Proceedings ESEC'89,
ed. C. Ghezzi &. J. McDermid.
Springer-Verlag. 243-259.

Software engineering implications for formal refinement

Alan Dix

Human-Computer Interaction Group and
Department of Computer Science
University of York
YORK, YO1 5DD
0904 430000
alan@uk.ac.york.minster

ABSTRACT

Formal methods are widely proposed as an important part of the software design process, but the design of large systems imposes software engineering constraints on the refinement of these specifications into coded modules. The need to separate the role of system building from the refinement of particular components means that relationships between specification units during the refinement process must be *reified* (that is made into objects) in the software development data-base. The traditional quasi-independent development of system modules can be applied more strongly in the presence of formal specifications, but care must be taken in order to retain the goal of *proportionate effort* between requirements changes and redevelopment cost. Two ways of addressing these requirements are proposed, the presence of *semantic interfaces* between specification components as data-base objects and the use of *shared parameters* to generic specifications to represent shared sub-specification. In both these cases the interface specification forms the *focus of negotiation* for shared design decisions. In addition a higher level structuring concept is introduced, the *collection* which describes the requirements for a set of modules and their inter-relationship.

Keywords - formal specification, refinement, modularisation

1. Introduction

Formal specification is increasingly important in discussions (although not necessarily practice) about system development. Unambiguous description of the high level decomposition of systems is an obvious candidate for their use: the behaviour of individual modules being specified, and the behaviour of the entire system being defined by these specification components and their properties of composition. Once we get to this level however, it is the algebraic properties of specifications as objects that is of prime importance, rather than the nitty gritty of the particular notations chosen. This paper will operate at this level, deliberately not assuming any particular specification technique. There is a basis for this approach at the theoretical level, for instance in the study of *institutions*[1,2] which express many specification building operations in a way independent of the underlying logic and notation.

We will assume that there is some form of (electronic or paper) data-base recording objects in the software development process. Throughout there will be an emphasis on the *reifying* of inter-module relationships and design decisions into entities in the software development data-base. That is concepts that may exist only in the developers mind or embedded within a particular specification are to be lifted out into objects that can be manipulated at the data-base level.

In the rest of the introduction, we shall discuss some of the software engineering issues that have prompted the work in this paper, look at the sorts of building operations available for typical specification languages, and finally examine the properties of refinement operations.

The succeeding three major sections will then deal with the three major results:

- The necessity for semantic interfaces between specifications, and the importance of this as a focus for negotiation.

- Using shared parameters to generic specifications as a means of expressing sharing constraints between sub-specifications.
- The proposal for a high level structuring concept, the collection, describing a set of modules and their inter-relationships.

1.1. Software engineering requirements

The building of medium to large applications places various technical and management requirements. In particular, three of these drive much of the discussion in this paper:

1. *Configuration independent of specification* – proof in the data-base
   The correctness of a particular configuration should be verifiable from the relationships in the data-base alone.
2. *Proportional effort*
   Small changes in requirements should require only small amendments to the final coded system and intermediate objects, **whilst of course retaining correctness.**
3. *Independent work units*
   It should be possible to delegate the responsibility of producing parts of a system to independent individuals or groups.

Looking at the first point, the person responsible for the construction of specific configurations should not be expected to perform proofs or analyses on the particular specification and implementation components. The proper domain of this role is the software development data-base, recording the relationships between software components. So for instance, when the configuration manager wishes to put two components together to make a system variant. It should not be necessary to perform some proof of consistency on these two at this stage. Whether the two components are compatible, and whether, when composed, they have the desired properties should be decideable within the data-base. The relevant facts should already be extant in the data-base having been verified by the person responsible for the specific components. That is, all appropriate objects and relationships should be *reified* into data-base entities† rather than being buried within the text of a particular specification.

Even when the same person is performing both roles, as developer and configuration manager, the confusion of these roles should be avoided. The roles correspond to levels of abstraction in the design process, so confusing the roles confuses these levels of abstraction.

The goal of proportional effort can never be achieved in its entirety, as design decisions which are consistent with the original requirements may be at conflict with changed requirements. The practical issue is one of containing particular design decisions within well defined parts of a system.

