting from `product` and `order-service`:

```
object stock
  inheriting
    from product through prd
    from order-service through orv
  events
    replenish(nat)
  attributes
    qoh:int
  valuation N:nat; O:ord
    ([]qoh)≥0                                  (V0)
    [replenish(N)]qoh = qoh+N                  (V1)
    [deliver(O)]qoh = qoh-req(O)               (V2)
    [arrival(O)]qoh = qoh                      (V3)
    [upgrade(N)]qoh = 1000                     (V4)
    [set-price(N)]qoh = qoh                    (V5)
  safety O:ord
    Per(deliver(O)) → req(O)≤qoh                (S)
end
```

Besides inheriting from `product` and `order-service`, we are adding the new attribute symbol `qoh` meaning that, in a stock, we shall also be recording for the product the quantity on hand that is currently available. Notice that we are assuming that a function symbol `req` (standing for the quantity that each order requests) is imported from the data type inherited from `order-service` so that (V2) specifies that a delivery decreases the quantity on hand by the requested amount. It is important to stress that (V2) specifies the effects of deliveries on the stock independently of the fact that there is sufficient quantity on hand to satisfy the order. In its turn, the safety axiom states that, indeed, deliveries are only permitted when there is enough quantity-on-hand to satisfy the requested amount. This means that, if this pre-condition is violated, the object will enter a non-normative state. Such a situation can be explicitly dealt with (error recovery) by including, for instance, a flag `overdrawn` together with

```
req(O)>qoh → [deliver(O)]overdrawn = true
```

and, possibly

```
(overdrawn = true) → Obl(replenish(-2*qoh))
```

requiring that a corrective action (replenishing the stock with twice the amount overdrawn) be taken. This is the advantage of separating non-normative states from inconsistency. The latter is a situation from which it is not possible to recover within the logic. With the deontic approach, we are allowing for recovery from non-normative situations within the logic. See [34] for more details on this subject.

The new event symbol stands for replenishments of the stock. The new valuation formulae specify the effects of both the new events (replenishments) and the old events (those inherited from `product` and `order-service`) on the new attribute. However, notice that the effects of replenishments on the inherited attributes were not specified. We shall see below how these effects are implicitly defined. With respect to the new valuation axioms, (V0) illustrates two important aspects. On the one hand, we are using the operator [] to refer to the values that the attributes take in the empty trace. Hence, (V0) is an initialisation condition. On the other hand, we are not specifying completely the value that the new attribute takes initially, but merely requiring that it be non-negative. This means that we shall accept any implementation of products that assigns a non-negative value to the quantity-on-hand. The other valuation axioms have the usual form. With (V4) we are implying that an upgrade replaces the existing stock with a thousand new items (for the sake of argument).

4.2. Semantics of inheritance

In order to define the semantics of the inheritance construct, we have to say what are the signature and the set of axioms of the resulting specification. The resulting signature is obtained by adding the new vocabulary symbols to the signatures of the specifications we are inheriting from. Formally, we define the signature resulting from

```
object name
  inheriting
    from name1 through m1
    ...
    from namen through mn
  importing data types data
  events Γ
  attributes A
  valuation V
  safety S
  liveness L
end
```

through $\theta(\mathtt{name})=\theta(\mathtt{name}_1)\cup\ldots\cup\theta(\mathtt{name}_n)\cup(\Sigma(\mathtt{data}),A,\Gamma)$.

In terms of the resulting set of axioms, it will have to include the set of axioms of the specifications we are inheriting from, as well as the new axioms that are given through `data`, V, S and L. That is, we shall have the inclusion

$$\Phi(\mathtt{name}) \supset \Phi(\mathtt{name}_1)\cup\ldots\cup\Phi(\mathtt{name}_n)\cup\Phi(\mathtt{data})\cup V\cup S\cup L$$

However, equality is not enough. Intuitively, because we want inheritance to "preserve structure", ie to preserve the components it inherits from, we need extra axioms reflecting this structure. These extra conditions may be formalised through the notion of morphism

between theory presentations. Indeed, as explained in section 2, morphisms "define" the notion of structure that our formalism will support. In our case, we want this formal notion of structure to reflect what we consider to be the main structuring principle of object-orientation: locality. Hence, we are interested in morphisms between two theory presentations that give us locality-preserving mappings.

As we have seen in section 2, morphisms of theory presentations are defined through signature morphisms. A particular case of a signature morphism is inclusion, the case that matters for inheritance. However, a general definition of morphism between two signatures as defined in section 3.1 can be given as follows:

Definition 4.2.1: Given two object signatures $\theta_1=(\Sigma_1,A_1,\Gamma_1)$ and $\theta_2=(\Sigma_2,A_2,\Gamma_2)$, a morphism σ from θ_1 to θ_2 consists of

- a morphism of algebraic signatures σ_u: $\Sigma_1 \rightarrow \Sigma_2$;
- for each f: $s_1,\ldots,s_n \rightarrow s$ in A_1 an attribute symbol $\sigma_a(f)$: $\sigma_u(s_1),\ldots,\sigma_u(s_n) \rightarrow \sigma_u(s)$ in A_2;
- for each g: $s_1,\ldots,s_n \rightarrow E$ in Γ_1 an event symbol $\sigma_u(g)$: $\sigma_u(s_1),\ldots,\sigma_u(s_n) \rightarrow E$ in Γ_2 □

That is to say, a signature morphism identifies the symbols in θ_2 that are used to interpret the symbols of θ_1. Given such a signature morphism, it is straightforward to define for each formula f over θ_1 its translation under σ which we shall denote by $\sigma(f)$. Once again, in the case of inheritance, the translation reduces to identity so that each formula of θ_1 (the signature we are inheriting from) may be considered as a formula of θ_2.

We have also seen in section 2 that a morphism of theory presentations is just a signature morphism such that each theorem of the domain presentation is a theorem of the codomain. We also mentioned that, in order to check this condition, it would be helpful to be able to check just the axioms of the domain presentation instead of all its theorems, so that trivial cases such as inclusion between the axioms of the two specifications would ensure immediately the existence of a morphism. However, we also pointed out that this is not the case for every logic, and it is easy to see that this is what happens in our case: many of the properties of the component specifications will have been derived assuming encapsulation. For instance, the derivation of the property "prices never decrease" assumed that only price-increases and version upgrades were allowed to update prices. Unless the new specification is able to guarantee that locality is to be preserved, a morphism cannot exist between the two theory presentations because the "meaning" of the domain presentation (its theorems) will not be preserved. This is why the axioms of the new specification defined through the inheritance construct cannot be reduced to the union of the inherited ones and those of the body. That is, as we argued above, there is an "implicit" assumption, locality, whose preservation must also be required in order to obtain a morphism.

