Conservative Extensions, Interpretations Between Theories and All That!(*)

TSE Maibaum
Department of Computing
Imperial College
180 Queen's Gate
London SW7 2BZ UK
tsem@doc.ic.ac.uk

Abstract. About twenty years ago, together with a group of collaborators, some conjectures were developed about the fundamental principles of a theory of specification. These principles included the use of interpretations between theories to underpin the concept of representation and parameterisation, conservative extensions to underpin the concept of modularity and extralogical equality to deal with multiple representations. It was quickly realised that there were fundamental metalogical properties which amounted to 'laws' of specification. An example is provided by the role of Craig interpolation in the composability of implementations and parameter instantiation. Further work on institutions added some fundamental ideas about generalising some of these concepts to logics other than many sorted first order logic and pointed out the categorical nature of many of the constructions. Recent work has highlighted the possibility of 'internalising' some of the meta concepts involved and led to a re-examination of the fundamental principles. For example, extralogical equality and general interpretations are not as fundamental as we thought twenty years ago. The purpose of the paper is to present a retrospective on this work and outline the basic principles of a general theory of specification as we now see it.

1 Introduction

It is now about 20 years since I started work on specification. About 10 years ago, I was asked to give an invited talk at IFIP'86 in Dublin and I used it as an opportunity to rehearse the ideas and philosophy underlying the work of myself and my collaborators over that first 10 years. The kind invitation to present this invited talk has given me the opportunity to look back over the last 20 eventful years and critically assess the ideas, philosophy, and technical cornerstones of the approach. Which of these has stood the test of time? Which now appear less fundamental and, perhaps, even been dumped overboard? Which ideas and technical perspectives have had to be added?

My interest in specification began at about the time I first went to visit the Pontifícia Universidade Católica do Rio de Janeiro (PUC/RJ) where there were not too many algebraists around, but there were some clever logicians with a deep interest in Computing (namely, Paulo Veloso and Roberto Lins de Carvalho). We began ([8]) by trying to assess how the ideas about abstract data types which had appeared during the mid to late '70s could be recast in (many sorted) first order logic (FOL). The motivation was simply that FOL was more expressive and that this could not but help the engineer in the specification task. Very early on, we had developed an abiding interest in understanding what specification was for. Hence, we saw this increased expressivity as a potentially useful easing of the difficulties inherent in writing specifications.

(*) This work was partially supported by the Esprit WG 8319 (MODELAGE) and through the EPSRC grants GR/K67311, GR/K68783, and GR/G57895.

The analysis of the seminal report [14] by Hans-Dieter Ehrich was also a very important element in the process of establishing our foundational notions. Why did correct implementations not compose correctly in all cases? What were the engineering assumptions on which the technical developments were based? What results were dependent on the formalism used (some variant of equational logic or FOL or whatever) and which were 'universal'?

We quickly established the following paradigm for specification activity as a rational reconstruction of software engineering practice, as perceived through our formalistically tinted glasses: Let us consider program development by means of stepwise refinements. Here one postulates some abstract data type (ADT), suitable for the problem at hand, which has to be implemented on the available system. The end product consists of (the text of) an abstract program manipulating the postulated ADT, together with a suite of (texts of) modules implementing successive ADTs on more concrete ones until reaching the available executable level. See also [31,35,43]. Now one needs some knowledge about the relevant properties of the abstractions involved. This is provided by the axioms in the specifications of the ADTs. The proof that the abstract program does exhibit the required behaviour consists of syntactical manipulations that derive the verification conditions from the ADT specification. Similarly, the correctness of the implementations of the ADTs is verified by syntactical processes, as we shall elaborate upon in the sequel.

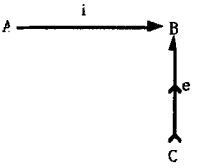
Let us examine more closely what is involved in implementing an abstract data type A on (in terms of) another one, C. The result will be a module representing objects of A in terms of those of C, and operations and predicates of A by means of procedures using operations and predicates of C. We can abstract a little from the actual procedure texts by replacing them by specifications of their input-output behaviours. These amount to (perhaps incomplete) definitions of the operations and predicates of A in terms of those of C and can be regarded as axioms involving both the symbols of A and of C. Similarly, the representation part describes the abstract sorts in terms of the concrete ones, which can be abstracted into axioms introducing the new sorts and capturing (some of) the so-called representation invariants [30,29].

With this abstraction in mind, we are ready to describe this situation in terms of formal specifications, i. e. theories presented by axioms [50,41].

One extends the concrete specification C by adding symbols to correspond to the abstract ones in A, perhaps together with some auxiliary symbols. Since one does not wish to disturb the given concrete specification C, this extension B should not impose any new constraints on C. This can be formulated by requiring the extension B of C to be conservative [46] in the sense that B adds no new consequence to C in the language of the latter.

One then wishes to correlate the abstract symbols in A to corresponding ones in B, much as procedure calls are correlated with their corresponding bodies. But, the properties of A are important, for instance in guaranteeing the correctness of the abstract program supported by A. Thus, in translating from A to B, one wishes to preserve the properties of A as given by its axioms. Hence, one needs a translation $i: A \rightarrow B$ that is an interpretation of theories [46] in the sense that it translates each consequence of A to a consequence of B.

We thus arrive at the concept of an implementation of A on C as an interpretation i of A into a conservative extension B (sometimes called a mediating specification) of C [41]. This is depicted as an implementation 'triangle' below and is often called a "canonical implementation step" [50].



In stepwise development, it is highly desirable to be able to compose refinement steps in a natural way. Let us consider the situation depicted below. Here, one has a first implementation of A on C (with mediating specification B) and a second implementation of C on E (with mediating specification D). See figure 1a below. Now, one would like to compose these two implementations, in an easy and natural manner, so as to obtain a composite implementation of A directly on E. An immediate question that arises is: what would its mediating specification be?

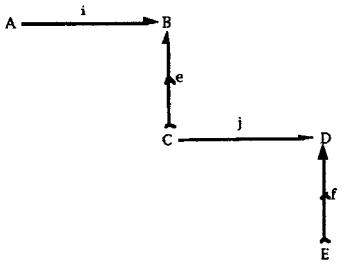


Figure 1a

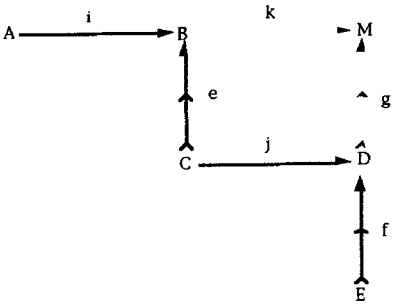


Figure 1b

This is where an important property, the so called Modularisation Property, comes into play. It will allow one to obtain such a mediating specification M, together with an interpretation k of B into M and a conservative extension g of D into M. In other words, it will enable one to complete the rectangle, thereby obtaining a composite implementation of A directly on D, consisting of a composite interpretation of A into M together with a composite conservative extension of E into M. See figure 1b above.

Thus, an immediate benefit of this view is the ability to iterate implementation steps: an implementation of A by C "composes" naturally with one of C by E to yield an implementation of A by E. Here it is worthwhile noting that this composition mimics exactly what a programmer does in simply putting together the corresponding modules (with appropriate linking information).

Another dividend stems from the fact that this view concentrates on the logical aspects of implementation. For, recall that in passing from C to B we add formulae rather than programs. These formulae record the design decisions taken in the implementation, not yet their actual coding into a program text. Therefore, we achieve orthogonality: the process of coding actual modules is independent of - and can proceed in parallel with - the

process of further (logical) refinement, say, in implementing C by E. The successive refinements record the various design decisions.

We saw the purpose of this specification activity (directed to abstract data types) as supporting, and being driven by, program verification. The axioms asserted for a data type were only useful to the extent that they were required in order to prove the correctness of a program using the data type (by whatever verification method was being employed to affect this) and to the extent that they then prescribed required characteristics of an implementation of the data type.

This pragmatics was complemented by another observation about the assumptions underlying data type theory (as then expressed). The early papers motivated the use of initial algebras by stating that the specifier would have in mind a specific algebra whose complete description he/she wanted to construct. Here, 'complete' is used in the logical sense: all atomic sentences are decided (as they of course must be in a model). From a software engineering perspective this is completely unrealistic. Firstly, specifiers never(?) have a complete description of what they specify. Most often, one of the main uses of the specification activity is to build a better understanding of what is being specified. The expectation from traditional science and engineering disciplines is that complex phenomena have no complete descriptions, in the logical sense[1]. Secondly, if the design activity really is about adding implementation oriented detail, then the only way in which this detail can be added sensibly is by making the design more complete. The motivation of supporting program verification by having data types with appropriate properties to affect this just reinforces the idea that specifications should be as 'complete as necessary, but no more so'!

One could then go on to argue that, almost always, even an implementation (of some specification) is not complete in this logical sense. There are some (many?) details left undecided in a program as they are deemed irrelevant to making the program 'work'. This observation then implies that implementations cannot possibly be algebras or models, but are classes of such.

Moreover, we cannot actually work directly with algebras/models in an engineering sense. We can only work with their descriptions. (Of course, we do work with algebras/models in a mathematical or scientific sense.) This analysis may then lead one to posit that specification and software engineering are activities focused on proof theoretic/syntactic manipulations and not on manipulation of their semantic counterparts. These latter are, of course, an indispensable aid to understanding and possibly analysis, but not central to software engineering pragmatics ([37,41,39,50]).

2 A Specification Praxis

In the end, we focused on the following assumptions and tools. They can be classified into methodological/philosophical assumptions and technical assumptions and tools.

---

1 Worse than this, physics has theories about related phenomena which are actually inconsistent at some of the overlaps!

Methodological/Philosophical Assumptions:

(i). Specification supports an engineering activity and this activity consists of manipulating descriptions. Hence, from the point of view of engineering, specification is a proof theoretic pursuit;

(ii). Because of the need for intellectual and engineering economy, the process of design is based on capturing only essentials at any point in the design process. Hence, 'loose' semantics is mandatory and any requirement for (logical) completeness is inherently unsound;

(iii). As in the case of programming languages, there is no prospect of there being a single universally accepted formalism for all specification work. Different problem characteristics will demand different tools for their solution. Given the possible proliferation of specification formalisms, there is some requirement on developing a 'science' of specification. That is, we are interested in finding universal laws which characterise the formalisms or their use, independently of the specific characteristics of an individual formalism.