For similar reasons work on different parts of a system are rarely totally independent as design decisions which clarify ambiguous, or arbitrary features of the requirements will have an impact on several parts. Again the practical issue here is to make these wide impact decisions at the appropriate level, and to percolate them to the appropriate work units. We wish to avoid the situation where incompatible decisions are made in different parts of the system.

The normal way to address these latter two issues is using the system modularisation. Part of the art of functional decomposition is choosing an appropriate module breakdown. This should both enable design decisions to be made within each module as independently as possible and further foresee possible changes in requirements, isolating the effects to a small number of modules. If the modules are sufficiently independent, then a change of requirements will only result in changes to the modules *directly* affected. In a traditional (non-formal) situation this independence of modules is ensured by informal documentation and well-typed interfaces. Where independence is not preserved the dependencies should be available in the data-base in order to assess the ramifications of change.

---

† the word entity is used loosely here, not assuming any particular data model

1.2. Formal specification and component building operations

Most formal specification techniques include their own structuring and construction mechanisms. These include adding extra types, functions or properties to existing specifications, renaming and name hiding. Their interest is primarily in construction and refinement within the specification domain. When considering the larger software engineering context, we should consider two types of specification object.

1 *Pure specification* - objects introduced in order to facilitate the construction and comprehension of the system of a whole, or within an individual module.

2 *Codeable modules* - specifications that will eventually give rise to codeable objects.

The distinction is one of use, rather than being inherent in the specification text itself (although the intended use is often obvious). For example, a specification of stacks of natural numbers, would probably be expected to eventually correspond to an actual coded module, whereas a specification of what it means for a function to be invertible would probably be used as a packaged constraint.

Within the life-cycle of a system the two types of object will have varying importance. Early on, during requirements definition, we will not be interested in whether a particular specification object represents a codeable entity. However, as soon as we produce a modular decomposition intended to facilitate independent refinement and implementation, we *de facto* assume that the modules are codeable, and that the final system can be built *constructively* from the independently coded components. This pattern may iterate at lower levels during the refinement process of course.

If we intend a certain object to be codeable, then the sort of specification building operations we can use is constrained. The particular set of allowable operations depends somewhat on what are acceptable linkage operations on the final code. If we allow a building operation to syntactically access the code then, for example, it may be permissible for a mathematical package to perform symbolic differentiation of functions in its component modules. For the purposes of this paper however, we will assume that the eventual target system will be constructed using reasonably traditional linkage operations and it is these that should be reflected in the specification building operations.

The appropriate constraint† on the specification building operations is that they should be *persistent*.[3] This is a formal notion which is used, with slight variation, by many specification languages. This can be given a precise definition in terms of category or set theoretical semantics, or in terms of the notation's proof theory. For instance, a rough proof theoretic definition would be: any sentence over the alphabet of A, which is true of A embedded in B is also true when A is considered on its own. In short, it says that if a specification B is built using specification A, then in a sense, A is still there, uncorrupted within B. This is clearly a good pragmatic condition, as when we eventually link together the coded modules, we don't expect the code of the modules to change their semantics. (It often does, but that's usually classed as a bug!). In fact, not only is it useful from a pragmatic viewpoint, but it is also a useful constraint from a formal one as it helps one to reason about the specifications.

Persistence is particularly important when we consider *generic* or *parameterised* specifications. Non-persistence in these situations corresponds to a "hidden" precondition for the application of the generic specification to a particular parameter. It is often demanded[3] or deemed highly desirable[4] that all generic specifications should be persistent. If a use of a specification A within specification B is intended to be persistent, then we can usually achieve the same effect by "lifting" A into a parameter, $B[A]$. This equivalence means that we can deal with the single issue of generic specifications, rather than two mechanisms. It does not imply that the final implementation language supports generic code units. Often a generic unit is only used once, but if not the last implementation step can involve copying and renaming if necessary. Such facilities would make this last stage substantially cleaner however.