It is easier to see why this happens by showing how morphisms are reflected at the level of models. Besides relating the two languages, a signature morphism also defines a relationship between the interpretation structures of the signatures:

Definition 4.2.2: Given two object signatures $\theta_1=(\Sigma_1,A_1,\Gamma_1)$ and $\theta_2=(\Sigma_2,A_2,\Gamma_2)$, and a morphism σ from θ_1 to θ_2, we define for every θ_2-interpretation structure $\mathbb{S}=(\mathscr{U},\mathscr{I},\mathscr{P},\mathscr{O})$ its *reduct along* σ as the θ_1-interpretation structure $\mathbb{S}|_\sigma=(\mathscr{U}|_\sigma,\mathscr{I}|_\sigma,\mathscr{P},\mathscr{O})$ where

- for every $s\in S$, $s_{\mathscr{U}|_\sigma}=\sigma_u(s)_{\mathscr{U}}$, $\mathsf{E}_{\mathscr{U}|_\sigma}=\mathsf{E}_{\mathscr{U}}$;
- for every $f: s_1,\ldots,s_n\to s$ in $\Omega_1\cup\Gamma_1$, $f_{\mathscr{U}|_\sigma}=\sigma_u(f)_{\mathscr{U}}$;
- for every $f: s_1,\ldots,s_n\to s$ in A_1, $\mathscr{I}|_\sigma(f)(a_1,\ldots,a_n,\omega)=\mathscr{I}(\sigma_a(f))(a_1,\ldots,a_n,\omega)$ □

The first two clauses define the usual reduct functor of algebraic specifications [13]: the algebra $\mathscr{U}|_\sigma$ is obtained from $\mathscr{U}$ by forgetting the interpretation of the additional parameters. The last one tells us how the interpretations of the attribute symbols in the reduct are to be computed from the given ones. In the case of inclusion, this condition says that the interpretations are kept unchanged.

These definitions are such that the truth of formulae is preserved under translation when we map interpretations to their reducts. That is, given a θ_2-interpretation structure $\mathbb{S}$, a θ_1-formula f is valid in $\mathbb{S}|_\sigma$ iff $\sigma(f)$ is valid in $\mathbb{S}$. This result is known as the satisfaction condition [27], and ensures that if a specification over θ_2 contains the translations of the axioms of a specification over θ_1 as theorems (in particular as axioms), then all models of the θ_2-theory will be mapped to models of the θ_1-specification. However, the same results do not hold if we take θ_2-loci instead of θ_2-interpretation structures as models because the reduct of a θ_2-locus is not necessarily a θ_1-locus. That is, locality is not necessarily preserved under reducts: an event that is generated with respect to θ_2 is not necessarily generated with respect to θ_1, and it is possible that there are no axioms in the θ_2-specification that require that such events do not interfere with the attributes inherited from θ_1. In our example, this is reflected by the fact that the axioms of `stock` do not imply that the events that are not generated with respect to `product` (ie, that are neither upgrades nor price updates) do not interfere with the attributes of `product` (ie, with `version` and `price`), and mutatis mutandis for `order-service`. This is why it is not sufficient to include all the axioms of the specifications we are inheriting from as axioms of the new specification.

Hence, our notion of "structure preservation" for object specifications will have to require additionally that for every θ_2-locus $\mathbb{O}$, its reduct $\mathbb{O}|_\sigma$ be a θ_1-locus. Naturally, we would like that this requirement be among the axioms of the θ_2-specification. For that purpose, we extend the language by including signature morphisms as formulae, much the same way as proposed in [27] to express data constraints:

Definition 4.2.3: For every pair of object signatures θ and θ', and morphism σ between them, we include $\theta' \stackrel{\sigma}{\Rightarrow} \theta$ as a θ-formula. A θ-interpretation structure makes $\theta' \stackrel{\sigma}{\Rightarrow} \theta$ true iff its σ-reduct is a θ'-locus. □

We can now define the set of axioms associated with the inheriting specification:

$$\Phi(\texttt{name}) = \Phi(\texttt{name}_1) \cup \ldots \cup \Phi(\texttt{name}_n) \cup \Phi(\text{data}) \cup V \cup S \cup L \cup \\ \cup \{\theta(\texttt{name}_1) \stackrel{\text{id}}{\rightarrow} \theta(\texttt{name}), \ldots, \theta(\texttt{name}_n) \stackrel{\text{id}}{\rightarrow} \theta(\texttt{name})\}$$

where by id we are denoting inclusions. Hence, we have as axioms more than the union of the inherited axioms and the new ones: we have in addition the inclusion morphisms as formulae stating that the objects that we are inheriting from are to be preserved. This is a particular case of a morphism between theory presentations:

Theorem 4.2.4: A morphism σ between two theory presentations (θ_1, F_1) and (θ_2, F_2) is a signature morphism $\sigma: \theta_1 \rightarrow \theta_2$ such that $(F_2 \Rightarrow_{\theta_2} \sigma(f))$ is valid for every $f \in F_1$, and $(F_2 \Rightarrow_{\theta_2} \theta_1 \stackrel{\sigma}{\Rightarrow} \theta_2)$ is also valid. □

We should point out that the inclusion of the signature morphisms as axioms allows us to dispense with explicit valuation axioms such as

```
[replenishment(N)]price = price
```

requiring that the new events do not interfere with the old attributes. In fact, the signature morphisms are weaker than the explicit non-interference valuation formulae in the sense that they only require non-interference when an event generated according to the new symbol does not interpret an event symbol inherited from the component specification as well. Hence, by including the signature formula instead of the valuation axiom we are allowing for refinements of our specification where replenishments coincide with price increases (a vaguely familiar policy, alas). In the presence of the valuation axiom, such a refinement would be inconsistent. However, we should point out that a refinement that would make replenishments decrease prices will be inconsistent in our setting as price decreases are not supported by the component specification (`product`) that encapsulates prices. This is the view that is consistent with the encapsulation of data: locality preservation means that we are respecting the fact that we had already identified all the events that were allowed to manipulate the attributes declared in the specification; inheriting from such a specification allows for adding more attributes and their respective events, but requires that the events that are added either behave like the inherited ones, or do not interfere with the inherited attributes.

Hence, from the specification point of view, it is very important to record that a morphism exists between two specifications. Therefore, the inheritance construct will also enrich the

context diagram with edges labelled by morphisms recording that the specification that we have built by inheritance is linked via morphisms to the specifications we have inherited from. Naturally, in the case of inheritance, the labels of the edges are inclusion morphisms. So, besides

```
⟦object name
  inheriting
    from name1 through m1
    ...
    from namen through mn
  importing data types data
  events Γ
  attributes A
  valuation V
  safety S
  liveness L⟧ 𝒩odes = 𝒩odes ∪ (name,(θ(name),Φ(name)))
```

where $\theta(\mathsf{name})$ and $\Phi(\mathsf{name})$ were define above, we shall have

```
⟦object name
  inheriting
    from name1 through m1
    ...
    from namen through mn
  importing data types data
  events Γ
  attributes A
  valuation V
  safety S
  liveness L⟧ ℰdges(namei,name) = (mi,(μ(mi): θ(namei) → θ(name)))
```

where each $\mu(m_i)$ is an inclusion morphism. The edges for pairs other than those connecting the given nodes to the new one are kept unchanged.