Technical Assumptions and Tools:

(i). The units of construction of specifications are not terms (as in languages like CCS) or formulae (as in Z), but theories.

Notes: This is simply because substitution (of terms for variables), on the one hand, and use of logical connectives, on the other, are not expressive/general enough for specification construction. The use of logical connectives for specification construction is common (as in Z or various formalisms for concurrency [1, also related is 58]), but is clearly ill-suited to software engineering requirements, if only because they do not help us deal well with scoping of extralogical symbols, renaming, hiding, etc.

(ii). The basic operations for building specifications would appear to be extensions of theories and interpretations between theories.

Notes: Extensions are mainly of use in building individual specifications, by adding further application specific (extralogical) symbols and/or properties. There are two specific subclasses of extensions which play important roles in specification construction: definitional extensions (and their generalisations) and conservative extensions. The former enable the introduction of abbreviations for concepts without essentially changing the underlying theory.

(iii). Conservative extensions form the logical basis of modularity in specification, certainly, and possibly in software engineering, generally.

Notes: Conservative extensions are essential for explaining parameterisation [38,39,40,41,50] (as the 'body' of the parameterised specification is a conservative extension of the specification of the formal parameter) and the concept of implementation.

(iv). The concept of representation/refinement in software engineering is based on interpretations between theories.

Notes: The usual conception of representation was based on a combination of a language translation (signature morphism), which is a syntactic map from the 'abstract' language to the 'concrete' one, combined with a required map from models/structures

associated with the 'concrete' domain to corresponding models/structures of the 'abstract' one. We observed that what we were really interested in doing was preserving the required properties of the abstract entity, so as to not invalidate any proofs of program properties based on it. This is exactly what interpretations between theories capture directly, inducing (indirectly) the map from 'concrete' structures to 'abstract' ones. This direct manipulation of required properties contrasts sharply with the indirect map from 'concrete' models to their 'abstract' counterparts. Interpretations can also be used to explain parameter instantiation. The requirement that a formal parameter is correctly instantiated by an actual parameter is captured exactly by the condition that the instantiation is affected by an interpretation.

(v). Equality must be extralogical because, like other predicates, it must be refined during implementation. This is because 'abstract' values usually have multiple 'concrete' representatives and 'abstract' equality cannot simply be associated with 'concrete' identity.

Notes: The problem with equality was revealed in early work on data types as a 'structure clash'. Certainly, the fact that equational logic is based on a logical equality prevented the choice actually available in FOL between the version with logical equality and the version without. In fact, the adoption of boolean valued functions in the algebraic approach was motivated by two different (but related) issues: how to deal with the requirement for 'tests'/relations in a setting which allows only functions and how to deal with the problem of representing equality. This expedient solution introduced a methodological problem which has not been resolved, namely the relation between the (meta)logical notion of truth in logic, as realised by the judgements of the logic, and the internalised (extralogical) notion of truth, as realised by boolean valued functions. (But see also [11].)

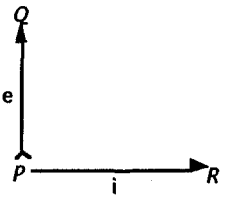
(vi). Relativisation predicates (required in the usual definitions of interpretations to characterise the potentially reduced domains of 'abstract' values as represented by some subset of the corresponding 'concrete' values) provide a simple and logically neat means of dealing with subsorting.

Notes: Subsorting is introduced to deal with relationships between domains and subdomains of values. So, they are used, essentially, to deal with sets and subsets. Subsets are characterised by predicates (properties) and the subset relationships may be represented by the connective of implication. The use of relativised quantifiers (i.e., quantifiers which are relativised to a subdomain of values as characterised by some property) is a simple and straightforward extension of FOL which then deals directly with the issues raised by subsorting.

The Appearance of a Universal Law

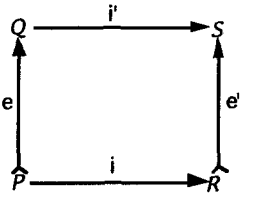
It soon became clear that the composability of implementations and instantiation of parameters required the same underlying mechanism. That is, given figure 2a, (where, for parameter instantiation i is the 'fitting' interpretation and e is the insertion of the formal parameter into the parameterised one, and for composing implementations e is the conservative extension used in the first implementation step and i is the interpretation part of the second), we required that we can complete the diagram automatically to figure 2b:

(S is then the result of the instantiation operation for a parameterised specification and, in the case of implementation composition it is the mediating specification of the composed implementation.) The conjecture that the above result (i.e. that given (A), one can always obtain (B)) was true for FOL was put forward in 1981. It was also observed that the result was not true for the combination of equational logic and initial algebraic semantics and that this accounted for the negative results in [14] concerning composition of implementations.



(A)

Figure 2a



(B)

Figure 2b

The first proof of what became known as the Modularisation Property was put forward in 1982-3 and was based on an observation of Martin Sadler. (See [41] for early attempts at the proof.) What was very surprising indeed to us was that the requirement that from (A) one can obtain (B) in FOL was equivalent to a very important and well-known meta property of FOL encapsulated in what is known as the Craig Interpolation Lemma. This property has a number of formulations, all equivalent for FOL, but we used the most common one:

Let A, B, C be formulae of FOL and L_A, L_B and L_C the collections of extralogical symbols appearing in A, B and C, respectively. If $A \vdash_{FOL} C$ (i.e., A derives C in FOL), then there exists B such that $A \vdash_{FOL} B$, $B \vdash_{FOL} C$ and $L_B \subseteq L_A \cap L_C$. That is, if we can derive C from A, then there is a proof of C from A which uses an intermediate result B such that the extralogical symbols which appear in B occur in both A and C. A computer scientist would recognise this as a kind of modularity result: those symbols not in A and C cannot 'interfere' in the proof. What was very surprising was that this meta property of FOL, developed completely outside the scope of software engineering, was a necessary and sufficient condition for a construction which was directly motivated by software engineering concerns[2].

Interestingly, equational logic does not have the interpolation property and we ascribed the problems identified in [14] to the absence of this meta property. An obvious question to ask if one's favourite formalism does not possess this crucial modularity property is how to 'fix' it. There are two obvious possibilities[3]: either extend the formalism to one

2 In fact, the first proof of this result was based on Robinson's Joint Consistency theorem and its connection with interpolation first aroused our interest.

3 Interpolation properties may be seen as expressing completeness properties with respect to the logical connectives. One needs the existence of some connectives to express interpolants for formulae involving others. For example, for equality one may need a 'conjunction' and an 'implication' - either at the object or the metalevel. Hence, conditional equational logic will have some forms of interpolation properties which normal equational logic will not have. Thus, expanding the logic to regain interpolation properties may be seen as 'completing' the expressivity of the logical connectives.

which does have the property or, alternatively, restrict the specifications which are 'acceptable' to a subclass which does enjoy this property. Therefore, we were not surprised when the work on persistence emerged, reflecting the latter 'design' choice. (Of course, the work on persistence was not directly connected by its inventors with the interpolation and modularity properties.)

Some years later, the important rôle of the Craig Interpolation property was independently observed by the group working with Bergstra [3]. They attributed the lack of certain modularity properties in their formalisms to the absence of the interpolation property. Further work on this revealed an important observation [44]. The various formulations of the Craig Interpolation property for FOL are equivalent. However, for other logics, such as equational logic, the various formulations are not necessarily equivalent. In particular, equational logic does have some versions of the property, but not the crucial one. This version of the property, called Splitting Interpolation in [44], is the following:

(CIP) For formula A and sets of formulae G and G', if $G \cup G' \vdash_{FOL} A$, then there are formulae $B_1,\ldots,B_n$ such that:

a) $G \vdash_{FOL} B_i$ for $1 \le i \le n$,

b) $G' \cup \{B_1, \ldots, B_n\} \vdash_{FOL} A$, and

c) $L_{\{B_1,\ldots B_n\}} \subseteq (L_G \cup L_{G1}) \cap L_A$ (where L_H, for set of formulae H, is the obvious generalisation of L_A, for formula A).

Hence, if A can be proved from G∪G', then there is a set of interpolants $\{B_1,\ldots B_n\}$, each of which is a consequence of the first part of the premises G, and such that A is a consequence of the second part of the premises together with the set of interpolants and such that each B_i only contains extralogical symbols common to G∪G' and A. Crucially, neither equational logic nor conditional equational logic have this property.

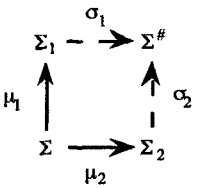
Of even greater import for future developments, it was then noted that the modularisation result was actually asserting the preservation of conservative extensions by pushouts in the category of first order theories (or presentations) and interpretations. In retrospect, this is straightforward, but it was a bit of surprise to us category-phobes! The observation having been made, there was an obvious route to generalisation (first mooted in [38]), in terms of π-institutions ([18]). These latter are a proof theoretic generalisation of institutions, with consequence replacing satisfaction as the focus of attention[4].

This result could appear to have the status of a Universal Law. Consider the following generalisation of CIP: A π-institution has the CIP if for every pushout diagram (in the underlying category of signatures)

---

4 The satisfaction condition of institutions is replaced by the structurality principle: for every signature morplusin $\sigma : L \to L'$, if $G \vdash_L A$, then $Gram(\sigma)(G) \vdash_{L'} Gram(\sigma)(A)$. Notice that $\vdash$, is indexed by languages/signatures. A generalisation of this is the concept of weak structurality, required to deal with many common situations. We require that for each $\sigma: L \to L'$ there is a set $loc(\sigma)$ of formulae (over L') such that if $G \vdash_L A$, then $Gram(\sigma)(G), loc(\sigma) \vdash_{L'} Gram(\sigma)(A)$. (There are also some compositionality requirements on loc with respect to composition of morphisius.) The purpose of $loc(\sigma)$ is to internalise structural/meta constraints for structures over L via formulae over L'. For FOL this includes nonemptiness of the domains of relativisation predicates and closure of the domains of relationisation predicates under the 'concrete' operations corresponding to abstract ones.