More problematic, some specification notations do not address the issue of persistence or controlled generic specifications. For instance, Z's schema calculus[5] provides very rich tools for the incremental building of specifications and expression of requirements, but does not really address this issue. However, experience with other notations suggests that the addition of such features would not be difficult, the freedom of the standard schema language being used within a codeable module (or before the modular decomposition), and the tighter controls applied perhaps at the level of complete Z documents.

---

† In fact, one can be slightly more liberal than this constraint, object oriented systems for example tend to be conservative but not persistent.

1.3. Refinement

One of the reasons persistence is so important, is because of its relationship with refinement. It ensures the important refinement condition that if B is a generic component and if A' refines A then $B[A']$ refines $B[A]$. This is known as *horizontal* composition.[6]

In fact, the reason why this condition holds is not really important to the configuration manager, merely that the refinement relation obeys certain algebraic laws. The *vertical* composition condition,[6] is simply that the refinement relation is transitive (if A'' refines A' and A' refines A, then A'' also refines A). In addition, the horizontal composition law has a more general form, namely for any building operation **C**:

$$(\forall i \quad A'_i \text{ refines } A_i) \quad \Rightarrow \quad \mathbf{C}[\![A'_1, A'_2 ..]\!] \text{ refines } \mathbf{C}[\![A_1, A_2 ..]\!]$$

In particular, if B and B' are generic components, and B' refines B, then $B'[A]$ refines $B[A]$.

There are many possible reasons for a refinement step. It may reflect an actual behavioural refinement, reflecting a design choice about the final system or it may be a functional refinement, moving towards a computable form or data representation. As we get closer to the code level, the possibilities are greater again, pragmatic issues, such as machine level representation, paradigmatic ones such as language choice or garbage collection strategy, and eventually, the generated relationship between source and compiled code. Verifying, that the refinement condition is met may require extensive proof at the specification level, or may be a side-effect of the construction process, as with transformation systems or compilers. Whatever the reason and low-level verification conditions however, when it comes to system construction, the algebraic composition laws are all that is needed.

We saw previously that the requirements of proof in the data-base must be met by reifying all appropriate objects and relationships into data-base entities. The most important relationship will probably be the refinement or implementation relationships. We will use the symbol **sat** (satisfies), to denote this relationship, as it is less loaded than the terms "refines" or "implements", although in the text the word "refinement" will be mostly used. Thus A **sat** B holds whether A is the compiled form of B, or is simply a more precise abstract specification than B.

1.4. Notation

We have already encountered most of the notation that will be used. The discussion takes place in the context of a software development data-base, referred to usually as simply "the data-base". Within this the principal entities are specification (or code, the distinction being unimportant) modules. Module names are italicised with initial capitals (A, *Compiler* etc.) and may be decorated (A', B^*).

The relation of refinement, or satisfaction, as we have said above, is denoted by: A **sat** B

Generic modules are slightly more complex. A module B which is parameterised over all specifications satisfying conditions in I is denoted: $B[a{:}I]$. Its instantiation by a module A which satisfies I is $B[A]$.

Because the parameter specification I and the module used A are often both specifications, we can get the situation where a module A can be used in both roles $B[a{:}A]$ and $B[A]$. The lower case a is used as a formal parameter name in order to distinguish these two cases.

Lower case letters are also used as "local" names for specifications to express sharing (or lack of it), so that

$$C[x, x, y] \quad \textbf{where } x = A,\ y = A$$

is an instantiation of a generic module C where all the parameters are copies of A, but the first two are the *same* copy of A, the last one being different. This will become more clear when we discuss sharing in §3.

2. Semantic interfaces - orthogonal development

This section looks primarily at the development of pairs of modules, where one makes use of the other. The next major section will consider more complex cases.

2.1. Independent refinement

Let us assume that we have decided on a top level modular breakdown of a system, and let's look at two specification components A and B. B uses functions and types from A. We now start the refinement process, A is refined to A' and B is refined to B', further refinement then takes place to A'' and B''. The question is, can the refinement step from B' to B'' make use of the knowledge in A'. This would be perfectly acceptable from the point of view of correctness, but it violates the independence of the modules.