4.3. Reasoning from inheritance

The advantage of keeping a record of the structure of a specification (given in terms of the morphisms) is that this structure may be used when reasoning about its properties. Indeed, morphisms have an associated property preservation result:

Proposition 4.3.1: Let σ be a morphism between (θ_1,F_1) and (θ_2,F_2). Then, for every formula f of the positional language, if $(F_1 \Rightarrow_{\theta_1} f)$ is valid, so is $(F_2 \Rightarrow_{\theta_2} \sigma(f))$. Also, for every formula f of the temporal language, if $(F_1 \overset{n}{\Rightarrow}_{\theta_1} f)$ is valid, so is $(F_2 \overset{n}{\Rightarrow}_{\theta_2} \sigma(f))$. □

That is to say, we can export all properties of a specification along a morphism. As a consequence, specification by inheritance as defined above ensures that we inherit all the

properties of the object. From a proof-theoretic point of view, this says that we can use all the theorems proved within the initial specification as "lemmas" for the proof of theorems of the target specification. Hence, if the specification under `name` was built by inheriting from $\texttt{name}_1,\ldots, \texttt{name}_n$, we shall have for every formula f_i of the component specification $\texttt{name}_i$,

$$\frac{\texttt{name}_i \Rightarrow f_i}{\texttt{name} \Rightarrow f_i}$$

and for temporal formulae:

$$\frac{\texttt{name}_i \overset{n}{\Rightarrow} f_i}{\texttt{name} \overset{n}{\Rightarrow} f_i}$$

For instance, we shall have the following inference rules of monotonicity for any positional formula f over $\theta(\texttt{product})$:

$$\frac{\texttt{product} \Rightarrow f}{\texttt{stock} \Rightarrow f}$$

and for temporal formulae:

$$\frac{\texttt{product} \overset{n}{\Rightarrow} f}{\texttt{stock} \overset{n}{\Rightarrow} f}$$

And also, mutatis mutandis, for `order-service`. For instance, because we have proved from the axioms of `product` that prices do not decrease, we are allowed to conclude from the second rule that this is also a property of the normative behaviour of stocks. It is in this sense that we conceive inheritance: we inherit all the properties of the object. But we do not rule out the possibility of adding more properties.

Because the new specification returns a signature and a collection of axioms, we are also able to apply the inference rules of the logic to derive its properties. For instance, it is easy to prove that

$$\texttt{stock} \overset{n}{\Rightarrow} (\text{zero} \leq \text{qoh})$$

is valid, ie that "the quantity-on-hand is non-negative" is a property of every normative state of a stock. The rule that allows us to derive this property is quite similar to the one we used in section 3.3 to derive that prices never decrease:

$$\frac{\begin{array}{c}\texttt{stock} \Rightarrow ([](\text{zero}\leq\text{qoh}))\\ \texttt{stock} \Rightarrow (\texttt{Per}(\text{set-price(N)}), (\text{zero}\leq\text{qoh}) \rightarrow [\text{set-price(N)}](\text{zero}\leq\text{qoh}))\\ \texttt{stock} \Rightarrow (\texttt{Per}(\text{upgrade(N)}), (\text{zero}\leq\text{qoh}) \rightarrow [\text{upgrade(N)}](\text{zero}\leq\text{qoh}))\\ \texttt{stock} \Rightarrow (\texttt{Per}(\text{deliver(O)}), (\text{zero}\leq\text{qoh}) \rightarrow [\text{deliver(O)}](\text{zero}\leq\text{qoh}))\\ \texttt{stock} \Rightarrow (\texttt{Per}(\text{arrival(O)}), (\text{zero}\leq\text{qoh}) \rightarrow [\text{arrival(O)}](\text{zero}\leq\text{qoh}))\\ \texttt{stock} \Rightarrow (\texttt{Per}(\text{replenish(N)}), (\text{zero}\leq\text{qoh}) \rightarrow [\text{replenish(N)}](\text{zero}\leq\text{qoh}))\end{array}}{\texttt{stock} \overset{\text{n}}{\Rightarrow} (\text{zero}\leq\text{qoh})}$$

As usual for static safety properties (ie, properties that involve only one state), we have an induction rule: we prove that the property holds initially (first premiss), and that it is invariant under arbitrary permitted events. Locality further allows us to prove this invariance property by considering only events generated from the event symbols present in the signature. Hence, we have a premiss for each of the event symbols present in the signature (including, obviously, those inherited from the component specifications). The proofs of the antecedants of the rule are straightforward. We shall only prove the case of deliveries:

1. $(\text{req(O)}\leq\text{qoh}) \rightarrow (\text{zero}\leq\text{qoh-req(O)})$	(theorem of integers)
2. $[\text{deliver(O)}]\text{qoh} = \text{qoh-req(O)}$	(V3)
3. $(\text{req(O)}\leq\text{qoh}) \rightarrow (\text{zero}\leq[\text{deliver(O)}]\text{qoh})$	1, 2, substitution
4. $\texttt{Per}(\text{deliver(O)}) \rightarrow (\text{req(O)}\leq\text{qoh})$	(S)
5. $\texttt{Per}(\text{deliver(O)}) \rightarrow (\text{zero}\leq[\text{deliver(O)}]\text{qoh})$	3, 4, $\rightarrow$
6. $\texttt{Per}(\text{deliver(O)}), (\text{zero}\leq\text{qoh}) \rightarrow (\text{zero}\leq[\text{deliver(O)}]\text{qoh})$	5, monotonicity
7. $\texttt{Per}(\text{deliver(O)}), (\text{zero}\leq\text{qoh}) \rightarrow [\text{deliver(O)}](\text{zero}\leq\text{qoh})$	6, zero and $\leq$ being rigid

This example also allows us to illustrate the derivation of liveness properties. Such properties require that "something good will eventually happen", eg "a pending order will eventually be delivered". Naturally, the satisfaction of liveness properties must be checked only for normative behaviours. The instantiation of the general rule that allows us to derive such properties (see [16]) is, for the case above,

$$\frac{\begin{array}{c}\texttt{order-service} \Rightarrow (\text{pending(O)=true} \rightarrow \texttt{Obl}(\text{deliver(O)})\\ \texttt{order-service} \Rightarrow ([\text{deliver(O)}](\text{pending(O)=false}))\end{array}}{\texttt{order-service} \overset{\text{n}}{\Rightarrow} (\text{pending(O)=true} \rightarrow \mathbf{F}(\text{pending(O)=false}))}$$

That is to say, if we know that in certain conditions an event is obligatory, and that that event establishes some property, then we can infer that, under those conditions, the property will eventually hold. (Recall that **F** is the temporal operator "sometime in the future"). These rules provide us with the "semantics" of obligation in the sense that they show how the obligation predicate can be used. Because the antecedants are theorems of `order-service`, the consequent expresses that the required liveness condition is a property of all normative behaviours of order-services. Furthermore, from the monotonicity rules linking `order-`

