Layered Specifications to Support Reusability and Integratibility

Roland T. Mittermeir and Evelin Kofler

Institut für Informatik
Universität Klagenfurt
AUSTRIA

Abstract

Building systems from prefabricated parts - be it called "by components integration" or "by reuse" - differs from classical software design. The designer is not free to define the system breakdown structure strictly in a top-down way. the methodology has to allow for postponing certain interface decisions to a rather late point in time.

This postponement is permissible only, though, if other aspects are very precisely specified. Having the right model of specification of both, the target (system) and source (component) will substantially aid the retrieval and integration of candidate components.

This paper argues for a layered approach towards system specification. It shows, how relational specifications will help system designers not to bind themselves too early into premature decisions and how designs aiming for heavy reuse can grow by enriching specifications stepwise.

KEYWORDS: *Software Reuse, Software Integration, Software Specification*

1. Introduction

1.1. Motivation and overview

These days, it is evident, why one wants to reuse software to enhance software production productivity and, at the same time, software quality. Comprehensive presentations of the topic of software reuse can be found in [BIGG 89], [TRAC 88], or [FREE 87]. However, it has also been pointed out very clearly that it is much harder to achieve the aims of software reuse than one would ever believe (e.g. [BIGG 87] or various case study reports in Vol. II of [BIGG 89] or in [TRAC 88] [FREE 87]).

The problems with bringing software reuse into practice is partly due to lack of adequate means for describing software properly (even if progress has been made in this area by classification schemes [PRIE 87], [ROSS 89], [MITT 87a], and by advances in software specification). Another part of the problem is attributable to the fact that current software design methodologies are insufficient to guide software developers towards designs which are intrinsically more general than the application they are currently working on. These technical reasons are very important hurdles for developping software systems by integration of prefabricated components. Other extremely important inhibitors for software reuse, such as organizational and psychological factors also need to be put into consideration. However, they are not dealt with in this paper.

In the following subsection, we will briefly focus on the technical presumptions of our approach, crossing the borderline between "Design for Reuse" and "Design with Reuse". Further, the "Shopping Paradigm" for scripting software systems will be motivated. Section two describes the intertwining of reuse- and integration potential with methodological and specification aspects. On these premises, we present the mathematical basis of our approach and we will show, how proxi-matches can be defined with relational software specification. This constitutes the basis for the presentation of layered specifications and their impact on the development of systems by integration of readymade components. Finally, we will compare this approach to other well known specification models.

1.2. Premises

We agree with B. Meyer [MEYE 87] and with most other authorities on software reuse that object-orientedness would be an important key towards success. If we combine an object-oriented approach on the

0-8186-2697-6/92 $03.00 © 1992 IEEE

components level with a scripting approach on the systems level [KAPP 89], [NIER 90], [NIER 91] we would end up with a perfect environment for prototyping and for application development by component integration.

Naively, one might hope that adding the necessary methodological guidelines to object orientedness would solve the major, if not all problems of current software engineering or at least of system development based on software reuse.

Unfortunately, this is not the case. Not only, that we are just lacking adequately populated libraries of reusable objects, not only, that object-oriented languages have not yet really found their way out into the business world. We are usually no longer in a situation where we can build new applications from scratch. We have to build applications on top of existing applications, have them interface (often quite tightly) with existing applications, or consider our job just to be tied to upgrading, i.e. modifying existing applications. In this realm, there is little room for object-orientedness - and in fact, prima vista little room for controlled reuse, unless - as already pointed out by Matsumoto [MATS 83] - special effort is invested to raise the level of abstraction (and quality) of a component to make it reusable in a new project.

The notion that special predevelopment efforts have to be spent with a component to make it reusable and Basili's statement [BASI 90] that software maintenance and software reuse are two converging challenges, sheds new light on software development. If we are to integrate reusable components in a software factory as envisioned in [CALD 91] we have to give up the basic notion of the top down paradigm which dominated software engineering methodologies for over twenty years.

We can no longer start from some highly abstract but all-encompassing requirements statement and add design- (and implementation) decisions until the final, executable, and hopefully wanted system is reached. Neither would prototyping approaches - which are inherently little more than the rapid passages through various small waterfalls - be adequate. We have to come to a development model, where we start out with a system which we probably do not want to have, but which can be changed in a disciplined way, until we arrive at a (Note: not "the") solution which satisfies our (the user's) needs.