Its impact will be in terms of proportionate effort. Consider a change in requirements that leads from A to A^*, but which leaves B unchanged. We develop a new refinement sequence for this to A'^* and A''^*. This is an acceptable effort so far, affecting only the module whose requirements have changed. Unfortunately, if the development of B'' used the information in A' then it may not be valid with the new specification A'^*, requiring reworking along the B stream also. Of course, these effects can snowball, in the worst case requiring the reworking of the entire system! Even when there is no change in requirements, the development team for A would be "locked" into the decisions made at the stage $A \rightarrow A'$. We should thus see the specification A as an *interface specification*, and only allow the development of B to know about it.

2.2. Semantic interfaces

Of course, the above slightly glossed over the fact that even when the B stream uses A as its interface specification, it can only accommodate changes in requirements for the A stream which are in some way upwardly compatible with the original specification. In order, to be more flexible, a slightly stronger restriction can be made. A second specification I_{BA}, the interface specification for B's use of A can be defined. That is we propose that specification modules have *semantic interfaces* between them. A will be a refinement of I_{BA}, but I_{BA} will be chosen so that likely changes in A will still satisfy I_{BA}. Of course, this cannot be guaranteed, but that is precisely the art of functional decomposition. The interface specification may be the same for all users of a module, or may be different for different users.

Again, in order to make discussion easier, we will assume that the use of A by B is lifted into an instantiation of a generic component. That is wish to develop the system specified by $B[A]$, where B is a generic specification and I_{BA} is its parameter specification ($B[a:I_{BA}]$). The B stream of the development process, must produce a generic component satisfying $B[a:I_{BA}]$. This now brings violations of independence to the surface. If B's development team attempt to make use of information in A, or later in the development process A', this is clear, as they now only satisfy the specification $B[a:A']$, which is a partial instantiation of $B[a:I_{BA}]$ and therefore does not properly implement it.†

If this interface is preserved we obtain a high level of flexibility to changes in requirements and independence of work streams. This is true orthogonal development: refinement of modules proceeds independently, and changes in requirements to modules can be processed independently.

2.3. Non-orthogonal development

It may be, that this high level of independence hinders the efficiency of a particular development stream unacceptably: the fire walls have to be broken slightly. We discuss elsewhere[7] where this is necessary, and how this process of *non-orthogonal development* can be aided, in particular a method of introducing controlled structural change is suggested there, called *interface drift*. It suffices here to say that non-orthogonal development should be avoided or at least put off until as far down the refinement sequence as possible, but that it is sometimes necessary.

Assuming this does occur however, and the A stream developers are prepared to comit themselves to some design choices, it is not sufficient for them to simply chuck a few axioms over to the B stream. The B stream have contracted to produce $B[a:I_{BA}]$, this contract must be renegotiated, to reflect the extra information in the interface: $B[a:I_{BA}']$ say. However, typically neither the B stream nor the A stream will be the owner of the interface specification, rather the owner will be the team responsible for the modular decomposition. This is entirely right, as this interface was designed in order to accommodate possible requirements changes. The interface specification thus forms the *focus for negotiation* between the various interested parties.

---

† This is the familiar contravariance property for parameter types.

2.4. Reifying interface specifications

This interface specification is clearly an important entity in the refinement data-base, and must therefore be reified into it. The interfaces between components, and whether a component satisfies the appropriate interface specification should be recorded in the data-base, so that the configuration manager can assess the correctness of a step without resort to the specifications themselves. Specification notations do not always make this easy. They are often described such that the burden of proof for the correctness of an instantiation step is at the time of instantiation.[3] This is not usually inherent in the language, but is an important methodological point.

A typical data-base might contain the following information.