`service` and `stock`, we know that this is also a liveness property of the normative behaviour of stocks. That is, we may infer

$$\texttt{stock} \stackrel{\text{n}}{\Rightarrow} (\text{pending(O)=true} \rightarrow \mathbf{F}(\text{pending(O)=false}))$$

4.4. Particularisation

The previous section emphasised the fact that our mechanism of inheritance is such that the locality requirement associated with the specifications we are inheriting from is to be preserved. This means that we regard the specification we are inheriting from as having identified the set of events that are responsible for manipulating the attributes that were declared. However, in some situations, we may want to revise that specification and extend it in such a way that the locality requirement is violated, namely by adding new events that update the "old attributes". In this case, the inheritance construction is not adequate because it will return an inconsistent specification. Hence, we shall consider another construction, specification by particularisation, and show how it differs from inheritance as defined above. The syntax of a particularisation is the following

```
particularisation = object name particularisation of spec by body
```

Consider, for instance, the specification of fashionable products (those that are bound to be out of fashion and, hence, prone to be sold out as bargains after Christmas):

```
object fashionable-product
  particularisation of product by
  events
    sale
  valuation
    [sale]price = 50%price
end
```

The signature of the resulting specification is defined as above for inheritance. The difference lies in the axioms of the specification: they will not include the requirement on locality preservation, ie the signature morphism will not be included as an axiom. Hence, we shall have

$$\Phi(\texttt{fashionable-product}) = \Phi(\texttt{product}) \cup \{\texttt{[sale]price = 50\%price}\}$$

Likewise, the effects of particularisation on the context are similar to those of inheritance except that, this time, we shall not modify the morphism component because the specification that we have built does not preserve the structure of the ones we started from. Hence, we shall have

```
⟦object name particularisation of spec by
  importing data types data
  events Γ
  attributes A
  valuation V
  safety S
  liveness L⟧ Edges = Edges
```

That is to say, no new edges are added.

From a proof theoretic point of view, we are only able to import to the new theory the axioms of the previous one, not its theorems. That is to say, we do not inherit all the properties that were proved for the original specification. This is because their proof possibly depended on the locality principle, which is not necessarily preserved under particularisation. For instance, the property that "prices never decrease" is no longer valid for fashionable products. Indeed, we shall not be able to prove

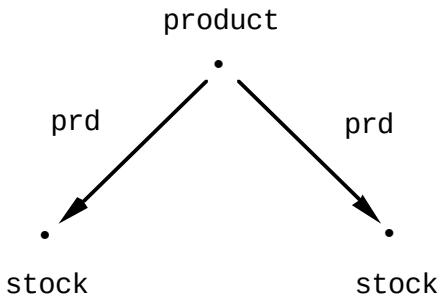
$$\texttt{fashionable-product} \overset{\Box}{\Rightarrow} (\text{price} \leq \mathbf{X}\text{price})$$

Hence, whereas inheritance gives us a mechanism by which we are able to "import" all the properties of a specification, and hence "use" the whole object, particularisation allows us only to use the code of the given specification. Nevertheless, particularisation is a useful mechanism to apply to objects that are still "open" in the sense that they are not yet ready to be assigned a locality. This may be due to the fact that not all the events that are meaningful with respect to the attributes have been identified. Hence, our proof-theory will not allow for exporting properties of these objects before they are "closed".

5. AGGREGATION

The last construct that we shall analyse, aggregation, provides us with a mechanism for building a complex specification by picking up a collection of specifications already defined (ie, stored in the context), organising them by saying how they are related to each other (through morphisms), and providing the names by which they will be identified in the complex specification.

For instance, now that we have specifications of products and stocks, assume that we want to specify a system that deals with two stocks (north and south) of the same product. We want to be able to pick-up two copies of the stock specification and say that they share the same copy of the product specification, ie that we have only one product involved. This configuration can be given through the diagram



whose nodes are labelled by names of specifications imported from our context (ie, the node labels belong to $\mathcal{N}odes$), and whose edges are labelled by names of morphisms also imported from the context (ie, the edge labels belong to $\mathcal{E}dges$). Naturally, if an edge of the configuration diagram is between two nodes labelled n and n', the label of the edge should belong to $\mathcal{E}dges$(n,n'). Finally, we want a mechanism that returns a specification out of that diagram, and that says no more and no less than the diagram, so that we may manipulate it as just another specification (eg, inherit from it).

This mechanism can be formalised in category theory through colimits. Intuitively, a colimit is an operation that, given a diagram, returns the least object that has the "same information" as the diagram. In a way, it computes a "least upper bound" of the objects that we want to aggregate. It contains all the information of its components, and no more than what is necessary to glue them together. This glue is provided by the arrows of the diagram. Hence, before detailing the semantics of aggregation, we shall give a brief introduction to what is involved in a colimit.

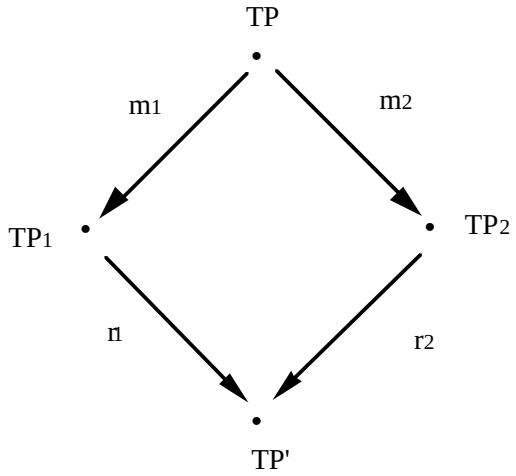
5.1. Background

From a configuration diagram as the one depicted above, we obtain a diagram in the category $\mathcal{THE}$ of theory presentations by replacing each node label n by π(n) (ie, by its label in the context) and each edge label e by μ(e). Then, we are able to compute the colimit of this diagram. This construction "completes" the diagram by adding a new node labelled by some theory presentation and an edge labelled by some morphism linking each node of the original diagram to the new node. The new morphisms are such that the diagram commutes, ie by composing the morphisms along two paths from some given node to the new node we obtain the same morphism. This means that the theory presentation that we obtain preserves the structure that was given through the original diagram. The colimit construction further ensures that the theory presentation that we obtain is the "minimal" one that "contains" the given ones as components in the sense that it discriminates as much as possible (identifies as little as possible) between the components. In our case, this means that we shall obtain different copies of the parameters of stocks except for those that were inherited from product

of which only one copy is provided because the arrows indicate that these parameters are to be shared by the two stocks. In particular, different copies of order-service will be obtained meaning that, a priori, each stock processes its own orders.