While this might sound like hacking and, therefore, infeasible for large scale systems development, it will be neither hacking nor infeasible, if we have a sound theoretical foundation and a sound methodology on which this approach can be based. The theoretical basis needs to accommodate incompleteness and must neither depend on scale nor on direction of system development.

2. Specification-aspects and reuse-potential

2.1. Building systems according to specifications

Current Software-Engineering wisdom calls for a top-down process to develop new software systems. The ideal of this process is, that the system is constructed in a stepwise decomposition process in such a way, that the previous steps must not explicitely build on knowledge of later steps, i.e. the higher level strata are developed in well conceived ignorance of later design decisions and of design details introduced later.

Challenges to this notion may come already from the Structured-Design principle of maximizing (optimizing) the fan-in of subordinate modules [YOUR 79]. It further comes from the empirical studies conducted by Curtis et al [CURT 89], which have shown that experienced programmers do (briefly) delve into very low level concepts even during the most high level design tasks. - Why are they doing it? Because they need to anticipate the consequences of their decisions and very informally check whether they can support these consequences before they proceed in their high level task. But this delving down happens only informally and invisibly. Formally, design at a given level of system decomposition happens as a black-box activity with respect to all lower level components, but as a white-box activity with respect to all components between the current focus and the top level specification.

From a reuse-perspective this has to be considered harmful, because it causes the definition (and freezing) of boundaries and interfaces without having a chance to benefit from components which exist already in some software repository, waiting to be reused and integrated into a new system. From a reuse-perspective it would be desirable to come up only with a relatively preliminary design (like the unpronouncable sketch in the experienced designers head), check it against the software repository (as the experienced designer does, when silently looking inward at past low-level experience) and then continue by refining the system and solidifying design decisions (notably interface-related decisions) under consideration of the most favourable set of components of the repository utilizable for the task at hand.

To allow such a design approach we need on one hand a mechanism to formally and precisely (!) express vagueness (cf. [WEIN 79], who asked already in 1979 "why not" to "draw it with a wiggle"), and on the other hand also some general guidance as to what kind of components might be found in the repository.

2.2. Shopping paradigm - informal description

The term "shopping paradigm" comes from my experience when buying a new outfit. Usually, I do not specify the detailed look of the buttons of my new suit, nor the exact design of the tie, nor the depth of the pockets in the trousers. But I do know that the suit should be bearable at a given set of occasions, it should perfectly fit my size, and the pockets should be conveniently deep. The shirt is to go with the suit, it should not be of a certain fabric. Finally, the tie should look good and match well with the rest.

Even this vague specification is not given at full length to the shopkeeper - even if she might know some of it from my previous patronage of her shop. I just indicate my size as well as stile and purpose of the suit needed. Then, I am allowed to try several pieces (some kind of very rapid prototyping - but this is no problem for the shopkeeper, since a huge number of pieces are available, just waiting for being tried) and we finally zero in on the two most likely alternatives which are then fitted by needles in length and width (still rapid prototyping, but with more effort than previously). Before my final selection is brought over to the taylor, we choose a shirt and the tie, well considering her advice and my taste.

The better I can specify my needs, the faster this process runs. But in any case, before I had entered the shop, I would have been unable to fully specify the outfit I will finally have bought. The specification always becomes also clearer in my head when I see the alternatives at hand. I doubt whether other people buy their outfits very differently! Why do they - or why do they pretend to - buy (or make) software all that differently?

2.3. Location of rigour in specifications

The reason, why people behave so differently when developping software than when buying less complex commodities can be found in the teachings of the first quarter of the software engineering century: "Go top -> down; divide first and conquer later!"