$B\,[a{:}I_{BA}]$			B is generic with parameter specification I_{BA}
A	**sat**	I_{BA}	A is a specification satisfying I_{BA}
Sys	==	$B\,[A]$	Sys (the target system) is B instantiated with A
			this is a legal instantiation as A satisfies I_{BA}
A'	**sat**	A	A' is a refinement of A
A''	**sat**	A'	
$I_{BA}{}'$	**sat**	I_{BA}	$I_{BA}{}'$ refines the parameter specification I_{BA}
A''	**sat**	$I_{BA}{}'$	and A'' satisfies this as well as I_{BA}
B'	**sat**	$B\,[a{:}I_{BA}{}']$	B' is a refinement of B using extra parameter information

From this information, the configuration manager can tell that $Sys_1 = B'[A'']$ is a valid implementation of Sys, but not $Sys_2 = B'[A']$ as A' does not satisfy the parameter specification $I_{BA}{}'$. The refinement of the parameter specification to $I_{BA}{}'$ is the relevant point for negotiation between interested parties. No knowledge of the contents of any of these specifications is necessary, merely the data-base relationships.

3. Sub-structure sharing using generics

A problem that arises in most specification notations, is that of sub-structure sharing, or to put it more bluntly, naming. If two specifications C and B both refer to the specification A, is it the *same* specification that is intended or do we mean *copies*? There is clearly no "right" solution and different notations use different rules, for instance Clear assumes default sharing[4] (explicit copying can get round this) as does ACT-ONE,[3] whereas ML modules require explicit sharing.[8]

When the specifications refer to mutable objects (as with say VDM or Z)[9,5] this issue is more complicated, as we also have the issue of whether (for example) two stacks of integers represent the same *run-time* object. Here however, we are primarily interested in whether they have the same *semantics* (when considered each in isolation). In fact, the techniques outlined below can also be used to deal with this related problem rather neatly, but it is shared semantics that is of primary interest.

We might think that name hiding removes most of the possibilities for unintentional sharing. There is however the possibility of semantic leakage, for instance imagine we have two circular buffers defined; one of integers, and the other of strings. Both buffers are defined using the same specification of modulo arithmetic where the modulus base is left intentionally undefined. This specification is not present in the export list of the buffer specifications. However, if we assume that the two buffers share the same sub-specification, then we know that they will eventually share the same coded form of modulo arithmetic, with the same base. Thus if we use the two buffers with a similar pattern, we may omit an overflow check for one of them. If on the other hand we assume copies, we require both checks as different versions of modulo arithmetic with different bases may be used.

However, when we use semantic interfaces between our specifications, life is easier. If the shared sub-specification is not mentioned in the interfaces, then knowledge about it is not available to the user of the buffers. Hence there is no semantic leakage. Thus semantic interfaces restrict the range of the problem, but do not remove it entirely.

3.1. Implication of sharing on refinement

To asses the impact of sharing upon the refinement process, we consider an example of two modules with a shared sub-specification.

A system includes two specification modules B and C which both use a specification *Complex_numbers*.

Moreover, this specification is present in their interfaces and shared. The need for the shared sub-specification is apparent, as C contains a function f returning a complex number and B has another function g with a complex number parameter and the system as a whole makes use of compositions of the form $g(f(x))$.

C and B are then given to two programmers to refine and implement. C's implementor decides to represent complex numbers by real and imaginary parts. B's implementor represents them by size and direction. Both implementors have faithfully implemented their specifications, but the resulting modules cannot be included together in the final system without modification. The solution may involve frequent costly conversions, or recoding of B or C. Some typing systems may not even detect this inconsistency, leading to even worse problems.

In general, if the initial shared specification is ground, (that is defines all operations completely) then it may be possible to integrate such modules using suitable data conversion. If the original shared specification is not ground then the functional design decisions taken during refinement of C and B may be totally incompatible. In either case, we wish to be warned about such situations or prevent them entirely. Clearly, the issue of sharing is not local to the client specification, but must be explicit in the data-base and in the specifications given to the individual implementors.

3.2. Generics for sharing

We could introduce a new mechanism for representing shared sub-specifications, however we can handle most cases using the existing mechanism of generic specifications. Most shared sub-specifications will be persistent and can therefore be lifted into an instantiation of a generic component. So for instance, if B and C have a shared sub-specification A, we deliberately turn B and C into generic specifications $B[a:I_{BA}]$ and $C[a:I_{CA}]$. The jobs of the teams for B and C are to refine these generic specifications. The implementation choices for A are the concern of a separate development team, and cannot therefore be made inconsistently by the B and C teams. The sharing, or otherwise of the sub-specification is made explicit in the system construction: *a shared parameter represents shared substructure.*

Again this neat separation is fine in theory, but in practice either the B or C development teams may need to know information about A's refinement in order to achieve efficiency goals. Again, their interfaces with A become the focus of negotiation, and the decision is made at the appropriate level.