We shall not be able to detail the construction of a colimit. See instead [2]. However, it is worthwhile to provide some intuitions on what is involved. For this purpose, we shall illustrate a special case, pushouts, that correspond to diagrams that have the shape of the one above. In fact, a general colimit construction can be obtained by performing a sequence of pushouts, so that we shall be illustrating the core of the construction (so to speak).

Hence, assume that we are given three theory presentations TP, TP_1 and TP_2 together with two morphisms m_1 and m_2 between TP and, respectively, TP_1 and TP_2. As we explained above, we shall obtain a theory presentation TP' and morphisms r, r_1 and r_2 from TP, TP_1 and TP_2. Because these morphisms will satisfy the condition $r=m_1;r_1=m_2;r_2$ (expressing the commutativity of the diagram as mentioned above), we shall be interested only in r_1 and r_2.



The axioms of the resulting theory presentation are completely determined by the signatures and the morphisms. Indeed, we shall have

$$\Phi(TP')=r_1(\Phi(TP_1))\cup r_2(\Phi(TP_2))\cup\{\theta(TP_1)\xrightarrow{r_1}\theta(TP'),\theta(TP_2)\xrightarrow{r_2}\theta(TP')\}$$

That is to say, the new theory presentation will have as axioms the translations of the axioms of the component theory presentations together with the the signature morphisms expressing that the locality associated with the components is to be preserved.

The signature of TP' is obtained from the disjoint union of the signatures $\theta(TP_1)$ and $\theta(TP_2)$ by identifying the symbols in those signatures that are imported from $\theta(TP)$ via the morphisms m_1 and m_2. This construction is known as an amalgamated sum. Naturally, it remains to specify exactly which symbols will be in $\theta(TP)$ and, thus, define the new morphisms. (The pushout construction is only defined up to isomorphism). These names

will have to be provided somehow by the specifier. We shall adopt a convention that will use the so called "dot notation". We shall ask the specifier to provide a name for each node of the diagram. This name will also be used as an edge of the resulting context linking the given node and the new node (ie, as the name of the resulting morphism). The naming convention will be as follows: assuming that `name`, $\texttt{name}_1$ and $\texttt{name}_2$ are assigned, respectively, to TP, TP_1 and TP_2, a symbol `s` in TP_i will be named $\texttt{name}_i\texttt{.s}$ in TP' except when the symbol was imported via m_i from TP in which case it will be named `name.s`. We shall also assume that, in case we want to keep the same symbols (ie, define inclusion morphisms), we indicate it by using "`nil`". For instance, assuming that the nodes labelled by stock are given the names `north` and `south`, and that the node labelled product is named `nil`, the resulting signature will have the following attribute symbols:

```
price: nat
version: nat
north.pending(north.ord): bool
south.pending(south.ord): bool
north.qoh: int
south.qoh: int
```

Events symbols would be named in the same way. Naturally, the same principles will apply to the data types. Above, we were assuming that the sort symbols `nat` and `bool` were to be shared between the specifications (resulting in only one copy), whereas the sort symbol `ord` was to be duplicated, meaning that each stock has its own identification for the orders it receives.

5.2. The aggregation construct

From the example above, it is easy to see that, as input to the aggregation construct, we have to provide a diagram and names for renaming the symbols of the constituent specifications. Our syntax will be as follows:

```
aggregation = object id aggregating diagram end
diagram = list-nodes list-edges
node = node natural label id name id
edge = edge natural '→' natural label id
```

That is to say, we shall use natural numbers to define the intended diagram. For instance, the example above would be specified as follows

```
object 2-stocks-of-1-product
  aggregating
    node 0 label product name nil
    node 1 label stock name north
    node 2 label stock name south
    edge 0→1 label prd
    edge 0→2 label prd
end
```

Naturally, a full specification support system would include appropriate graphical notations for the definition of the diagrams. As we have already said, our goal in this paper is not to provide a working specification language, but to show how some basic constructs like aggregation can be given a proof-theoretic semantics.

The effects of this construct on the context will thus be given by

$$[\![\textbf{object}\ \mathsf{name}\ \textbf{aggregating}\ \textbf{node}\ \mathsf{p}_1\ \textbf{label}\ \mathsf{l}_1\ \textbf{name}\ \mathsf{m}_1\ \ldots\ \textbf{node}\ \mathsf{p}_n\ \textbf{label}\ \mathsf{l}_n\ \textit{name}\ \mathsf{m}_n\ \textbf{edge}\ \mathsf{p}_{i_1}\!\rightarrow\!\mathsf{p}_{j_1}\ \textbf{label}\ \mathsf{e}_1\ \ldots\ \textbf{edge}\ \mathsf{p}_{i_m}\!\rightarrow\!\mathsf{p}_{j_m}\ \textbf{label}\ \mathsf{e}_m]\!]\ \mathcal{N}odes = \mathcal{N}odes \cup (\mathsf{name},(\theta(\mathsf{name}),\Phi(\mathsf{name})))$$

where $\theta(\mathsf{name})$ is the signature that results from the colimit construction using the above mentioned naming convention (using the identifiers m_1, …, m_n and the dot notation) and $\Phi(\mathsf{name})$ is the set of formulae that is obtained by taking the union of the axioms of the specifications involved, after having been suitably translated along the naming conventions, together with the signature morphisms as axioms requiring that the locality of the components involved be preserved. That is to say, we shall have

$$\Phi(\mathsf{name})=r_1(\Phi(l_1))\cup\ldots\cup r_n(\Phi(l_n))\cup\{\theta(l_1)\xrightarrow{r_1}\theta(\mathsf{name}),\ldots,\theta(l_n)\xrightarrow{r_n}\theta(\mathsf{name})\}$$

where $r_1,\ldots,r_n$ are the signature morphisms that result from the colimit construction (ie, the renamings that map each symbol of the constituent signatures to the symbol to which it corresponds in the new signature).

Notice that the resulting set of axioms does not include valuation formulae requiring non-interference between the north and south stocks, specifying the (lack of) effects of events from the south stock on the attributes of the north stock and vice versa, as in

```
[north.replenishment(N)]south.qoh = south.qoh
```

This would seem to be intuitively necessary, or left to some kind of frame rule [28]. However, this is the rôle of the signature morphisms included as axioms of the aggregation:

the locality preservation requirement expressed through the morphisms guarantees that each event from the south stock either occurs in concurrency with an event from the north stock (ie, the north and south symbol is given the same interpretation), or does not interfere with the attributes of the north stock, and vice versa. However, as we have seen for inheritance, notice that the requirement expressed via the signature morphism is weaker than the non-interference valuation axiom. Indeed, the signature morphism leaves open the possibility of having event symbols inherited from different specifications to generate the same events (recall that we are not working with initial models of the universe signature) and, hence, allows for further refinements of our specifications. For instance, we are allowing for replenishments of the north and the south stocks to coincide. Although we are not imposing it, we might decide later on to require it explicitly in order to enforce some new policy. The general principle is that event symbols inherited from different specifications are usually unrelated and, hence, they should be allowed the maximum degree of freedom in terms of "occurring simultaneously" (generating the same events). In fact, our model of parallelism is not of pure interleaving, but allows for "concurrency" of non-interfering events. The non-interference formulae would prevent this by discriminating the events generated from different event symbols in terms of their effects on the attributes, ie they would make them observationally different. This semantics of aggregation is also more akin to the notion of minimal combination of the component objects: enforcing interleaving seems to be a design decision imposed on their joint behaviour and, hence, it should not be inherent to the aggregation construct.