The principle of "divide (first) and conquer (later)" is well warranted, whenever separation of concern is the issue - and it is the issue when developping (software-) systems from scratch. As soon as we start to build a system by integrating prefabricated components, we cannot draw our component boundaries at will. Any mechanical or electrical engineer would immediately lose the job after approaching the first major design task in a "divide first and conquer later" attempt. The approach in these classical engineering disciplines is "read the parts-catalogues of potential suppliers first and design your system later". Thus, the design task, and hence the specification of interfaces and components, is not the disaggregation of a singular, monolytic (and often erratic) needs statement into codeable pieces. The traditional design task consists of satisfying well understood needs in the most elegant (efficient, ...) way by standard components. Hence, it is not a unidirectional walk from top (spec) to down (code, component) but the bouncing back and forth from needs to solutions. A bouncing between high level concepts and concrete (not necessary low level) realizations in very much the way experienced software engineers are working covertly anyway [CURT 89]. They may not do so openly, because the methodology, notably their specification methodology prohibits such deviations from the heighth of abstraction to the deep end of the project.

In the sequel we will show an approach to specify systems and components in situations where the detailed implementation of an algorithm (the aspect left open by classical specifications) is immaterial, but where tradeoffs of functionality - and hence arbitration of borderlines between components - will become a key issue. The approach is based on the assumption that there exists not only an amply stuffed library of readymade components, but also that these components have been designed in such a way that they have crisp and integratable (normalized [MITT 89a]) semantics.

The following chapter presents a formal model for expressing components just by the semantics they exhibit on their interface and which allows for the degrees of freedom needed for transferring the shopping paradigm described in 2.2. into software development.

3. Specifications to support reuse and integration

3.1. Requirements

It is generally acknowledged that software reuse pays off the better, the larger (more complex, more involved, ...) the components to be reused are. On the other hand, it is equally acknowledged, that chances for reuse become dimmer, the larger (or more complex, ...) the components in the library are. Therefore, given the choice, the developer will use at any time those components which are most comprehensive and which fit the current needs most accurately.

This implies however, that reuse will be confined neither to a particular level in the system (de-)composition hierarchy, nor to a particular phase in the project (c.f. [MITT 90]). Thus we require from the theoretical basis of our approach **neutrality of scale** and **implementation independence**.

Further, we are not interested in how the software is built. We are only interested in how it behaves. Therefore, any specifications which would in one way or another address the internal structure of a component - and be it only by means of far fetched analogies, which are in no way meant to constrain the implementation - are not adequate. We are interested only in the **functional behaviour**[1] of the components in their purest form.

Finally, "Design with Reuse" implies that preexisting components are fitted together. Hence, a specification supporting Design with Reuse has to **support a bottom up, compositional approach**. This essentially reverses current system development paradigms and, therefore, most of the specification approaches currently highly esteemed do have problems with this requirement. Their development paradigm is generally top-down decomposition and not bottom-up composition of systems.

An assessment of several approaches to specify software according to the criteria mentioned above led us to relational software specifications as described in the next paragraph. It not only fullfills the criteria mentioned above; its semantics are also sufficiently close to those of the relational underpinning of data bases, so that it can also be used as the basis for specifying software bases as described in [MITT 87b].

3.2. A relational model of software

3.2.1. Outline of relational specifications

The relational approach to specify software has been introduced by Mili [MILI 83] and is extended in [MILI 87]. [MILI 86b] shows a nice example of its usage, [MILI 86a] gives a comparison to other approaches. A comprehensive description can be found in [MILI 89]. The approach is summarized here only to the extent needed for the rest of this paper.

Assume the Pascal-like type definitions

```
TYPE    $t_1$ = onetype;
        $t_2$ = anothertype;
        ...
        $t_n$ = sometype;
```

1) It is well understood, that behaviour encompasses far more than functional behaviour. However, we want to separate concerns. Therefore, for the time being,we abstract from any performance-, human-interface-related-, or any other behavioural aspects.

and

```
        $t_r$ = record
                $r_1$ : $t_1$;
                $r_2$ : $t_2$;
                ...
                $r_n$ : $t_n$
              end
```

and the procedure declarations

```
  procedure p1 ($i_1$: $t_1$;  $i_2$: $t_2$;  ...;  $i_n$: $t_n$;  o:$t_r$)
and
  function f1 ($i_1$: $t_1$;  $i_2$: $t_2$;  ...;  $i_n$: $t_n$): $t_r$
```

then procedure p1 and function f1 map (irrespective of their otherwise potentially very different semantics) from the value-space (domain) of the types

$$t_1 \times t_2 \times ... \times t_n$$

to the value-space (domain) of

$$t_r,$$

which happens by accident to be also

$$t_1 \times t_2 \times ... \times t_n.$$

To distinguish between types and the set of values associated with this type, we will refer to the latter in the sequel by S. Such that val-set(t_i) = S_i = $\{s_i \mid s_i$ is a legitimate value of type $t_i\}$.