3.3. Example of generics for sharing

We will develop the above example slightly. To simplify matters, we assume that $I_{BA} = I_{CA} = A$

$B[a:A]$			B is generic with parameter specification A
$C[a:A]$			C ditto
(i)† S	==	$Sys[B[x], C[x]]$ **where** $x = A$	system with shared A
(ii) S'	==	$Sys[B[y], C[z]]$ **where** $y = A, z = A$	system with no sharing (two copies of A).
A'	sat	A	
A^*	sat	A	two possible design choices for A
B'	sat	$B[a:A]$	
B^*	sat	$B'[a:A^*]$	B^* refines B' using the design choice A^*
B''	sat	$B'[a:A']$	B'' ditto, except using A'
C'	sat	$C[a:A']$	C' also uses design choice A'
S_1	==	$Sys[B'[A], C'[A]]$	possible systems to refine S and S'
S_2	==	$Sys[B'[A], C'[A']]$	
S_3	==	$Sys[B^*[A^*], C'[A']]$	
S_4	==	$Sys[B'[A'], C'[A']]$	

S_5 == $Sys[B''[A'], C'[A']]$

The specification S_1 is illegal as the instantiation of C' is not by a refinement of A'.
S_2 and S_3 are both legal and satisfy S'. However, they do not satisfy S as they violate the sharing constraint. S_3 corresponds exactly to the case of incompatible representation choices mentioned earlier.
S_4 is legal and satisfies both S and S', as does S_5. However S_5 is probably more efficient given B''''s better knowledge of its parameter.

In addition, none of B^*, B'', C' satisfy the "contractual" specifications B and C, being partial instantiations, rather than straight refinements. They would all need to be referred back to the module interface level before being allowable objects in the refinement stream. It would of course have been perfectly reasonable for the individual teams to have experimented with different refinements to A on their own, but they would have been in no doubt that this was not an acceptable part of the refinement process without further negotiation.

It should be emphasised again, that all the above judgements were made solely on the contents of the database. The proofs (or justifications!) of the particular refinement relations must be provided by the development teams.

4. Higher level structure - collections

If we assume a coded module size of between 100 and 1000 lines of code it is clear that any medium to large system will contain hundreds if not thousands of modules. If you then take account of all the specification documents, possibly at several levels of refinement, some sort of structuring is necessary from a human management point of view. There may be some sort of semi-hierarchical structure with a top-level decomposition into perhaps a dozen or so modules, which themselves make use of other modules etc.[10]

Is it sufficient for this structure to be informal, or is there need for a formal higher level structuring concept? By looking at an example where one company contracts out work to another it will become clear that such formal structures are necessary. We will call this structure a *collection*. In any large project many such internal contracts will be required hence collections should be a normal part of the refinement database.

4.1. Example - the Lisp collection

Consider a leading software firm, Environments Plc who specialise in building state of the art programming environments. They wish to produce a window-based lisp environment, but are primarily interested in the interface aspects, and wish to sub-contract the development of the underlying lisp system. They formalise their requirements for this system into a set of specifications:

Sexp	abstract descriptions of S-expressions
Interpreter [*s*:*Sexp*]	semantics of lisp interpreter
Object	object code for compiler
Compiler [*s*:*Sexp* , *o*:*Object*]	the lisp compiler
Executor [*s*:*Sexp* , *o*:*Object*]	execution tools for compiled code

Clearly the semantics of the lisp interpreter are independent of the particular representation of S-expressions chosen, hence this is developed as a generic specification. Likewise for the compiler and executor.

Environments Plc approach a firm specialising in lisp systems, Lisp Inc, with their requirements. Lisp Inc will be asked to produce five coded modules: $< S, I, O, C, E >$, satisfying the following conditions:

S	**sat**	*Sexp*
I	**sat**	*Interpreter* [*S*]
O	**sat**	*Object*

† In practice only one of S or S' would be present as the *internal* correctness of *Sys* would depend on the sharing constraint, we should imagine (i) and (ii) as being two possible states of the data-base.