We say that a π-institution has the Craig Interpolation Property (CIP) iff for every pushout diagram in the category of signatures



the following property holds: for every $G_1 \subseteq Form(\Sigma_1)$, $G_2 \subseteq Form(\Sigma_2)$, $A_2 \in Form(\Sigma_2)$ such that $\sigma_1(G_1), \sigma_2(G_2), \Phi(\sigma_1), \Phi(\sigma_2), \Sigma^\# \sigma_2(A_2)$, there is a family $I \subseteq Form(\Sigma)$ (of interpolants) such that

- $G_1, \Phi(\mu_1), \Sigma_1 \mu_1(w)$ for every $B \in I$
- $G_2, \Phi(\mu_2), \mu_2(I), \Sigma_2 A_2$

Notice the use of the pushout construction to formalise the notion of "common language" that is required for stating the property (as in [49])[5]. The universal law may then be stated as follows:

The Modularisation Theorem: For a given π-institution, CIP if and only if MP.

More recently, in the context of FOL, Paulo Veloso has been studying very carefully the role of the CIP and its interaction with the deduction rule (another very important metalogical principle) and the modularisation property, [53,52]. These are interesting results and point to very fertile areas of further research.

The effective use of this result should be seen in the adoption of appropriate design principles for specification formalisms. If CIP is not present, we can expect difficulties with parameterisation and implementation. The evidence supporting the important role of interpolation in relation to modularity is now very strong. This area requires much further attention.

3 Equality, Subsorting, Relativisation, etc

There can no longer be any doubt that our early adoption of loose semantics for specifications has been vindicated by developments. None of the work on non-algebraic formalisms adopted this technical tool. (This is obvious in work on program synthesis, program construction, reactive system specification via modal and temporal logics. It is also the modus operandi for 'competing' formalisms such as Z and VDM.) Even in work arising from the algebraic tradition, adoption of loose semantics is almost universal (and adopted by some very early on, e.g. PLUSS).

It may be observed that one instruction on loose semantics which is gaining some

5 If the π-institution satisfies the (strong) structurality property, then the definition simplifies to: if $\sigma_1(G_1), \sigma_2(G_2), \Sigma^\# \sigma_2(A_2)$ there is a family $I \subseteq Form(\Sigma)$ (of interpolants) such that $G_1, \Sigma_1 \mu_1(B)$ for every $w \in I$ and $G_2, \mu_2(I), \Sigma_2 A_2$. This is more akin to the formulation we would expect. The definition we gave was to be expected because, in non structural π-institutions, language is interpreted not directly on the target language but on a theory of the target language. Note the use of pushouts to formalise the notion of common language in earlier formulations of CIP.

popularity is the use of $L_{\omega_1\omega}$ to characterise finitely generated structures.[6] This was adopted in the early 80's ([39,41]) so as to recapture a useful aspect of initiality - being able to underpin some forms of induction. It was easily demonstrable that the usual inductive schemes for standard data structures were a straightforward consequence in $L_{\omega_1\omega}$ of adopting axioms characterising finite generability. This logic had many nice characteristics akin to FOL, including the CIP. The pachage of loose specifications, extralogical equality, FOL formulations of required properties and using $L_{\omega_1\omega}$ to characterise finite generability was reinvented in [57]. Recently, $L_{\omega_1\omega}$ appears to have generated renewed interest in the specification community [4]. A different, but potentially very exciting, use of it is made in [11] where it plays the role of a development logic used to express aspects of design undertaken in the framework of FOL. For example, it may be used to capture in a 'finite' presentation in $L_{\omega_1\omega}$ theories which may not be finitely presentable in FOL, but are required as intermediate steps in a development. Examples of such theories are the results of hiding operations or pullbacks of presentations in FOL. A nice feature of the work is that the entailment relation of the development logic is a conservative extension of the entailment for the specification logic, thus reusing at the 'specification of logics' level the modularity encapsulated via conservative extensions within the specification formalism (i.e., at the level of theories within the logic being specified).

It would appear that extralogical equality is also gaining some adherents as the appropriate way of dealing with the equivalence problem ([36,32]). It is therefore of some interest to note that recent work of Paulo Veloso shows that its role is not as fundamental as originally thought. In fact, extralogical equality may still be enormously useful from an engineering point of view, but mathematically its use can be obviated, in a very precise sense. This is also true for the use of relativisation predicates in interpretations between theories (making the presentations of and reasoning about interpretations technically simpler) and also subsorting via (relativisation) predicates[7].

The interesting result is that the introduction of such a symbol by definition defines a conservative extension of the original theory and, further, that this extension is inessential, in the sense that for any formula involving the new symbol there is a logically

---

6 $L_{\omega_1\omega}$ is an extension of FOL in which one is allowed countably infinite conjunctions and disjunctions in constructing formulae. The number of quantifiers in a formula must still be finite. The associated proof calculus has an infinitary rule of some kind (allowing an infinite set of premises from which to draw a finite conclusion). Finitely generated structures can be characterised by asserting that, for any variable, it must be equal to a variable free term. For finite languages, this can be done via a countable disjunction.

7 It has been demonstrated in [42] that the classical theory of definitions, addressing the introduction into an extralogical FOL language of function and relation/predicate symbols can be extended to enable the introduction by definition of new sorts. Classically, a definition in FOL extends a language L with a (function or relation) symbol a (with appropriate typing) and a corresponding defining axiom (of the form $a(x_1, ,x_n)=y \rightarrow A(x_1, ,x_n)$ for a function symbol and $a(x_1, ,x_n) \rightarrow A(x_1, ,x_n)$ for a relation symbol. In the former case, the formula A must satisfy some strict criteria to ensure thst it describes a function. In both cases, $A \in Form(L)$ and so recursive definitions are not allowed. There is, however, a large body of literature about such recursive definitions and when they make sense.

equivalent formula in the unextended language[8]. What Paulo Veloso has demonstrated ([42]) is that sorts can also be introduced 'by definition' into a language L. The crucial observation is that the introduction of a new sort requires the simultaneous introduction of new function and relation symbols to 'connect' the new sort with the old ones.

Let us first describe how an extension by introduction of a product sort is constructed. We have a specification $P=\langle L,G\rangle$ whose language includes sorts s_1 and s_2, but neither sort t nor operations p_1 or p_2. We first extend language L by introducing sort t and operations p_1, from sort t to s_1, and p_2, from sort t to s_2. We then extend axiomatisation G to $G\cup\{(pjs),(pji)\}$:

$$(\forall x_1{:}s_1)(\forall x_2{:}s_2)(\exists y{:}t)[p_1(y){=}x_1 \wedge p_2(y){=}x_2] \qquad (pjs)$$

$$(\forall y,y'{:}t)[(p_1(y){=}p_1(y') \wedge p_2(y){=}p_2(y')) \rightarrow y{=}y'] \qquad (pji)$$

The new sort t contains 'pairs' of values from s_1 and s_2 with $\{(pjs),(pji)\}$ characterising p_1 and p_2 as the usual projections.

Let us now describe the construction of an extension by introduction of a sum sort. We have a specification $P=\langle L,G\rangle$ whose language includes sorts s_1 and s_2, but neither sort t nor operations i_1 or i_2. We first extend language L by introducing sort t and operations i_1, from sort s_1 to t, and i_2, from sort s_2 to t. We then extend axiomatisation G to $G\cup\{(ijs),(idi),(ii1),(ii2)\}$.

$$(\forall y{:}t)[(\exists x_1{:}s_1)y{=}i_1(x_1) \vee (\exists x_2{:}s_2)y{=}i_2(x_2)] \qquad (ijs)$$

$$(\forall x_1{:}s_1)(\forall x_2{:}s_2)\neg i_1(x_1){=}i_2(x_2) \qquad (idi)$$

$$(\forall x_1,u_1{:}s_1)[i_1(x_1){=}i_1(u_1) \rightarrow x_1{=}u_1] \qquad (ii1)$$

$$(\forall x_2,u_2{:}s_2)[i_2(x_2){=}i_2(u_2) \rightarrow x_2{=}u_2] \qquad (ii2)$$

The new sort t contains the 'disjoint union' of sorts s_1 and s_2 via the injections i_1 and i_2.

We now consider the construction of an extension by the introduction of a sort which is a subsort of an existing one. Consider a language Lwith unary predicate r over sort s, as well as a specification $P=\langle L,G\rangle$. We shall say that predicate r is an *appropriate relativisation predicate* for specification P iff the non voidness of r is derivable frm G. In such a case, if sort t and operation j are not in L, we extend language L by introducing sort t and operation j, from sort t to s. We then extend axiomatisation G to $G\cup\{(jr),(ij)\}$:

---

8 In fact, definitional extensions are special cases of expansive extensions: for every model of the theory over language L, there is an expansion (simply adding a new function/relation) to a model of the definitional extension. In fact, what makes definitions special is that for any model over L there is a unique expansion to interpret the defined symbol. Also, interestingly, conservative extensions properly subsume expansive extensions. Many examples of useful extensions are conservative but not expansive. Further, there is no simple proof theoretic counterpart to expansiveness and the model theoretic counterpart to conservative extension is problematic. See [55,56] for extensive details. See [10] for an amusing attempt to define the difference out of existence!

$(\forall x{:}s)[r(x) \leftrightarrow (\exists y{:}t)x=j(y)]$ (jr)

$(\forall y,y'{:}t)[j(y)=j(y') \rightarrow y=y']$ (ij)

The new sort contains only values in the nonempty subdomain of s defined by r.

Finally, we look at the introduvtion of a quotient sort. Consider a language L with binary predicate q over sort s, as well as a specification $P=\langle L,G\rangle$. We shall say that predicate q is an *appropriate equivalence predicate* for specification P iff the usual congruence properties of q are derivable from G. In such a case, if sort t and operation p are not in L, we extend language L by introducing sort t and operation p from s to t. We then extend axiomatisation G to GU{(sp),(pq)}:

$(\forall y{:}t)(\exists x{:}s)y=p(x)$ (sp)

$(\forall x,x'{:}s)[p(x)=p(x') \leftrightarrow q(x,x')]$ (pq)

The new sort contains values which are quotient classes of the values of s generated by q.

The properties of these sort definitions are exactly analogous to the usual definitions. They allow us to internalise the main constructions which we use to build data types from existing ones[9].

So what are the ramifications of this? Firstly, the ability to introduce quotient sorts means that we can work with FOL with (logical) equality by encapsulating the congruence defined by ause of extralogical equality over an exiting sort as logical equality over a newly introduced quotient sort. The connection between the congruence over the old sort and logical equlity over the new one allows us to use conventional equality reasoning in the result of an implementation step. Secondly, the use of a relativisation predicate and corresponding relativised quantifiers and formulae may be obviated by the introduction of a new sort which is a 'subsort' of an existing sort. For interpretations, this means that we can replace a translation which maps an 'abstarct' sort to a subdomain of a 'concrete' one by a map to a new sort which represents the subdomain. (An analogous statement may be made about replacing subsorting via predicates by a new sort representing the 'subsort'.) Thirdly, the requirement to use concrete versions of functions (or relations) which reflect the need for component based value representations (as in representing 'abstract' stacks by 'concrete' arrays and pointer pairs) may be obviated by the use of product sorts.