Given these prerequisits, we can claim that p1 and f1 can be specified by relations mapping from $S_1 \times S_2 \times ... \times S_n$ to S_r. The actual values yielded by p1 and f1 (their actual behaviour) can be seen from their program function [p1] and [f1] respectively (c.f. [MILL 75]).

In order to maximize the chance of finding a reusable component which satisfies the needs of the task to be solved, we want to specify the required behaviour as loosely as possible. Therefore, we want to allow for non-determinism and thus we do not use the program functions of p1 and f1, [p1] and [f1] respectively, but the following relations:

$$R_{p1} \subseteq S_1 \times S_2 \times ... \times S_n \times S_r$$

or, for brevity: $R_{p1} \subseteq S_r \times S_r$

and

$$R_{f1} \subseteq S_1 \times S_2 \times ... \times S_n \times S_r$$

or, for brevity: $R_{f1} \subseteq S_r \times S_r$

The differences between p1 and f1 will be accounted for in the particular definition of the subset operator used above. We can give the detailed definition of this relation by explicitely adding the predicates of p1 and f1 to our specification as in

$$R_{p1} = \{(s,s') \mid s \in S_r \wedge s' \in S_r \wedge pred_{p1}\}$$

and

$$R_{f1} = \{(s,s') \mid s \in S_r \wedge s' \in S_r \wedge pred_{f1}\}$$

where $pred_{p1}$ and $pred_{f1}$ might be considered as arbitrary (possibly different) well formed formulae expressed in first order predicate calculus.

3.2.2. Basic notations and definitions

In order to introduce some notational conventions, let R be

$$R \subseteq S \times S;$$

such that

$$R = \{(s, s') \mid s \in S \ \wedge \ s' \in S \ \wedge \mathrm{pred}_1\}$$

Let U, the universal relation, be

$$U = S \times S$$

Let I, the identity relation, be

$$I = \{(s, s) \mid s \in S\}$$

Let $\emptyset$, the empty relation be

$$\emptyset = \{\}.$$

We denote the domain and range of relation R, respectively by

$$\mathrm{dom}(R) = \{s \mid s \in S \wedge (\exists s' \in S: (s, s') \in R)\}$$
$$\mathrm{rng}(R) = \{s' \mid s' \in S \wedge (\exists s \in S : (s, s') \in R)\}$$

Further, we refer with

$$s.R = \{s' \mid (s, s') \in R\}$$

to the image set of an element s by relation R.

We assume that union, intersection and cross-product are defined on these relations in the usual way. For its importance for the rest of the paper, we may just single out the cross product of two relations R_1 and R_2:

$R_1 \times R_2 = \{(s, s') \mid \exists t \in S: (s, t) \in R_1 \ \wedge \ (t, s') \in R_2\}.$

The above definitions will suffice to address the issue of how to accommodate generality and how to build systems according to the paradigm sketched above.

Before going to the next section, however, the reader may check, that by modelling software in these relational terms, we do indeed consider only a module's functionality in its most abstract form. There are no implementation hints whatsoever. Assuming that pred_{f1} and pred_{p1} are identical, we would not see, whether, in this particular case, any one of the two, f1 and p1, would have been implemented as a function or procedure. Neither would we see, whether they compute their results algorithmically (which would be the default assumption - we do usually consider the relation just given in its intensional form) or by a table lookup operation (thus having the relation implemented in its extensional form).

3.3. Accommodating generality

3.3.1. Basic relationships

As pointed out in [BenC 90], these relations can be placed in hierarchical order according to the degree of "definedness". The properties of the resulting lattice structure has been carefully studied in [BOUD 90].

Intuitively, we could say that

relation R_1 is more defined than relation R_2,
if it carries more input-output information than R_2.