C **sat** *Compiler* [S , O]
E **sat** *Executor* [S , O]

That is, they will decide upon a representation for S-expressions and object code (S and O). They will supply a lisp interpreter that satisfies the semantics in *Interpreter*, but which uses the chosen representation S. Similarly the compiler and executor must both use the *same* representation of S and O. The sharing constraints are necessary, as it would be sad to have a compiler and executor that didn't have the same understanding for the object code!

It would not be suitable to ask for a single module which includes all the functionality, as the different modules will be required in different contexts. For instance, Environments Plc may want to develop a structure editor for S-expressions, this will use S but none of the other modules. An introductory level lisp system may need the interpreter only, and likewise a stand alone compiler would only need C, with E the executor supplying run-time support. A fully integrated environment may use all the modules.

4.2. Collections

In this simple case, the requirements could easily be given as an informal document, but it is clear that as systems scale up, this will become unacceptable. The requirements for such a set of modules with their inter-relationships needs to be treated as a formal entity in its own right. I will call such an entity a *collection.*

A simple collection will have three elements:

- *Imports* - A set of specifications supplied by the client, that will be used to specify the requirements.
- *Exports* - A set of module names that will constitute the delivered product.
- *Constraints* - A set of satisfaction relations giving the requirements for each module in the export set, including sharing constraints.

The collection then becomes the contractual entity between client and supplier. It is also a likely unit for the high level decomposition of a system.

4.3. Relation to existing mechanisms

It should be noted that the set of modules specified in a collection has identical properties to those of a set of modules suitable for instantiating a generic specification. For instance, in the earlier example of shared subspecifications, $S = Sys[B[A], C[A]]$ requires a collection $< a, b, c >$ satisfying:

a **sat** A
b **sat** $B[a]$
c **sat** $C[a]$

Thus collections and their correctness can be handled using the same mechanisms as the correctness of generic instantiation. Alternatively, we could regard collections as being the primary concept, and regard generic modules as parameterised over a single collection which encapsulates the parameter sharing constraints. Either way we have a single conceptual mechanism.

4.4. Non-parametric sharing constraints

Earlier, we said that most sharing can be represented as shared parameters to generics. However, the lisp collection above suggests a situation that requires a slightly more complex constraint.

In formulating the semantics of the interpreter, compiler and executor, reference will almost certainly be made to a single description of the semantics of the chosen dialect of lisp. Let's call the specification of this *Lisp_semantics*. There are some constructs that are typically left undefined in standards documents such as `car(nil)`. Environments Plc, may choose to define the semantics of these constructs explicitly in *Lisp_semantics*, leaving no implementation choices for the developers. However, it is more likely that they would prefer to leave this to Lisp Inc, as they are likely to have a better knowledge of the likely performance ramifications of these rather arbitrary choices. What Environments Plc do want to ensure is that both the interpreter and compiler make the *same* choices.

There are several ways to go about this:

1. An additional constraint specification *CS* can be introduced with axioms like

   $$\forall f, s \quad interpreter\_apply(f, s) = executor\_apply(compile(f), s)$$

   The specification *CS* would be parameterised over possible interpreters, but it would not be a true generic specification as it would constrain its parameters beyond their parameter specifications. The collection would then include an extra constraint condition of the form:

   $$CS[I, C, E]$$

   The implication being that the axioms in *CS* are satisfied for the actual modules delivered.

2. An additional deliverable *L* could be asked for. This would be a refinement of *Lisp_semantics* which resolves unspecified features in it. Lisp Inc would be free to choose whatever refinement was most advantageous from the point of view of performance, simplicity etc. In addition, it would be included as an additional requirements "parameter" to *Interpreter*, *Compiler* and *Executor*

   L **sat** *Lisp_semantics*
   I **sat** *Interpreter* $[S \parallel L]$
   .....