The scientific conclusion is that we can avoid the complications introduced by these mechanisms by simply extending our languages by definitions. We can then do our (meta and object level) reasoning in a much simpler mathematical worldwhich is inessentially different from the original. This is a very nice mathematical result, but it may not say very much about engineering prcatice. It is likely(?) that the more (mathematically) cumbersome setting of extralogicalequality, subsorting via relativisation and the lack of explicit product sorts is a more effective engineering tool. More work (of a methodological nature) will need to be done to decide this question.

4 From Configuring Systems to Configuring Programs

In the late '80's, the observation that the Modularisation Property was actually based on the existence of pushouts in the appropriate category of specifications and interpretations

9 Work is proceding to extend these ideas to allow the introduction of inductively defined sorts.

led to an interest in an understanding of how systems were constructed from component parts. This focus on configuration was motivated by the need to distinguish between and to explain the act of using a pre-existing specification as a basis for defining an extended one (i.e., providing language mechanisms to reflect the construction of an extension) and the activity of causing some components to share (in a 'physical' sense) some subcomponent. The former is illustrated by extending a specification of natural numbers to a stack of the same, while the latter is illustrated by configuring a producer/consumer system to communicate through a shared buffer. Making this distinction in the setting of FOL (or algebraic formalisms) is difficult as the idea of behaviour appears to be inherent in understanding the intention of sharing subcomponents in this sense[10].

In the early 70's, J.Goguen proposed the use of categorical techniques in *General Systems Theory* for unifying a variety of notions of system behaviour and their composition techniques [25,27]. His approach has been summarised in a very simple but far reaching principle: "given a category of widgets, the operation of putting a system of widgets together to form a super-widget corresponds to taking a colimit of the diagram of widgets that shows how to interconnect them".

The evidence that we have been able to obtain over the last 7 or 8 years would suggest that this 'maxim' would appear to have the status of a universal law. Its ramifications have led to interesting insights and new developments which open exciting avenues of research.

The technical motivation for the work was the following question: How could the observation that a large system, described in terms of its overall behaviour, cannot be given whole because of its size be ameliorated by constructing the overall behavioural description from that of its parts? In our work on requirements engineering using Modal Action Logic, industrial experiments demonstrated that structuring of specifications was an engineering necessity. It was clear, that parameterisation was an orthogonal issue and the crux of the problem was the one outlined above - sharing of subcomponents by parts of the system. Given that we were ever more wedded to the idea that specification was theory manipulation and that relationships between theories were established via interpretations between theories, the categorical connection became obvious and, together with José Fiadeiro, we explored how this could actually be done.

We shall now illustrate the approach using linear temporal logic [28,17] using a mixture of both propositional and first order versions, as convenient. In wishing to model reactive systems, computations provide us with a semantic domain in which we can reason about the properties of a system (safety and liveness) using a temporal logic [1,17]. In preparation for relating specifications and the programs as defined in the previous section, we shall illustrate the categorical account of specifications through a category of temporal theories as in [17].

Specifications are themselves built over signatures (the extralogical language of the specification). In the case of the temporal logic that we have in mind, these consist just of pairs of sets, $\tau=(\Pi, \Lambda)$, of nonrigid (state dependent) constants – corresponding to attribute and action symbols, respectively. A morphism $\sigma:(\Pi, \Lambda)\rightarrow(\Pi', \Lambda')$ is a pair of total

10 This may be seen as a reappearance of the old philosophical canard - use versus mention.

functions $\sigma_\Pi: \Pi \to \Pi'$, $\sigma_\Lambda: \Lambda \to \Lambda'$. Temporal signatures constitute a category *t-SIGN*. The temporal language defined over a signature is as follows: Given a temporal signature $\tau=(\Pi, \Lambda)$, the language of terms over a sort $s \in S$ is:

$$t_s ::= a \mid c \mid f(t_{1_{s_1}}, \ldots, t_{n_{s_n}}) \mid \mathbf{X}t_s \text{ for } a \in \Pi_s,\ c \in \Omega_{<>,s} \text{ and } f \in \Omega_{<s_1,\ldots,s_n>,s}.$$

The language of temporal propositions is, for $p \in \Lambda$:

$$\phi ::= (t_{1_s} =_s t_{2_s}) \mid p \mid (\phi_1 \supset \phi_2) \mid (\phi_1 \wedge \phi_2) \mid (\neg\phi) \mid \mathbf{beg} \mid \mathbf{X}\phi \mid \phi_1 \mathbf{U} \phi_2$$

The special operators are **beg** (denoting the initial state), **X** ($\mathbf{X}\phi$ holds in a state when ϕ holds in the next state), and **U** ($\phi \mathbf{U} \psi$ holds when ψ will hold sometime in the future and ϕ holds between now and then). A temporal theory is a pair (τ, Φ) where Φ is a set of τ-propositions such that $\phi \in \Phi$ for every $\Phi, \vdash_\tau \phi$. A presentation of a theory (τ, Φ) is a pair (τ, Ψ) such that $\Phi = \{\phi: \Psi, \vdash_\tau \phi\}$. By $,\vdash_\tau$ we mean the usual consequence relation for linear, discrete temporal logic [e.g., 28][11]. Morphisms between theories (and presentations) also require a translation between the temporal languages: Given a sigature morphism $\sigma: \tau \to \tau'$:

$$\sigma(t) ::= \sigma(a) \mid c \mid f(\sigma(t_1), \ldots, \sigma(t_n)) \mid \mathbf{X}\sigma(t)$$

$$\sigma(\phi) ::= (\sigma(t_1) = \sigma(t_2)) \mid \sigma(p) \mid (\sigma(\phi_1) \supset \sigma(\phi_2)) \mid \neg\sigma(\phi) \mid \mathbf{beg} \mid \mathbf{X}\sigma(\phi) \mid (\sigma(\phi_1) \mathbf{U} \sigma(\phi_2))$$

A morphism of theory presentations σ: $(\tau_1, \Phi_1) \to (\tau_2, \Phi_2)$ is a signature morphism $\sigma: \tau_1 \to \tau_2$ such that $\Phi_2, \vdash_{\tau_2} \sigma(\phi)$ for every $\phi \in \Phi_1$. Theory presentations and their morphisms constitute a category *SPEC*. That is, a morphism of presentations is a signature morphism that defines a theorem preserving translation between the two theories. It is a generalisation of interpretations between theories. These are standard notions within *institutions* [26,45,49]. For instance, it is a property of institutions that if the category of signatures admits colimits, so does the category of theory presentations. Hence, *SPEC* admits colimits of finite diagrams (i.e. is finitely cocomplete).

Colimits of specification diagrams are computed over the colimit of the underlying signature diagram: a pushout of two morphisms μ_1: $(\theta, \Phi) \to (\theta_1, \Phi_1)$ and μ_2: $(\theta, \Phi) \to (\theta_2, \Phi_2)$ is given by the specification (θ', Φ') and morphisms σ_1 and σ_2 such that θ', σ_1 and σ_2 are a pushout of μ_1 and μ_2 as signature morphisms and $\Phi' = \sigma_1(\Phi_1) \cup \sigma_2(\Phi_2)$.

That is, the set of axioms of the composite specification is the union of the translations of the axioms of the components. Because the union of sets of formulae has the same logical value as their conjunction, the categorical approach complies with the "composition as conjunction" idea put forward in [1] for parallel composition of reactive systems and also, in a related sense, in [58]. However, we should stress that our approach is more "structured" in the sense that formulae are not being considered individually as units of construction but

11 That is to say, a theory over a signature is a set of propositions closed under consequence (it contains all of its theorems). A presentation of a theory is a set of propositions whose closure (set of theorems that can be derived) is that theory.

are organised into modules (theories) that have a meaning in terms of the structure of the system – hence the use of morphisms for establishing interconnections through the language of these theories, something that cannot be achieved at the level of individual formulae.

That reactive system specification can be presented modularly via temporal or modal logic theories and colimits is no longer a surprise, although perhaps not yet commonly accepted. (As noted immediately above, it is still stuck in the more primitive, unmodularised settings where formulae and connectives rule.) A perhaps more radical and much more surprising application of the same ideas is possible in the world of programs. The original observation that this is possible is due to José Fiadeiro and it helps to demonstrate the ubiquity of Goguen's original observation (lending weight to its being a universal law) and pointing to a possible answer to a very old question: How do programs fit into this world of specifications and designs?

In [22], we showed how parallel program design in the language COMMUNITY (similar to UNITY [9] and IP – Interacting Processes [24], but using a richer model of system interconnection and superposition) can be formalised using the same categorical techniques[12]. From a categorical point of view, programs are objects and morphisms capture superpositions. The colimit construction corresponds to a generalised parallel composition operator with synchronisation constraints (i.e., to superimposition in the sense of [24]).

A COMMUNITY program P has the following structure:

$$
\begin{aligned}
P \equiv\ & \textit{data}\ \Sigma \\
& \textit{read}\ R \\
& \textit{var}\ V \\
& \textit{init}\ I \\
& \textit{do}\quad []_{g\in\Gamma}\ \ g: [B(g) \rightarrow \ ||_{a\in D(g)}\ \ a:=F(g,a)]
\end{aligned}
$$

where: Σ represents the data types that the program uses, given through a signature (S,Ω) in the usual algebraic sense [12]. For simplicity, we shall assume that the data types are fixed and omit the *data* clause from programs; R is the set of external attributes, i.e. the attributes that the program needs to read from its environment (*open* attributes in the sense of IP); V is the set of local attributes (the program "variables"); A is the union (assumed disjoint) of R and V, the set of attributes of the program; attributes are typed – every attribute $a\in A$ has an associated sort s; A_S will denote the set of attributes of sort s; the distinction between the two classes of attributes is necessary to formalise superposition, namely forms of program interconnection that result from superposing regulators over base programs – a regulator can read the attributes of the base program but cannot update them; Γ is the set of *action names*; each action name has an associated statement (see below) and can act as a *rendezvous* point for program synchronisation; I is a condition on the attributes – the initialisation condition; for every action $g\in\Gamma$, B(g) is a condition on

12 The underlying computational model is also similar to Action Systems [2], but we should point out that the Action Systems approach, at least in relation to transformational development, is oriented to *decomposition* of systems into components. Our focus is on *composition*.

the attributes – the *guard* of the action; for every action $g \in \Gamma$, $D(g) \subseteq V$ is the set of attributes that action g can change; we also denote by D(a), where $a \in V$, the set of actions that can change a; for every action $g \in \Gamma$ and local attribute $a \in D(g)$, F(g,a) is an expression that has the same type as a.