Formally, a relation R1 is said to be **more-defined** than R2 if and only if (iff)

$(\mathrm{dom}(R_1) \supseteq (\mathrm{dom}(R_2)) \wedge (\forall s \in \mathrm{dom}(R_2): s.R_1 \subseteq s.R_2).$

Expressed verbally, this says that:

A relation R_1 is more defined than another relation R_2 iff

a) its domain is encompassing the domain of the other relation R_2 (first conjunct)

and at the same time

b) any of its individual input-output mappings carries the same or less entropy (there are the same or less output values associated) than R_2 (second conjunct).

We will denote in the sequel the fact, that R_1 is **more-defined** than R_2 by the operator "**>d**". Evidently, in the context of system development by integrating reusable components, any component which is more-defined than the specification of the system element needed will be a candidate component for integration.

For the sake of convenience, we introduce the inverse of "more-defined", i.e. "**less-defined**" and denote it by the operator "**<d**":

$$R_2 <d \ R_1 \quad \text{iff} \quad R_1 >d \ R_2$$

Note, that if $R_1 >d R_2$ and $R_1 <d R_2$ than $R_1 = R_2$. If neither $R_1 >d R_2$ nor $R_1 <d R_2$ then R_1 and R_2 are not substitutable in a reuse effort.

To benefit most from the flexibility called for in the shopping paradigm, we will further refine and then generalize on the definition of "more-defined".

Refining on the definition of more defined, it comes immediately to mind that one can separate the two conjuncts, thus reaching:

R_1 is **domain-extended** with respect to R_2 (R_1 **>e** R_2)
iff $(\mathrm{dom}(R_1) \supseteq \mathrm{dom}(R_2))$

R_1 is **more-functional (less-entropic)** than R_2 (R_1 **>f** R_2)
iff $(\forall s \in \mathrm{dom}(R_2): s.R_1 \subseteq s.R_2)$

The definition of "domain-extended" seems straight forward. The definition of "more-functional" can be motivated as follows: $R_1 >f R_2$ implies, that R_1 and R_2 are drawn from the same universal relation $U = S \times S$. Further, there exists some relation R_3 drawn from the same universal relation such that

$R_3 = F = \{(s,s') \mid \forall s_1, s_2, s_1', s_2' \in S: (s_1 = s_2) \Rightarrow (s_1' = s_2')\} \wedge R_3 >d\ R_1$

i.e. the input argument of R_3 has (in database terms) the key-property, or R_3 is a function F. One might draw a dimension of decreasing entropy from U to F. On this dimension, R_1 would have to be placed closer to F - i.e. have rather properties one would expect from a function - than R_2.

$$U <.\ R_2 <.\ R_1 <.\ F$$

While the latter may be achieved on the level of the individual mappings, it might also be achieved on the level of the defining sets of R_1 and R_2. If R_2 is defined on S x S and R_1 is defined on S x T with $T \subseteq S$ and $\mathrm{dom}(R_1) = \mathrm{dom}(R_2)$ and both R_1 and R_2 are needing somewhere all the elements of their defined range, R_1 has to be "more-functional" than R_2. Hence, we get:

R_1 is **range-restricted** with respect to R_2 (R_1 **>r** R_2)
iff $(R_1 >d\ R_2) \wedge \mathrm{rng}(R_1) \subseteq \mathrm{rng}(R_2)$

Note: We do need the first conjunct here, to assure that R_1 and R_2 are still comparable. Further, we avoid having a simpler conjunct (e.g. $\mathrm{dom}(R_1) = \mathrm{dom}(R_2)$) to allow for further relaxing on the comparability of domains and thus broadening the scope of "more-defined". Note further, that ">r" implies that the range of the first relation is narrower than the one of the second relation. Here, the symbol ">" seems to be counterintuitive. It is left this way, however, to remain consistent with the overall pattern, i.e. $R_1 >x\ R_2$ implies that R_1 carries more input/output-information than R_2.

With the three definitions given above, we have just refined on the original definition of "more-defined". Now we want to extend it to open up further avenues for reusability.