   This would differ from the other deliverables in that it would not be expected to be a coded module, merely a ground specification. It could be used for documentation purposes, or if we don't want the eventual users to make use of such soft features, we might merely ask that it exists without actually requiring it to be handed over. Similarly its use as a "parameter" differs from a normal one (hence the "||" separating it from the standard parameters). It is not used as a component to be built into the resulting specification, but instead is a constraint on the result. Because of this it obeys different algebraic laws from a normal parameter and is not expected to be persistent.

3. In a similar fashion to the above, one could require *L* as a deliverable, but instead of including it as a parameter to *Interpreter* it could be included as an extra satisfaction condition:

   $Sem[I]$ **sat** L
   .....

   Here *Sem* is assumed to be a generic specification that extracts the abstract semantics from the concrete interpreter. This implies that I (also C and E) will have to be designed to simultaneously satisfy two constraints.

Different constraints may find their expression most naturally using one or other of the above methods. In general however the latter two are to preferred over the former as they both reify the design decision into the data-base entity *L*.

4.5. Higher order collections

In the same way that a module describes the properties of a set of types and functions, a collection describes the properties of a set of modules. In the lisp collection the client might have wanted to remain uncomitted over the choice of S-expression representation and would therefore ask for a set of modules parameterised over this choice. That is, the collection requested would be generic over a module parameter. It is not hard to imagine requirements for collections generic over collection parameters. Hence, when considered at the level of their data-base relations, collections have similar properties to modules. It would not be unreasonable then to imagine collections of collections etc. This sort of generality would not be hard to achieve, but whether there is any great advantage over first order collections is unclear.

5. Conclusions

We have argued throughout that all the relevant information for system building should be reified in the refinement data-base. In particular, specifications corresponding to design decisions, particularly shared decisions, should be entities in this data-base.

Modules must have semantic interfaces between them, specifications limiting knowledge of each other, in

order to allow separate development. Generic modules are not just useful for reuse, but are an important structuring mechanism in specifications.

Expressing all sharing as the sharing of parameters to generic specifications both expresses this typically thorny problem in a succinct way using an existing mechanism, and also promotes the knowledge to the level of talking about specifications as objects. Problematical design steps are highlighted as instantiations of generic parameters. In general the interface specification is the point of negotiation between parties in the development process.

A higher level structuring unit, the collection, has been proposed that describes a set of module requirements and their inter-relations. This can form the contractual unit where the normal module is too small and is likely to be the natural unit of higher level functional decomposition.

Acknowledgements

This work was carried out under a SERC post-doctoral fellowship and is the fruit of innumerable discussions with various members of the Computer Science Dept. at York.

References

1. D. Sannella and A. Tarlecki, "Specifications in an arbitrary institution", CSR-184-85, University of Edinburgh, Dept. of Computer Science (March 1985).
2. D.T. Sannella and A. Tarlecki, "Extended ML: an institution-independent framework for formal program development", in *Proc. Workshop on Category Theory and Computer Programming*, Springer (1986).
3. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1*, Springer-Verlag (1985).
4. D.T. Sannella, "Semantics, implementation and pragmatics of Clear, a program specification language", CST-17-82, PhD thesis, University of Edinburgh (1982).
5. C. C. Morgan, *The schema language*, Oxford, Programming Research Group (1985).
6. D.T. Sannella and A. Tarlecki, "Toward formal development of programs from algebraic specifications: implementations revisited", in *Proc. 12th Colloq on Trees in Algebra and Programming*, Springer (1987).
7. A.J. Dix and M.D. Harrison, "Interactive systems design and formal development are incompatible?", in *Proceedings 1988 Refinement Workshop*, ed. J McDermid, (to be published Butterworth Scientific) (1989).
8. D.B. MacQueen, "Modules for standard ML", pp. 198-207 in *Proc. 1984 ACM Symp. on Lisp and Functional Programming* (1985).
9. D. Bjorner and C. B. Jones, "The Vienna Development Method: The Meta-Language", *Lecture Notes in Computer Science*(61) (1978).
10. D.L. Parnas, P.C. Clements and D.M. Weiss, "The modular structure of complex systems", pp. 408-417 in *7th International Conference on Software Engineering* (1984).