Formally, a *program signature* is a triple (V,R,Γ) where V and R are S-indexed families of sets and Γ is a 2^V-indexed family of sets. All these sets of symbols are assumed to be finite and mutually disjoint. Attributes are used as atoms in the definition of terms: Given a signature $\theta=(A,\Gamma)$, the language of terms is defined as follows: for every sort $s \in S$, for $a \in A_s$, $c \in \Omega_{<>,s}$, and $f \in \Omega_{<s_1,\dots,s_n>,s}$:

$$t_s ::= a \mid c \mid f(t_{1s_1},\dots,t_{ns_n})$$

The language of propositions is defined as follows:

$$\phi ::= (t_{1s} =_s t_{2s}) \mid (\phi_1 \supset \phi_2) \mid (\phi_1 \wedge \phi_2) \mid (\neg\phi)$$

Terms and propositions are used to define programs[13]. Given a signature $(A=V \oplus R,\Gamma)$, and a subset $V' \subseteq V$, a V'-command F maps every attribute $a \in V'_s$ to a term F(a) of sort s. Commands model multiple assignments. The term F(a) denotes the value that is assigned to a. If V' is empty (which is the case, for instance, for some communication channels), the only available command is the empty one: *skip*.

A *program* is a pair (θ,Δ) where θ is a signature (A,Γ) and Δ, the *body* of the program, is a triple (I,F,B) where: I is a θ-proposition (constraining the initial values of the attributes); F assigns to every action $g \in \Gamma$ a D(g)-command; and B assigns to every action $g \in \Gamma$ a θ-proposition (its guard).

It is easy to recognise in this definition the basic features of parallel programs, namely guarded simultaneous assignments: each action g defines the guarded command

$$[B(g) \rightarrow \|_{a \in D(g)} \; a:=F(g,a)]$$

There are, however, some distinguishing features of COMMUNITY that should be discussed: the typing and the naming of actions. Each domain D(g) consists of the attributes to which action g can make assignments. We shall also work with the dual notion, i.e. we define for every attribute $a \in V$ the set of the actions that can assign to a – $D(a)=\{g \in \Gamma \mid a \in D(g)\}$. There is a difference between the fact that an attribute a is not in the domain of action g and the fact that g performs the assignment a:=a. The difference between these two situations is important from the point of view of concurrency within programs. But, the idea is that actions are allowed to occur concurrently (i.e. as part of the same event), e.g. actions that come from two program components that were put together in parallel. Hence, an action presents only a partial view of the transformation that is performed by a (global) event, namely it is concerned with only a subset of the attributes of the program. The assignment of specific domains to actions is, thus, a means of controlling the interference between different program components.

---

13 For simplification, every boolean term b will be used as an abbreviation of the proposition (b=true).

The separation between action *names* (i.e., the set Γ) and the guarded commands they execute (as given by F and B) is important for the definition of superposition and also to support interaction in the sense of IP. COMMUNITY differs from IP in that every action is a potential point of interaction. Indeed, interaction names in COMMUNITY are not global as in IP: interaction is established outside the programs, at "system configuration time", by identifying action names belonging to different component programs.

An example of a program is the following:

$P_\Gamma \equiv$ *read* x:int
var a:int; d:bool
init d=false $\wedge$ a=0
do t : [$\neg$d$\wedge$x=a $\rightarrow$d:=true] [] r : [$\neg$d$\wedge$x$\neq$a $\rightarrow$a:= x]

Intuitively, this program is capable of successively reading (action r) the value of the external attribute x, stopping (action t) whenever it consecutively reads the same value or the first value it reads is 0.

Having defined programs over signatures, we now define signature morphisms as a means of relating the "syntax" of two programs: Given signatures $\theta_1=(A_1=V_1 \oplus R_1, \Gamma_1)$ and $\theta_2=(A_2=V_2 \oplus R_2, \Gamma_2)$, a *signature morphism* σ from θ_1 to θ_2 consists of a pair $(\sigma_\alpha: A_1 \rightarrow A_2, \sigma_\gamma: \Gamma_1 \rightarrow \Gamma_2)$ of (total) functions such that $\sigma_\alpha(V_1) \subseteq V_2$ and, for every action $g \in \Gamma$, $\sigma_\alpha(D_1(g)) \subseteq D_2(\sigma_\gamma(g))$.

Morphisms are intended to capture the relationship that exists between a program (system) and its parts (components). Hence, a signature morphism maps attributes of a program to attributes of the system of which it is a component, and the same for actions. Because the system "contains" the component, attributes of the component program cannot be read attributes of the system, thus justifying the restriction $\sigma_\alpha(V_1) \subseteq V_2$. No restriction is put on R_1 because read attributes of the component program can be attributes of another component program for the same system and, hence, elements of V_2. The restriction over action domains just means that the type of each action is preserved by the morphism. Notice that more attributes may be included in the domain of an action via a morphism. This is intuitive because, within a system, an action of a component may be shared with other components and, hence, have a larger domain. For simplicity, we shall ommit the indexes α and γ when referring to the components of a morphism. Program signatures and their morphisms constitute a category $\mathcal{SIG}$. Signature morphisms provide us with the means for relating a program with its superpositions. However, superposition is more than just a relationship between signatures[14], i.e. more than "syntax".

14 To capture its intended semantics, we have to analyse the bodies of the two programs involved. Given two programs (θ_1,Δ_1) and (θ_2,Δ_2) and a signature morphism σ: $\theta_1 \rightarrow \theta_2$, we have to look for relationships between Δ_1 and Δ_2 such that (θ_2,Δ_2) can be considered a superposition of (θ_1,Δ_1) via σ, i.e., for σ to be considered as a superposition morphism. We need a way of relating the models of the two programs as well as the terms and formulas that are used to build them. Given a signature morphism σ: $\theta_1 \rightarrow \theta_2$ and a θ_2-interpretation structure $\mathcal{S}=(\mathcal{T},\mathcal{A},\mathcal{G})$ (with $\mathcal{T}$ a transition system, $\mathcal{A}$ an assignment of state dependent values to attributes and $\mathcal{G}$ mapping action symbols to sets of events),

Signature morphisms define translations between the languages associated with each signature in the obvious way: given a signature morphism $\sigma: \theta_1 \rightarrow \theta_2$,

$$\sigma(t) ::= \sigma(a) \mid c \mid f(\sigma(t_1),\ldots,\sigma(t_n))$$

$$\sigma(\phi) ::= (\sigma(t_1)=\sigma(t_2)) \mid (\sigma(\phi_1)\supset\sigma(\phi_2)) \mid (\sigma(\phi_1)\wedge\sigma(\phi_2)) \mid \neg\sigma(\phi)$$

There are several notions of superposition in the literature [5,9,34,23,24], corresponding to different meanings of "preservation of the underlying program". We consider, in the first instance, *regulative superposition* in the sense of [24].

Viewed as a transformation (which is the view captured by morphisms), regulative superposition requires that the functionality of the base program be preserved in terms of the assignments performed on its variables, but it allows for the guards of its actions to be strengthened. This characterisation leads to the following definition of a (regulative) superposition morphism: A *superposition morphism* $\sigma: (\theta_1,\Delta_1)\rightarrow(\theta_2,\Delta_2)$ is a signature morphism $\sigma: \theta_1\rightarrow\theta_2$ such that

1. For every $g_1\in\Gamma_1$ and $a_1\in D_1(g_1)$, $,\theta_2 B_2(\sigma(g_1))\supset\sigma(F_1(g_1,a_1))=F_2(\sigma(g_1),\sigma(a_1))$;
2. $,\theta_2(I_2\supset\sigma(I_1))$;
3. For every $g_1\in\Gamma_1$, $,\theta_2(B_2(\sigma(g_1))\supset\sigma(B_1(g_1)))$;
4. For every $a_1\in V_1$, $D_2(\sigma(a_1))\subseteq\sigma(D_1(a_1))$.

Requirements 1 and 2 correspond to the preservation of the functionality of the base program: the effects of the instructions are preserved and so are the initialisation conditions. Requirement 3 allows guards to be strengthened but not to be weakened. Requirement 4 corresponds to a locality condition: new actions cannot be added to the domains of attributes of the source program. That is to say, no new actions can change the old attributes. Together with the fact that signature morphisms preserve the domains of actions, it implies that the domains of the attributes remain the same up to translation, i.e. $D_2(\sigma(a_1))=\sigma(D_1(a_1))$ for every $a_1\in V_1$[15].

As an example of a superposition morphism consider the following programs where $\varphi,\psi: int,int\rightarrow int$ are operations of the underlying data type:

---

its *σ-reduct*, $\mathcal{S}|_\sigma$, is the θ_1-interpretation structure $(\mathcal{T},\mathcal{A}|_\sigma,\mathcal{G}|_\sigma)$ where $\mathcal{A}|_\sigma(a) = \mathcal{A}(\sigma(a))$, and $\mathcal{G}|_\sigma(g) = \mathcal{G}(\sigma(g))$.

That is, we take the same transition system and interpret attribute and action symbols in the same way as their images under σ. Reducts provide us with the means for relating the behaviour of a program with that of the superposed one. Then, given a θ_1-formula ϕ and a θ_2-interpretation structure $\mathcal{S}=(\mathcal{W},\mathcal{A},\mathcal{G})$, we have for every $w\in\mathcal{W}$: $(\mathcal{S},w),\sigma(\phi)$ iff $(\mathcal{S}|_\sigma,w),\phi$. Readers familiar with institutions [26,45,49] will have recognised in this proposition the "satisfaction condition". Although the formalism that we work with in this paper is not an institution (*stricto sensu*), we shall make use of many of the categorical techniques that have been popularised by institutions.