3.3.2. Relationships describing proxi-matches of modules

The first extension covers the case, where we have two relations such that one can be obtained by means of a projection (in the database sense) from the former. I.e., given $R_1 \subseteq S_1 \times S_2 \times ... \times S_n$ and $R_2 \subseteq S_i \times S_j \times S_k$, $(1 \leq i, j, k \leq n)$ and $R_3 \subseteq S_i \times S_j \times S_k$ $(1 \leq i, j, k \leq n)$ we can say

R_3 is **projectively less-defined** than R_1 (R_3 **<dp** R_1)
iff $\exists R_2$: $(R_2 = \Pi_{i,j,k}(R_1) \wedge (R_3 <d\ R_2))$.

Conversely, we say
R_1 is **ip more-defined** than R_3 (R_1 **>dp** R_3)
iff $\exists R_2$: $(R_2 = \Pi_{i,j,k}(R_1) \wedge (R_3 <d\ R_2))$
("ip" stands for "by inverse projection").

"Projectively less-defined" ("ip-more-defined") requires that the intermediate relation R_2 is derived by a projection-operation. On the level of actual software this implies that some modifications need to be performed on p1 with $[p1] = R_1$ in order to satisfy specification R_3.

We will achieve a similar effect (and in fact, from the point of view of reuse, a more useful effect) by assigning certain constant values to those attributes, which are of no interest in a given application. (Note, if necessary, we can overcome some the theoretical awkwardness implied by this preselection by combining the results of different selection via union to obtain the desired result. Practically, software normalization should avoid this problem though) [MITT 89a,b].

Thus, we have
R_3 is **select-projectable** from R_1 (R_3 **~sp** R_1)
iff
$\exists R_2 \wedge (s_1, s_2, ..., s_n) \in \Pi_{(1 < g < n\ \wedge\ g \# i,j,k)} S_r$:
$R_2 = \Pi_{i,j,k}(\sigma_{(s1, s2, ..., sn)}(R_1)) \wedge (R_3 \sim d\ R_2))$

For ease of notation, we introduced the operator "**~x**" to specify, that a relation is x-comparable, i.e. either more- or less-defined under the level of refinement "x".

The above definition of "select-projectable" says, that a relation R_3 is projectable, (and more- or less-defined in its projected version), iff it is more- (or less-) defined than a projection on a selection of R_1, where this selection has been defined by binding all those attributes which do not partake in the projection to some constant value(s).

Note that select-projectability is a more stringent condition than projectively more (-less-) defined. Eg. $R_1 = \{ (1, 2, 3), (2, 3, 4), (3, 3, 6) \}$ is projectively less defined then $R_3 = \{(2,3), (3, \{5,6\}) \}$. However, the three possible select projections on the first attribute/argument - $RS_1 = \{(2, 3)\}$, $RS_2 = \{(3, 4)\}$, and $RS_3 = \{(3, 6)\}$ - are all unreleated to R_3.

While the notion of select-projectability looks formally clumsy, it is practically very simple. Given, the values are available, one needs only write a kind of wrapping where the values not needed are frozen to a constant.

Far easier, though very powerful (and possibly ladden with more substantial effort in the reuse case) is the notion of **type restrictions.** Note firstly, that type restrictions may, but need not, imply restrictions on the associated value set, and note secondly that type restrictions on domains and type restrictions on ranges will have quite different effects with respect to the notion of one relation being more-defined than the other. Further, the type-restrictions described below do not necessarily fit into the "lattice" spanned by the relationship "more-defined". Whether they are integratable in such a lattice depends on the particular relationships between the types involved.

Assuming now, we have the relations $R_1 \subseteq A \times B$ and $R_2 \subseteq C \times D$:

We say that R_2 is **domain type-restricted** with respect to R_1 (R_2 **<dtr** R_1)

iff C is a subtype (according to some type-hierarchy) of A

and we say that R_1 is **range type-restricted** with respect to R_2 (R_1 **>rtr** R_2)

iff D is a subtype (according to some type-hierarchy) of B.

If there are type restrictions on both, domain and range of R_2, R_1 and R_2 would not fit into our scheme of more-definedness. However, R_1 might still be a useful candidate in a reuse effort.