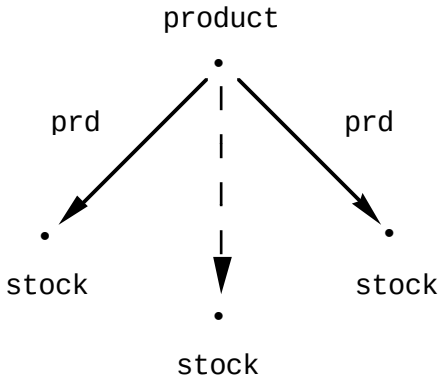
Hence, basically, the signature morphisms identify sub-components of an object that should not interfere apart from the specified interfaces. In our case, product would be that interface: for instance, according to our specification, the north and south stocks will synchronise when a new version of the product is released. In terms of implementation, and assuming a distributed configuration, this means that the release of a new version will be communicated simultaneously and with the same information to both stocks.

Naturally, this construct will also enrich the edges of the context, recording the new morphisms that result from the colimit of the diagram:

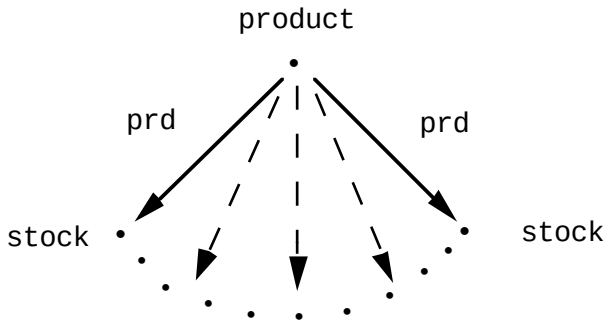
$$
\begin{array}{l}
[\![\ \textbf{object}\ \mathsf{name} \\
\quad \textbf{aggregating} \\
\qquad \textbf{node}\ p_1\ \textbf{label}\ l_1\ \textbf{name}\ m_1 \\
\qquad \ldots \\
\qquad \textbf{node}\ p_n\ \textbf{label}\ l_n\ \textbf{name}\ m_n \\
\qquad \textbf{edge}\ p_{i_1} \rightarrow p_{j_1}\ \textbf{label}\ e_1 \\
\qquad \ldots \\
\qquad \textbf{edge}\ p_{i_m} \rightarrow p_{j_m}\ \textbf{label}\ e_m\ ]\!] \ \mathcal{E}dges(l_i, \mathsf{name}) = (m_i, (\mu(m_i): \theta(l_i) \rightarrow \theta(\mathsf{name})))
\end{array}
$$

where each $\mu(m_i)$ is the morphism r_i that results from the colimit construction.

Aggregation thus provides a mechanism for the progressive and constructive build-up of our system by incorporating new components. This is the basis for achieving the desired degree of modularity in design. Revisions and extensions to the specification may be done by aggregating new components. For instance, we may extend our specification by aggregating another stock for the same product:



In this case, we would have to give another name to the new stock. And we can extend with as many components as we want. But, instead of extending the specification in this piecewise fashion, we might envisage parameterising the relationship between products and stocks by assigning a "class" of stocks to a product, corresponding to a "big diagram":



From the specification point of view, we would have to provide a sort symbol (from some data type) with which we want to parameterise the nodes labelled by `stock` in the diagram. This sort can be seen to provide the names of the stocks for the given product. We would then have

```
object stocks-of-a-product
  aggregating
    node 0 label product name nil
    node 1 label stock parameterised by s
    edge 0→1 label s.prd
end
```

corresponding to the "parameterised" diagram above. These are not "standard" constructions, as `S` is to be taken as a symbol and not as a set, and are formalised in [16]. Hence, we shall

not expand further on this issue, but we may add that the resulting specification (a particular kind of colimit) will end up with the symbols conveniently parameterised by the sort of names, eg we shall have

```
price: nat
version: nat
pending(s,ord): bool
qoh(s): int
```

as attribute symbols where `s` is the sort of stock names for the given product. This construction would have also been applied to define `order-service` from a specification `order` of orders, using the sort `ord` as parameter. This corresponds to the type mechanism defined in [10,12], where a type specification provides an object specification (the template) and a naming mechanism for instances of the type.

Aggregation and inheritance also often work together. Indeed, it is easy to see that multiple inheritance can be obtained from single inheritance and aggregation by aggregating first the specifications we want to inherit from, and then inheriting from their aggregation. The particular case of aggregation that we have used for inheritance did not allow for renaming: it was the simple case of taking the "union" of the given specifications. Hence, aggregation as defined above provides us with more flexible mechanisms for supporting inheritance, leading to a two step construction of complex objects: aggregation of the components and addition of detail.

However, it seems important to point out that, whereas aggregation merely "puts objects together" through selected interfaces and, thus, does not create any additional state information of its own, ie its state is just the composition of the given states, inheritance in general is used when we want to create additional state information which may use the given one, but which cannot be reduced to it. Hence, whereas the implementation of an aggregation needs just the implementation of its parts (plus the synchronisation between events), the implementation of a specification built by inheriting from a given one further requires an additional implementation effort.

5.3. Reasoning over aggregations

As we have seen for inheritance, the existence of morphisms between the theory presentations associated with the components and their aggregation implies that the properties of the components and the properties of their aggregation are related. Indeed, given a construction

```
object name
  aggregating
    node p1 label l1 name m1
    ...
    node pn label ln name mn
    edge pi1→pj1 label e1
    ...
    edge pim→pjm label em
end
```

we have for every formula f_i of the component specification l_i,

$$\frac{l_i \Rightarrow f_i}{\texttt{name} \Rightarrow \mu(m_i)(f_i)}$$

and for temporal formulae:

$$\frac{l_i \overset{n}{\Rightarrow} f_i}{\texttt{name} \overset{n}{\Rightarrow} \mu(m_i)(f_i)}$$

That is to say, we may export the properties of the components to their aggregation using the translations defined by the morphisms. For instance, we shall have the rule

$$\frac{\texttt{stock} \overset{n}{\Rightarrow} \texttt{(zero}\leq\texttt{qoh)}}{\texttt{2-stocks-of-1-product} \overset{n}{\Rightarrow} \texttt{(zero}\leq\texttt{north.qoh)}}$$

and mutatis mutandis for south stocks. Since the premiss was already proved (cf 4.3), we are allowed to infer the conclusion, ie that the quantity on hand in the north stock is always non-negative during normative behaviour.