15 This condition implies the following property: Let $\sigma: (\theta_1,\Delta_1)\rightarrow(\theta_2,\Delta_2)$ be a superposition morphism. Then, the reduct of every locus of (θ_2,Δ_2) is also a locus of (θ_1,Δ_1). Here locus is a model in which a change in a program variable occurs only in transitions which witness an action of the program. This is a semantic characterisation of encapsulation. See [17,19,20,22].

$P_b \cong$	*var*	a,b:int	$P_s \cong$	*var*	a,b,ao:int; d:bool
	init	$a>0 \wedge b>0$		*init*	$a>0 \wedge b>0 \wedge d=false \wedge ao=0$
	do	f : [true$\rightarrow$a:=φ(a,b)]		*do*	fr : [$\neg d \wedge ao \neq a \rightarrow a:=\varphi(a,b) \| ao:=a$]
	[]	g : [true$\rightarrow$b:=ψ(a,b)]		[]	g : [true $\rightarrow$ b := ψ(a,b)]
				[]	t : [$\neg d \wedge ao=a \rightarrow d:=$ true]

All the conditions above are satisfied by the mapping <aúa, búb, fúfr, gúg>, so that Δ_s is a (regulative) superposition of Δ_b. Notice that, according to this definition, it is possible for the "old" actions to assign to "new" (superposed) variables. For instance, *fr*, the image of *f*, assigns to the new attribute *ao*. However, the new actions, like *t*, cannot assign to the old attributes, like *a*. Moreover, the guard of an old action, like *f*, can be strenghtened.

Significantly, programs and superposition morphisms form a category $\mathcal{REG}$. That is to say, superposition morphisms compose (i.e., we can support iterated superposition), and the identity morphism (a kind of "empty" superposition) is a unit for composition. As we have already mentioned, there are other notions of superposition[16].

An interesting class of morphisms are those which do not allow guards to be strengthened. Such superposition morphisms are called *spectative* in [23]. They also correspond to the notion of superposition used in UNITY [9]. A *spectative superposition morphism* σ: $(\theta_1,\Delta_1)\rightarrow(\theta_2,\Delta_2)$ is a signature morphism σ: $\theta_1\rightarrow\theta_2$ such that σ is injective over attributes and actions, and the augmented requirements (with 1 and 4 as above):

2. $,\theta_2(I_2 \supset \sigma(I_1))$ and, for every formula ϕ in the language of θ_1, if $,\theta_2(I_2 \supset \sigma(\phi))$ then $,\theta_1(I_1 \supset \phi)$;

3. For every $g_1 \in \Gamma_1$, $,\theta_2(B_2(\sigma(g_1)) \equiv \sigma(B_1(g_1)))$;

Injectivity of σ means that no confusion is introduced among attributes nor among actions of the superposed program. Condition 3 now requires that guards remain unchanged and condition 2 requires that the strenghtening of the initial condition be conservative, i.e. it cannot put further constraints on the initial values of the attributes of θ_1. This is indeed an interesting and posssibly surprising reoccurrence of the idea of conservative extension.

Spectative superposition morphisms define acategory $\mathcal{SPE}$, where the objects are still programs, i.e. the categories $\mathcal{REG}$ and $\mathcal{SPE}$ just differ on the morphisms. It is, however, the morphisms that characterise the structural properties of a category, meaning that the different notions of superposition will have different algebraic properties[17]. The

16 Invasive superposition allows for new actions to update old attributes. Hence, they are not required to satisfy the locality condition (4). This is a potentially inappropriate breaking of encapsulation and its role in program construction may be problematic.

17 We can prove a fundamental property of spectative superposition: that it is model expansive. This property means that spectative superposition does not change the base program, i.e., through σ, the base program is extended without affecting its underlying behaviour: Let $\sigma:(\theta_1,\Delta_1)\rightarrow(\theta_2,\Delta_2)$ be a spectative superposition morphism. Then, for every model $\mathcal{S}$ of (θ_1,Δ_1), there is a model $\mathcal{S}'$ of (θ_2,Δ_2) such that $\mathcal{S}\sim\mathcal{S}'_\sigma$. (There is an interesting conundrum here: whereas the requirement of conservativeness on the strengthening of guards is an obvious logical condition, programs are not theories and, hence, we cannot simply assert that one is a conservative extension of the other. However, programs' models can be related via model expansion!)

difference between these these classes of morphisms has also been characterised [23] in terms of the preservations of the safety and liveness properties of programs.

We can illustrate the idea of using colimits to construct programs from components by imposing a regulator P_r over the program P_b via 'channel' C, which synchronises the actions b and g of P_b with actions x and r of P_r, respectively, on the one hand and ditto a, f and x,r on the other. (The program P_S illustrated above is then (up to isomorphism) the pushout of P_b and P_r via C using the second synchronisation.) The resulting program would detect situations in which $b=\psi(a,b)$ and situations in which $a=\varphi(a,b)$. However, it does not necessarily detect a situation in which both $a=\varphi(a,b)$ and $b=\psi(a,b)$. In order to achieve this, we need to synchronise the actions that detect the local fixpoints, i.e., the different occurences of t in the two different uses of P_r. This can be done by adding another communication channel C' to the configuration diagram of figure 3a, where C' ≡ *do* h : [*skip*].

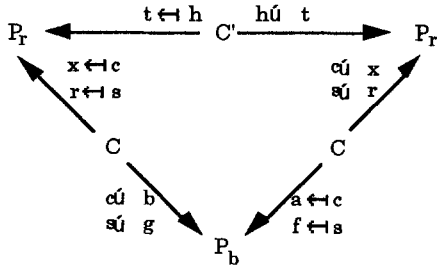


Figure 3a

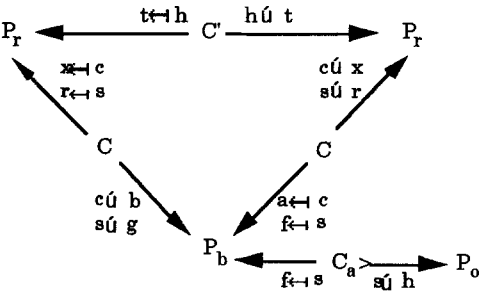


Figure 3b

The resulting program P_S' is the result of the double superposition. It is isomorphic to:

var	a, b, ao, bo : int; ad, bd : bool
init	a>0 ∧ b>0 ∧ ad=false ∧ bd=false ∧ ao=0 ∧ bo=0
do	fr : [¬ad ∧ ao≠a → a := φ(a,b) || ao := a]
[]	gr : [¬bd ∧ bo≠b → b := ψ(a,b) || bo := b]
[]	ft : [¬ad ∧ ao=a → ad := true]
[]	gt : [¬bd ∧ bo=b → bd := true]

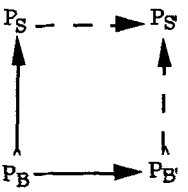
Now, we may wish to impose an 'observer' on this program which counts the number of assignments to a necessary to reach the fixpoint. We call this component an observer because we would not expect it to alter in any way the behaviour of the program to which it is applied. A spectative morphism is in order and we use the following program which counts the number of times which an action occurs:

P_O ≡	*var*	c : int;
	init	c=0
	do	h : [true→c:=c+1]

We need to synchronise incrementing c in P_O with f in P_b. We do this via $C_a \equiv$ *do* s : [*skip*]. We require that the morphism from C_a to P_O be spectative (and surjective on attributes and actions). This then gives us the diagram of figure 3b.

One of the main purposes of this construction is to introduce new attributes that may account for the observations that are required by the specification of some intended system. The ability to reuse an existing piece of software (program) to satisfy a specification should allow for both the superposition of a regulator, to tune the behaviour of the underlying program to the behavioural requirements of the specification, and the superposition of an observer over the regulator+program system, to account for the state observations required by the specification.

We can demonstrate a very important result: given such a spectative superposition P_S of a base program P_B, if P_B is independently extended to $P_{B'}$ (e.g., as a result of superposing a regulator) then there is a canonical spectative superposition $P_{S'}$ of $P_{B'}$ that provides for the observations added to P_B through P_S.



This property is an instance, for the world of programs, of the Modularisation Property discussed above[18]. It implies that any spectative superposition of a program is reflected in a unique way on any system of which the program is a component. Hence, it is possible to identify a system with its configuration diagram as done above in the context of regulative superpositions. That is to say, for the interconnection of P_O as above, the order in which the superpositions are made, including the spectative one, is immaterial. This means that the superposition of regulators and of monitors "commutes", i.e., both configuration techniques can be used as part of an incremental development process. We can superpose a monitor over a base program and later on superpose a regulator over the same base program without affecting the "status" of the first extension as a spectative superposition.

What now can be said about connecting programs and the specifications which they are to satisfy? Briefly, define for every program signature $\theta=(A=V\oplus R, \Gamma)$, the temporal signature $Spec(\theta)=(A,\Gamma)$. This mapping extends trivially to a functor $Spec$: $SIG \rightarrow t\text{-}SIG$ by mapping morphisms of program signatures to themselves. That is to say, we map a program signature to a temporal signature by taking the attributes as the non-rigid constants and the actions as the atomic propositions[19].

---

18 Since programs are not theories, the immediate connection of this result with interpolation is not obvious. However, interconnection of some kind there must be!

19 This is a good example of a mapping between two formalisms that are at different levels of abstraction: the information about which attributes are local and which are external is lost during the mapping process because the notion of temporal signature is not strong enough to capture it. Indeed, temporal logic is a formalism that can be associated with many other program design languages and, hence, its logical symbols do not commit the specifier to any particular encapsulation discipline.