Hence, we define **type compatibility** (R_1 **~tc** R_2)

iff (type (dom(R_1)) ~~ type (dom(R_2)))

$\wedge$ (type (rng(R_1)) ~~ type (rng(R_2)))

Here, "~~" stands for "compatible" where the detailed semantics of compatible have to be defined in a particular type- (or class-) hierarchy (specific ISA-relationship).

Note that the above has been defined for programs (procedures, functions). It applies likewise to processes holding states and to objects. With state bearing entities, we could in a first shot, blindfolded attempt abstract from the state and treat it as entropy in the input-output/behaviour. Obviously, this will not lead us very far. Therefore, one has to take care in a more refined modelling step of the state by treating it as an additional input and output argument (c.f. [MITT 89a]).

With objects, an additional problem pops up, since the messages sent to an object consist of a method-selector and the associated arguments. Again, at a higher level, one can blindfoldedly treat the message selector as the input argument (and perhaps even project its arguments away). In a second step, one has to recognize though, that objects are higher order entities, encompassing a set of relations to describe their respective methods, each of which is selected by a relational attribute of the object itself.

4. Formalizing the shopping paradigm

The story presented in section 2.2. (shopping paradigm) could be repeated for almost any ordinary good we are buying. With technical constructions and with complex systems, though, we cannot just let us drive through the amazing wealth of whatever software repository. We need some formal mechanism to specify what we want to be integrated into the system to be built as well as what we might obtain ready-made and what needs to be added as totally new software. In order to achieve this, we remember from the previous section that

$U = A \times B$	universal relation
$R \subseteq A \times B$	nondeterministic specification
$F \subseteq A \times B$	with $a \in A$ having the "key"-property, i.e. $\forall\ a \in A$: $\lvert a.F \rvert = 1$ deterministic specification
$\Omega = A \times \{\gamma\}$	miracle specification

with

$$U <d\ R <d\ F <d\ \Omega$$

I.e. by going sucessively from domain equations to relational composition and finally to functional composition, we will obtain a form of solidification of a design similar to the shopping paradigm.

The ordering given above suggests the following design steps:

1. Syntactic-Level- or Type-Level- (De-) Composition
2. Semantic Level- or Relational Composition
3. Pragmatic Level Refinement or De-Entropication

We will describe them in more detail in the following sections.

4.1. Syntactic-level- or type-level- (de-) composition

Given some design task, we can apply the normalization strategy [MITT 89 a,b] to achieve a breakdown structure which will have high chances of finding a match in a repository of normalized components. If the decoposition has been obtained from normalization, we will obtain a rejoinable structure anyway. If not, we have to assure that the components to be found can be joined to a workable system.

Assume the system Sys to be developped should satisfy the specification $F_{AC} \subseteq A \times C$.

If we remove any semantics from F_{AC}, we could specify it as U_{AC}, with $U_{AC} = A \times C$. Assume further, that a decomposition of F_{AC} into F_{AB} and F_{BC} seems advisable for whatever reason, we can check the formula

$$U_{AB} \times U_{BC} = U_{AC}$$

with

$$\text{rng}(U_{AB}) \sim\text{tc dom}(U_{BC}).$$

This seems to be a rather trivial step. However, we see, that we need only type compatibilty instead of strict equality. Hence, we could check the software repository for availability of components satisfying U_{AB} and U_{BC} and we could perform already some error checking on our refined specification such as:

$\mathrm{rng}(U_{AB})$ #tc $\mathrm{dom}(U_{BC})$ ...	error
$\mathrm{rng}(U_{AB}) \supset \mathrm{dom}(U_{BC})$ ...	potential for undefined input for second component; necessitates further study of F_{AB} or further search
$\mathrm{rng}(U_{AB}) \subset \mathrm{dom}(U_{BC})$ ...	any proof obligations for U_{BC} will be reduced to $U_{BC} // b \in \mathrm{rng}(U_{AB})$

Note: "//" denotes a condition operator. It indicates that the relation left to this condition operator has to satisfy the predicate given right to the condition operator. "#tc" stands for "not type compatible".

All we are checking at this level is whether the components to be selected would at least be type compatible. One might criticise this approach on the grounds that there are cases where modifications on the level of the types of a software component are faster done than modifications on the internal functionality. However, we claim that such modifications can be safely done only if the types belong to some very closely related family (or they are irrelevant to the extent that the respective module might be parameterized by some type-genericity).