Besides importing properties of the component specifications into their aggregation, we may also derive new properties that result from the specified interaction between the components. Among the properties that are usually of concern for aggregation are, of course, the possibility of deadlock or starvation, or mutual exclusion in the access to shared resources. In the case above, the components scarcely interact and, hence, there are no significant new properties to prove. See [16,49] for examples.

This modularity of the proof system also means that we do not need the whole configuration of the system to reason about it. Instead, we may select the relevant part of the system and apply our rules of inference to that fragment. The structural properties of our logic make sure that we are only allowed to derive properties that will still be valid for the whole system. This is an essential feature for a workable calculus of specification, as it is impossible to reason over large specifications as a whole: we must be able to use their structure.

6. CONCLUDING REMARKS

A formal counterpart to the notion of object as the basic building block for information systems design was presented. We have seen that it is possible to define a logic such that its category of theory presentations is able to provide the necessary mechanisms for formalising horizontal composition in object-oriented design. On the one hand, by integrating positional, deontic and temporal flavours, the logic is well suited to provide the required ability to capture structural and behavioural phenomena in a unique specification primitive. On the other hand, the notion of locality (encapsulation) was formalised around the signatures of the logic and incorporated into the logic by defining morphisms of theory-presentations to be locality-preserving. This means that locality was formalised as a "structural" notion of the logic, making it the criterion for recognising components of a complex system and, hence, the notion of modularity to be supported by the logic.

In the proposed framework, objects may either be specified directly as theory presentations, or they may be built from previously specified objects through inheritance or aggregation. Inheritance allowed us to extend given specifications by incorporating more state information but respecting the locality of the objects it inherits from. In this respect, it was compared with a mechanism of pure particularisation. Aggregation was presented as a means for specifying complex objects by defining a diagram of specifications where the edges specify the interaction between the objects by saying which components are to be shared among which objects. Other essential constructs for object-oriented specification such as class/type grouping and parameterisation were not yet addressed, but we do not see deep problems in formalising them in the proposed framework. Work in the direction of defining type mechanisms is reported in [16]. Research will proceed in the near future in the direction of parameterisation as this is another essential feature for assisting design, namely for addressing reusability.

Although it was not possible to provide the full proof theory that is being developed for this logic, we were able to show how the proposed proof-theoretic approach favours the use of the structure of the specifications for deriving properties. The work reported in [44] clearly demonstrates the usefulness and viability of using the structure of a theory in directing the search for proofs. It is hoped that the explicit recording of the structure of a specification in its axioms through the signature morphisms may provide effective support to theorem-proving techniques. Besides this aspect, we illustrated the derivation of classical safety (both static and dynamic) and liveness properties. More examples may be found in [16].

Finally, work will also proceed in the direction of formalising refinement (vertical composition) in this framework. Results on a model-theoretic account of refinement for

objects were already reported in [9]. This formalisation effort will allow us to assist the process of mapping the abstract objects specified during conceptual modelling to the intended implementation layers. As already mentioned, each implementation step is intended to consist of assembling concrete objects (which will ultimately be provided by the programming-database environment together with the underlying operating system) into a configuration that provides the functionality that is necessary to accomodate the requirements expressed at the abstract level.

ACKNOWLEDGMENTS

The authors wish to thank their colleagues A.Sernadas, H.-D.Ehrich and U.Lipeck for many stimulating discussions. This work was partially supported by the ESPRIT Basic Research Action nº 3023 (IS-CORE).

REFERENCES

[1] P.America, "Object-Oriented Programming: a Theoretician's Introduction", *EATCS Bulletin* 29, 1986, 69-84

[2] M.Arbib and E.Manes, *Arrows, Structures and Functors*, Academic Press 1975

[3] M.Atkinson, F.Bancilhon, D. DeWitt, K. Dittrich, D. Maier and S. Zdonik, "The Object-oriented Database System Manifesto", in W. Kim, J.-M. Nicolas and S. Nishio (eds) *First International Conference on Deductive and Object-oriented Databases*, 1989, 40-57

[4] C.Atkinson, S.Goldsack, A.DiMaio and R.Bayan, "Object Oriented Concurrency and Distribution in DRAGOON", *Journal of Object Oriented Programming* (in print)

[5] K.Bruce and P.Wegner, "An Algebraic Model of Subtypes in Object-Oriented Languages", *SIGPLAN Notices* 21(10), ACM 1986, 163-172

[6] R.Burstall and J.Goguen, "The Semantics of Clear, a Specification Language", in D. Bjorner (ed) *Proc 1979 Winter School on Abstract Software Specification*, LNCS 86, Springer-Verlag 1980, 292-332

[7] U.Dayal and K.Dittrich (eds), *Proc. of the International Workshop on Object-oriented Database Systems*, Los Angeles, IEEE Computer Society, 1986

[8] K.Dittrich (ed), *Advances in Object-oriented Database Systems*, LNCS 334, Springer Verlag, 1988

[9] H.-D.Ehrich and A.Sernadas, "Algebraic View of Implementing Objects over Objects", in W.deRoever (ed) *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Springer-Verlag (in print)

[10] H.-D.Ehrich, A.Sernadas and C.Sernadas, "Objects, Object Types and Object Identity", in H.Ehrig et al (eds) *Categorical Methods in Computer Science with Aspects from Topology*, LNCS 393, Springer-Verlag (in print)

[11] H.-D.Ehrich, A.Sernadas and C.Sernadas, "Abstract Object Types for Databases" (extended position paper), in [Dittrich 88], 144-149

[12] H.-D.Ehrich, A.Sernadas and C.Sernadas, "From Data Types to Object Types", *Journal of Information Processing and Cybernetics* EIK 26(1/2), 1990, 33-48

[13] H.Ehrig and B.Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Springer-Verlag 1985

[14] H.B.Enderton, *A Mathematical Introduction to Logic*, Academic Press 1972

[15] J.Fiadeiro and T.Maibaum, "Temporal Reasoning over Deontic Specifications", *Journal of Logic and Computation*, to appear

[16] J.Fiadeiro and T.Maibaum, *Towards Object Calculi*, Technical Report, Imperial College, London 1990

[17] J.Fiadeiro and A.Sernadas, "The Infolog Linear Tense Propositional Logic of Events and Transactions", *Information Systems* 11, 1986, 61-85

[18] J.Fiadeiro and A.Sernadas, "Specification and Verification of Database Dynamics", *Acta Informatica* 25, 1988, 625-661

[19] J.Fiadeiro and A.Sernadas, "Structuring Theories on Consequence", in D.Sannella and A.Tarlecki (eds) *Recent Trends in Data Type Specification*, LNCS 332, Springer Verlag 1988, 44-72

[20] J.Fiadeiro and A.Sernadas, "Logics of Modal Terms for Systems Specification", *Journal of Logic and Computation*, to appear