A consequence of this is that the "semantics" of the programming formalism will have to be translated, in part, to extralogical axioms in temporal logic. (Put in another way, conditions imposed on each program implicitly by the formalism of programs will have to be made explicit in the corresponding specification since the latter formalism does not impose the same 'discipline' of encapsulation[20].) Indeed, the mapping that really is of interest is the extension of *Spec* to a functor between $\mathcal{REG}$ and $\mathcal{SPEC}$ defined by: map every program (θ, Δ) to the theory presentation *Spec*(θ, Δ) whose signature is *Spec*(θ) and whose set of axioms *Spec*(Δ) consists of:

- the proposition $(\mathbf{beg} \supset \mathrm{I})$;
- for every action $g \in \Gamma$ and every $a \in D(g)$, the proposition $(g \supset \mathbf{X}a = F(g,a))$;
- for every $g \in \Gamma$, the proposition $(g \supset B(g))$;
- for every $a \in V$, the proposition $((\bigvee_{g \in D(a)} g) \vee \mathbf{X}a = a)$

These (extralogical) axioms do capture the semantics of the program: the first axiom establishes that I is an initialisation condition; the second set of axioms formalises assignment – if g is about to occur, the next value of attribute a is the current value of F(g,a); the third establishes B(g) as a necessary condition for the occurrence of g; and the last axiom (the locality axiom) captures locality (encapsulation) of attributes: if, in a given state, none of the actions of the domain of an attribute occurs, that attribute remains invariant during the next state transition [17]. For instance, the program P_S introduced in section 3 admits the following presentation:

Spec(P_S) ≡ $\mathbf{beg} \supset a>0 \wedge b>0 \wedge d=\mathrm{false} \wedge ao=0$

$fr \supset \mathbf{X}a=\varphi(a,b)$	$fr \supset \mathbf{X}ao=a$
$fr \supset \neg d \wedge a \neq ao$	$g \supset \mathbf{X}b=\psi(a,b)$
$t \supset \mathbf{X}d=\mathrm{true}$	$t \supset \neg d \wedge a=ao$
$fr \vee \mathbf{X}a=a$	$fr \vee \mathbf{X}ao=ao$
$g \vee \mathbf{X}b=b$	$t \vee \mathbf{X}d=d$

We have thus defined a mapping *Spec* from the objects of $\mathcal{REG}$ to the objects of $\mathcal{SPEC}$. In order to prove that this mapping extends to morphisms and, hence, defines a functor, it is sufficient to see that, given a program morphism σ: $(\theta,\Delta) \rightarrow (\theta',\Delta')$, the conditions laid down in the definition for program morphisms together with the axioms of *Spec*(θ,Δ) imply the axioms of *Spec*(θ',Δ')[21].

---

20 This should remind the reader of the weak structurality principles for specification morphisms, capturing via extralogical axioms over the target language the meta constraints over the domain language.

21 Notice that if a different notion of superposition (i.e. of program morphism) had been chosen, *Spec* might not be a functor, i.e. it might not map the program morphisms (of this new category) to specification morphisms. Indeed, the "semantics" of the programming language is more encoded in the morphisms than in the objects.

For this mapping to be really useful, we want structure preservation between programs and specifications. Compositionality, in a nutshell, is a property of the relationship between specifications and programs which ensures that a problem of correctness for a composite system can be decomposed into similar problems of correctness for the components of the system. Hence, compositionality requires a suitable relationship between the constructions available for building systems and the notion of correctness between systems and specifications.

We indicate how the functor defined above allows us to formalise the notion of satisfaction (correctness) between programs and specifications and to define compositionality as an algebraic property of the two formalisms – programs and specifications. A *realisation* of a specification S is a pair $<\sigma,P>$ such that $P:\mathcal{REG}$ and σ is a specification morphism $S\rightarrow Spec(P)$[22]. Now consider the $\mathcal{SPEC}$ diagram given by φ_1 and φ_2, interconnecting specifications S_1 and S_2 via a channel S. Let $<\eta,P>$, $<\eta_1,P_1>$, $<\eta_2,P_2>$ be realisations of S, S_1 and S_2, respectively (i.e., η: $S\rightarrow Spec(P)$, η_i: $S_i\rightarrow Spec(P_i)$), interconnected in a way that is consistent with the interconnection of the specifications, i.e., μ_i: $P\rightarrow P_i$ are such that $\eta;Spec(\mu_i)=\varphi_i;\eta_i$. Then, there is a unique way in which the pushout program P' is a realisation of the pushout specification S', i.e. there is a unique $\eta':S'\rightarrow Spec(P')$ such that $\beta_i;\eta'=\eta_i;Spec(\sigma_i)$[23]. See figure 4 below. See also [19,20,16,22].

We now have an answer to the 'age old question'[24]: how are programs and specifications related in the general setting of specification formalisms as general as FOL, algebraic languages and temporal/modal logics. This development was not foreseen when deciding to represent configuration of systems from components by colimits of configurations of

---

22 This notion of realisation is a generalisation of the *satisfaction relation* between programs and specifications. Traditionally, we say that a program P satisfies a specification S, P,S, if every computation of P is a model of S. Alternatively, in some calculi prorams and specifications are formulae and the morphism above is replaced by logical implication. Realisations generalise this notion by allowing the program and the specification to be over different signatures. More concretely, the program is allowed to have features that are not relevant to the specification. Hence the morphism from S to *Spec*(P) corresponds to the way in which P realises S, i.e., intuitively, it records the design decisions that lead from S to P (seen as a design exercise carried out in $\mathcal{SPEC}$).

23 Of course, we intend that this generalises to colimits. We should point out that this result holds for any functor *Spec* between categories of programs and of specifications. That is to say, it does not depend on the nature of the program design language (as long as it can be defined as a category) or of the specification logic (as long as it can be defined as an institution).

24 We had serious worrie starting 20 years ago about how programs arose from specifications and refinements. There was discussion about how development should proceed to the point where the last target language of a refinement was directly realisable in a programming language. But this begged two important questions: firstly, how was this last step actually realised when there was normally a change of logic (from that of the specification formalism to that of the programming language) involved and, secondly, what about the modules/clusters corresponding to each of the previous steps? In the terminology of the first section, the latter question can be rephrased as: The mediating specification B used in implementing A in terms of C specifies the data representations and operation implementations required to realise the development step; how exactly are these realised by programs?

component descriptions, but it should be seen as the emergence of a discipline (universal principle or law) about the relationship between programs and specifications[25].

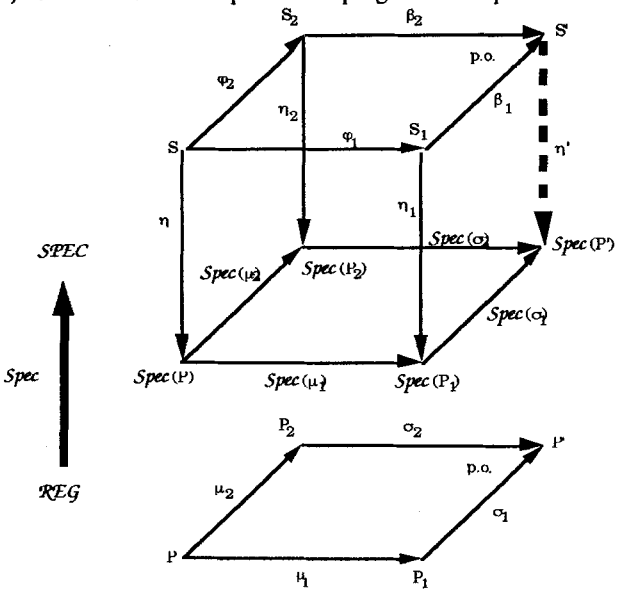


Figure 4

5 Concluding Remarks

Some twenty years ago, a group of us focused on an alternative framework from that being developed around the initial algebraic ideas put forward in [29,30,31,14,6]. It was also different from the frameworks used for VDM and Z. As I hope I have demonstrated above, the ideas have generally stood the test of time and have been general enough to meet unforseen demands. What has emerged is a general theory of specification, design and programming which would appear to be fit for purpose and which is underpinned by 'universal laws'.

The use of (presentations of) theories as units of construction, combined with loose semantics, provides an appropriate level of abstraction for the semantic domain used in

25 Notice that if a different notion of superposition (i.e. of program morphism) had been chosen, *Spec* might not be a functor, i.e. it might not map the program morphisms (of this new category) to specification morphisms. Indeed, the "semantics" of the programming language is more encoded in its morphisms than in the. It is also in this sense that we can talk about the classes of properties that a given notion of superposition preserves. For instance, if guards are allowed to be weakened, then *Spec* as defined above is not a functor because the property ($g \supset B(g)$) is not necessarily preserved by program morphisms. On the other hand, if condition 3 is strengthened, e.g., by replacing the implication by an equivalence, as for spectative morphisms, *Spec* may be more ambitious, for instance by abstracting liveness properties from programs (regulative superposition only preserves safety properties). Hence, it is in the preservation of morphisms that the "correctness" of the functor as a mapping between formalisms lies. Of course, if there is no functor, this may be because the 'willingness' for program components to cooperate is not reflected in the corresponding notion for specifications.

design theories[26]. The missing framework of operations over this domain is provided by category theory, which focuses on interpretations between theories as the appropriate relationship in terms of which the structure of specifications may be analysed. Extensions are the special cases of interpretations defined by injections. A particular kind of extension, the so called conservative extension, turns out to be the essence of modularity in design. The particular combination of conservative extension and interpretation used to underpin implementation composition and parameter instantiation requires that a certain universal construction (reflected in the Modularisation Property) is supported in the corresponding category of specifications and interpretations. The property of the formalism corresponding to this construction is a version of the well known meta property of logics, called the Craig Interpolation Property. As one studies the literature on specification theory and theoretical computing, one notices more and more the occurrence of interpolation properties and their role in explaining phenomena which are related clearly to modularity.

Some of the technical tools adopted in the early work (extralogical equality, subsorting and relativisation) turned out to be less fundamental, in that their use could be internalised via appropriate (inessential) extensions to the framework. Thus, from a scientific point of view, these techniques are unnecessary and avoidable. However, it would appear that, for all intents and purposes, they are still a required engineering tool, avoiding 'clutter' in design.

The emergence of the categorical framework for specification prompted us to demonstrate the ubiquity of another principle, originally put forward by Goguen, what might be called 'the widget principle'. This proposes that systems may be built from components by using colimits of diagrams of components. The assertion that this works for specifications in various formalisms (and ,further, that frameworks for formalisms, such as VDM, B and Z, which were not originally envisaged as part of such a framework, can be straightforwardly adapted to the framework) now has extensive evidence to support it. More surprisingly, it also works for program construction in some interesting languages (ones likely to be of greater use in mobile, distributed systems). This widget principle should, therefore, be raised (no pun intended) to a universal law of specification. The principle and the categorical framework also allow us to relate design to programming and, more generally, frameworks based on different formalisms which, nevertheless, must be used together in the construction of a single system.

Acknowledgements and disclaimers: Many people have contributed to the work outlined above, some with seminal ideas. I would like to thank particularly José Fiadeiro, Martin Sadler and Paulo Veloso. I would like to thank also Ed Ashcroft, Juan Biccaregui, Roberto Lins de Carvalho, Paulo Cunha, Antonio Furtado, Armando Haeberer, Samit Khosla, Kevin Lano, Carlos Lucena, Mike Levy, Tarcisio Pequeno, (the late) Atendolfo Pereda, Doug Smith, Sheila Veloso, and Eric Wagner and others. Although the ideas in this paper are mine, the mistakes are of course theirs!

References

1. M.Abadi and L.Lamport, "Composing Specifications", *ACM TOPLAS* 15(1), 1993, 73-132.

26 See [47,48] for work which is in very much the same spirit.

2. R.Back and R.Kurki-Suonio, "Distributed Cooperation with Action Systems", *ACM TOPLAS* 10(4), 1988, 513-554.
3. J.A.Bergstra, J.Heering and P.Klint, "Module Algebra", *J.ACM* 37(2), 1990, 335-372.
4. M.Bidoit, R.Hennicker and M.Wirsing, "Behavioural and Abstractor Specifications", *Science of Computer Programming* 25(2-3), 1995, 149-186.
5. L.Bougé and N.Francez, "A Compositional Approach to Superimposition", in *Proc. 15th ACM Symposium on Principles of Programming Languages*", ACM Press 1988, 240-249.
6. M.Broy, and M.Wirsing, "Partial abstract data types", *Acta Informatica* 18(1), 1982, 47-64.
7. R.Burstall and J.Goguen, "Putting Theories together to make Specifications", in R.Reddy (ed) *Proc Fifth International Joint Conference on Artificial Intelligence*, 1977, 1045-1058.
8. R.L.Carvalho, T.S.E.Maibaum, T.H.C.Pequeno, A.A.Pereda and P.A.S.Veloso, "A Model Theoretic Approach to the Semantics of Data Types and Structures", in *Proc. International Computer Symposium*, Feng Chia University, Taiwan, December 1982.
9. K.Chandy and J.Misra, *Parallel Program Design - A Foundation*, Addison-Wesley 1988.
10. R.Diaconescu, J.Goguen and P.Stefaneas, "Logical Support for Modularisation", in H.Huet and G.Plotkin (eds) *Proc. 2nd BRA Logical Frameworks Workshop*, Edinburgh 1991.
11. T.Dimitrakos, *A Formal Theory for (Computer Aided) Information Engineering*, PhD dissertation, University of London, 1997, in preparation.
12. H.Ehrig and G.Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Springer-Verlag 1985.
13. H.B.Enderton, *A Mathematical Introduction to Logic*. Academic Press; New York 1974.
14. H.-D.Ehrich, "On the theory of specification, implementation and parameterization of abstract data types", *J. ACM* 29(1), 1982, 206-227.
15. J.Fiadeiro, "On the Emergence of Properties in Component-Based Systems", in M.Wirsing and M.Nivat (eds) *AMAST'96*, LNCS 1101, Springer-Verlag 1996, 421-443.
16. J.Fiadeiro, A.Lopes and T.Maibaum, "Synthesising Interconnections", in D.Smith and J.P.Finance (eds) *Proc. IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, Chapman Hall, in print.
17. J.Fiadeiro and T.Maibaum, "Temporal Theories as Modularisation Units for Concurrent System Specification", *Formal Aspects of Computing* 4(3), 1992, 239-272.
18. J.Fiadeiro and T.Maibaum, "Generalising Interpretations between Theories in the Context of (π-)institutions", in G.Burn, S.Gay and M.Ryan, eds., *Theory and Formal Methods 1993*, Springer-Verlag Workshops in Computing, 1993, 126-147.
19. J.Fiadeiro and T.Maibaum, "Interconnecting Formalisms: supporting modularity, reuse and incrementality", in G.E.Kaiser (ed) *Proc. 3rd Symposium on Foundations of Software Engineering*, ACM Press 1995, 72-80.
20. J.Fiadeiro and T.Maibaum, "A Mathematical Toolbox for the Software Architect", in J.Kramer and A.Wolf (eds)*Proc. 8th International Workshop on Software Specification and Design*, IEEE Computer Society Press 1996, 46-55.
21. J.Fiadeiro and T.Maibaum, "Design Structures for Object-Based Systems", in S.Goldsack and S.Kent (eds) *Formal Methods in Object Technology*, Sringer-Verlag, in print.
22. J.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming*, in print
23. N.Francez and I.Forman, "Superimposition for Interacting Processes", in *CONCUR'90*, LNCS 458, Springer-Verlag 1990, 230-245.
24. N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley 1996.
25. J.Goguen, "Categorical Foundations for General Systems Theory", in F.Pichler and R,Trappl (eds) *Advances in Cybernetics and Systems Research*, Transcripta Books 1973, 121-130.
26. J.Goguen and R.Burstall, "Institutions: Abstract Model Theory for Specification and Programming", *Journal of the ACM* 39(1), 1992, 95-146.
27. J.Goguen and S.Ginali, "A Categorical Approach to General Systems Theory", in G.Klir (ed) *Applied General Systems Research*, Plenum 1978, 257-270.
28. R.Goldblatt, *Logics of Time and Computation*, CSLI 1987.
29. J.A.Goguen, J.W.Thatcher, and E.G.Wagner, "An initial algebra approach to the specification, correctness and implementation of abstract data types", in R.T.Yeh, ed., *Current Trends in Programming Methodology, vol. IV: Data Structuring*, Prentice Hall, Englewood Cliffs 1978.
30. J.V.Guttag, "Abstract data types and the development of data structures", *Comm.Assoc.Comput.Mach.* 20(6), 1977, 396-404.
31. J.V.Guttag and J.J.Horning, "The algebraic specification of abstract data types", *Acta Informatica* 10(1), 1978, 27-52.

32. R.Hennicker and C.Schmitz, "Object-Oriented Implementation of Abstract Data Type Specifications", *proc. 5th Intern.Conf.AMAST'96*, LNCS 1101, 1996, 163-179.

33. S.Katz, "A Superimposition Control Construct for Distributed Systems", *ACM TOPLAS* 15(2), 1993, 337-356.

34. R.Kurki-Suonio and H.Järvinen, "Action System Approach to the Specification and Design of Distributed Systems", in *Proc. 5th Int. Workshop on Software Specification and Design*, IEEE Press 1989, 34-40.

35. B.Liskov and S.Zilles, "Programming with abstract data types", *ACM SIGPLAN Notices* 9(4), 1974, 50-59.

36. Z.Luo, "Program Specification and Data Refinement in Type Theory", *Proc. TAPSOFT'91*, LNCS493, 1991, 143-168.

37. T.S.E.Maibaum, "The role of abstraction in program development", in H.-J.Kugler, ed. *Information Processing '86*, North-Holland, Amsterdam, 1986, 135-142.

38. T.S.E.Maibaum and M.R.Sadler, "Axiomatising Specification Theory", *Proc. 3rd Abstract Data Type Workshop*, Fachbereich Informatik 25, Springer Verlag 1984.

39. T.S.E.Maibaum, M.R.Sadler, and P.A.S.Veloso, "Logical specification and implementation", in M.Joseph and R.Shyamasundar, eds. *Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, Berlin, 1984, 13-30.

40. T.S.E.Maibaum and W.M.Turski, "On what exactly is going on when software is developed step-by-step", *Proc. 7th Intern. Conf. on Software Engin.* IEEE Computer Society, Los Angeles, 1984, 528-533.

41. T.S.E.Maibaum, P.A.S.Veloso, and M.R.Sadler, "A Theory of Abstract Data Types for Program Development: Bridging the Gap?", in H.Ehrig, C.Floyd, M.Nivat and J.Thatcher (eds) *TAPSOFT'85*, LNCS 186, 1985, 214-230.

42. M.C.Meré and P.A.S.Veloso, "Definition-like extensions by sorts", *Bull. IGPL*, **5** (4), 1995, 579-595. {Abstract in *Workshop on Logic, Language, Information and Computation* WoLLIC '94, Recife, 1994.}

43. T.H.C.Pequeno and C.J.P.Lucena, "An approach for data type specification and its use in program verification", *Information Processing Letters* 8(2), 1979, 98-103.

44. P.H.Rodenburg and R.J.vanGlabbeek, "An Interpolation Theorem in Equational Logic", Technical Report CS-R8838, Department of Compuetr Science, Centre for Mathematics and Computer Science, Amsterdam 1988.

45. D.Sannella and A.Tarlecki, "Building Specifications in an Arbitrary Institution", *Information and Control* 76, 1988, 165-210.

46. J.R.Shoenfield, *Mathematical Logic*. Addison-Wesley, Reading 1967.

47. D.R.Smith, "Constructing Specification Morphisms", *Journal of Symbolic Computation* 15(5-6), 1993, 571-606.

48. Y.Srinivas and R.Jüllig, "Specware™: Formal Support for Composing Software", in B.Möller, ed., *Mathematics of Program Construction*, LNCS 947, Springer-Verlag 1995.

49. A.Tarlecki, "Bits and Pieces of the Theory of Institutions", *Proc. Workshop on Categoriy Theory and Computer Science*, LNCS 240, Springer-Verlag 1986.

50. W.M.Turski and T.S.E.Maibaum, *The Specification of Computer Programs*. Addison-Wesley, Wokingham 1987.

51. P.A.S.Veloso, "Yet another cautionary note on conservative extensions: a simple example with a computing flavour", *Bull. EATCS*, 46, 1992, 188-192.

52. P.A.S.Veloso, "From Extensions to Interpretations: Pushout Consistency, Modularity and Interpolation", Tech.Rep.MCC01/95, DI/PUC, to appear in *Information Processing Letters*, 1997.

53. P.A.S.Veloso and T.Maibaum, "On the Modularisation Theorem for Logical Specifications", *Information Processing Letters* 53, 1995, 287-293.

54. P.A.S.Veloso, T.S.E.Maibaum, and M.R.Sadler, "Program development and theory manipulation", in *Proc. 3rd Intern. Workshop on Software Specification and Design*. IEEE Computer Society, Los Angeles, 1985, 228-232.

55. P.A.S.Veloso and S.R.M.Veloso, "Some remarks on conservative extensions: a Socratic dialogue", *Bull. EATCS* 43, 1991, 189-198.

56. P.A.S.Veloso and S.R.M.Veloso, "On conservative and expansive extensions", *O que no faz pensar: Cadernos de Filosofia* 4, 1991, 87-106.

57. M.Wirsing and M.Broy, "A Modular Framework for Specification and Implementation", *Proc TAPSOFT'89*, LNCS 351, 1989.

58. P.Zave and M.Jackson, "Conjunction as Composition", *ACM TOSEM* 2(4), 1993, 371-411.