[21] J.Fiadeiro, A.Sernadas and C.Sernadas, "Knowledge Bases as Structured Theories", in K.Nori and S.Kumar (eds) *Foundations of Software Technology and Theoretical Computer Science*, LNCS 338, Springer-Verlag 1988, 469-486

[22] H.Gallaire, J.Minker and J.-M.Nicolas, "Logic Databases: a Deductive Approach", *Computing Surveys* 16(2), 1984, 153-185

[23] J.Goguen, "Objects", *Int. Journal of General Systems* 1, 1975, 237-243

[24] J.Goguen, "Reusing and Interconnecting Software Components", *IEEE Computer* 19(2), 1986, 16-28

[25] J.Goguen, *A Categorical Manifesto*, Technical Report PRG-72, Programming Research Group, University of Oxford, March 1989

[26] J.Goguen, J.Thatcher and E.Wagner, "An Initial Algebraic Approach to the Specification, Correctness, and Implementation of Abstract Data Types", in R.Yeh (ed) *Current Trends in Programming Methodology*, Vol 4, Prentice-Hall 1978, 80-149

[27] J.Goguen and R.Burstall, "Introducing Institutions", in E.Clarke and D.Kozen (eds) *Proc Logics of Programming Workshop*, LNCS 164, Springer-Verlag 1984, 221-256

[28] J.Goguen and J.Meseguer, "Extensions and Foundations of Object-Oriented Programming", *SIGPLAN Notices* 21(10), ACM 1986, 153-162

[29] D. Harel, *First-Order Dynamic Logic,* LNCS 68, Springer-Verlag 1979

[30] F.Hayes and D.Coleman, *Objects and Inheritance: An Algebraic View*, Hewlett Packard 1989

[31] S.Kamin, "Inheritance in Smalltalk-80: a Denotational Definition", *ACM TOPLAS* 5(1), 1988

[32] S.Khoshafian and G.Copeland, "Object Identity", *ACM SIGPLAN Notices* 21(11), 1986, 406-416

[33] S.Khosla, T.Maibaum and M.Sadler, "Database Specification", in T.Steel and R.Meersman (eds) *Database Semantics (DS-1),* North-Holland 1986, 141-158

[34] S.Khosla and T.Maibaum, "The Prescription and Description of State-Based Systems", in B.Banieqbal, H.Barringer and A.Pnueli (eds) *Temporal Logic in Specification*, LNCS 398, Springer-Verlag 1989, 243-294

[35] W.Kim and F. Lochovski (eds), *Object-oriented Concepts, Databases and Applications*, ACM Press, Addison-Wesley, 1988

[36] W.Kim, J.-M.Nicholas and S.Nishio, *Proc. 1st International Conference on Object-Oriented Data Bases,* Kyoto 1989

[37] M.Lehman, V.Stenning and W.Turski, "Another Look at Software Design Methodology", *ACM Software Enginnering Notes* 9(2), 1984, 38-53

[38] T.Maibaum and W.Turski, "On What Exactly Goes On When Software Is Developed Step by Step" *Proc. 7th Int. Conference on Software Enginnering*, IEEE 1984, 528-533

[39] Z.Manna and A.Pnueli, "Verification of Concurrent Programs: The Temporal Framework", in R.Boyer and J.Moore (eds) *The Correctness Problem in Computer Science,* Academic Press 1981, 215-273

[40] L.McCarty, "Permissions and Obligations", *IJCAI 83*, 1983, 287-294

[41] M.Minsky and A.Lockman, "Ensuring Integrity by Adding Obligations to Privileges", in *Proc 8th IEEE Int. Conf on Software Engineering,* 1985, 92-102

[42] D.Parnas, "On the criteria to be used in decomposing systems into modules", *Communications ACM* 15, 1972, 1053-1058

[43] A.Pnueli, "The Temporal Logic of Programs", in *Proc 18th Annual Symposium on Foundations of Computer Science*, IEEE 1977, 45-57

[44] D.Sannella and R.Burstall, "Structured Theories in LCF", in G.Ausiello and M.Protasi (eds) *Proc. 8th Colloquium on Trees in Algebra and Programming,* LNCS 159, Springer-Verlag 1983, 377-391

[45] S.Schuman and D.Pitt, "Object-Oriented Subsystem Specification", in L.Meertens (ed) *Program Specification and Transformation*, North-Holland 1987, 313-341

[46] A.Sernadas, "Temporal Aspects of Logical Procedure Definition", *Information Systems* 5, 1980, 167-187

[47] C.Sernadas and G.Saake, *Formal Semantics of Object-oriented Languages for Conceptual Modeling*, submitted for publication

[48] A.Sernadas, C.Sernadas and H.-D.Ehrich, "Object-Oriented Specification of Databases: An Algebraic Approach", in P.Hammersley (ed) *Proc 13th VLDB Conference*, Morgan Kaufmann 1987, 107-116

[49] A.Sernadas, J.Fiadeiro, C.Sernadas and H.-D.Ehrich, "Abstract Object Types: A Temporal Perspective", in B.Banieqbal, H.Barringer and A.Pnueli (eds) *Temporal Logic in Specification*, LNCS 398, Springer-Verlag 1989, 324-349

[50] A.Sernadas, J.Fiadeiro, C.Sernadas and H.-D.Ehrich, "The Basic Building Blocks of Information Systems", in E.Falkenberg and P.Lindgreen (ed) *Information Systems Concepts: An In-depth Analysis*, North-Holland 1989, 225-246

[51] C.Sernadas, J.Fiadeiro, R.Meersman and A.Sernadas, "Proof-Theoretic Conceptual Modelling: The NIAM Case Study", in E.Falkenberg and P.Lindgreen (ed) *Information Systems Concepts: An In-depth Analysis*, North-Holland 1989, 1-30

[52] C.Sernadas, J.Fiadeiro and A.Sernadas, "Modular Construction of Logical Knowledge Bases: an Algebraic Approach", *Information Systems* 15(1), 1990, 37-59

[53] C.Sernadas, J.Fiadeiro and A.Sernadas, "Object-Oriented Conceptual Modeling From Law, in R.Meersman, Z.Chi e C.-H.Kung (eds) *The Role of Artificial Intelligence in Databases and Information Systems*, North-Holland 1990, 305-327

[54] D.Tsichritzis, *Active Object Environments*, Université de Genève, 1988

[55] P.Yelland, "First Steps Towards Fully Abstract Semantics for Object-oriented Languages", in Cook, S. (ed) *Proc. ECOOP'89 Conference*, BCS Series, Cambridge University Press, 1989

[56] R.Wieringa, J.-J.Meyer and H.Weigand, "Specifying Dynamic and Deontic Integrity Constraints", *Data and Knowledge Engineering* 4(2), 1989, 157-190

[57] R.Wieringa, *Algebraic Foundations for Dynamic Conceptual Models*, PhD Thesis, Vrije Universiteit te Amsterdam 1990

[58] S.Zilles, *Algebraic Specification of Data Types*, Project MAC Progress Report 11, MIT 1974, 28-52