4.2. Semantic level- or relational composition

At this level we will treat the semantics of the components involved. Again, since normalization is a semantic concept, we should firstly make sure that the components under investigation are normalized. On these premises, we could take the nondeterministic specification R_{XY} as a search key into our software repository [BenC 90]. The respective relational equation is

$$R_{AB} \times R_{BC} = R_{AC}$$

with $R_{AB} \subseteq U_{AB}$ and $R_{BC} \subseteq U_{BC}$ such that $R_{AB} = U_{AB} // P_1$ and $R_{BC} = U_{BC} // P_2$ and also R_{AB} and R_{BC} have to be ~d with respect to F_{AB} and F_{BC} respectively. (If possible, they should be more-defined than some implemented function from the repository. It would be helpful, though, if there exists at least some relation which is comparable, i.e. <d or >d, to the function needed).

From the predicates of the composants P_1 and P_2 we can deduce that

$$R_{AC} = U_{AC} // P_1 \wedge P_2 \wedge b \in \mathrm{rng}(U_{AB})$$

With this, R_{AC} is a specification which comes at least very close to the actual specification of the needed system Sys_{AC}.

4.3. Pragmatic level refinement or de-entropication

Assume that Sys_{AC} is more-defined than R_{AC} by the equation

$$Sys_{AC} = U_{AC} // N$$

We can obtain Sys by placing some additional integrity constraints on the currently composed prototypical system. Since the prototypical system does satisfy already certain predicates P_1 and P_2 as well as domain predicates due to step 1, we do not need to place N as additional integrity constraint on the prototype just obtained but only the constraint I, which is obtained from

$$N = P_1 \wedge P_2 \wedge I.$$

This yields

$$Sys_{AC} = R_{AC} // I = (R_{AB} \times R_{BC}) // I$$

How to incorporate the additional integrity constraints I into the system as well as to which extent I should be distributed into $(R_{AB} \times R_{BC})$ remains a design decision. Further, it will depend on whether the prototyping tool at hand will support such placement of additional integrity constraints. (cf. the software base concept of [MITT 87b]).

If the aim of the reuse effort is not prototyping but target system development, the approach outlined above still has its merit, since it provides the designer with a full range of choices from the software repository as well as with the chance to investigate a decomposition of the predicate N into conjuncts such that

$$N = n_1 \wedge n_2 \wedge \ldots \wedge n_k$$

with n_j being a predicate totally applicable to R_j.

On this basis, one can redesign the program satisfying R_j such that

$$R'_j = R_j // n_j = U_j // P_j \wedge n_j$$

and then recompose the system

$$Sys_{AC} = (\Pi_{j=1..k} R'_j)$$

One may note though, that in certain situations it might be impossible or undesirable to fully decompose N into subpredicates n_i such that $\Pi_i n_i = N$. If $N = (\Pi_i n_i) \wedge n_r$, this remaining predicate n_r has still to be placed as constraint on the overall system, i.e. it has to be supported by the software which glues [POWE 90] the individual components together.

5. Comparison to other specification approaches

Comparing the relational approach to software specification with other approaches, one might wonder, to which extent the features described in section 4 are akin to this approach, or whether the same lines of arguments can also be followed when using more established approaches.

Taking "Z" [SPIV 89] as a representative of currently favoured specification languages, "loose specifications" come immediately to mind. They can be used in a similar way as the layered approach with relational specification. However, they differ in several respects. The similarity is due to the fact, that with loose specifications, certain predicates may be left incompletely defined. Thus, the specification will yield also a relation and not a function. The specification will be completed, when the (global) variable, left unbound at the outset, will finally be bound to some constant value.

The main difference to the relational approach is, that the underlying philosophy of "Z" is top-down specification of systems. This principle is not fully kept, since even with "Z"-specifications, one might join "Z"-schemas by the sequential composition operator ";" . In fact, one might define schema expressions using various composition operators. However, the basic principle, that with each schema not only the behaviour of a component at the interface, but also its internal structure (state variables) are to be defined, limits the usage of "Z" for building (specifying) systems via integration of reusable components.

A further aspect allowing looseness in "Z"- and other specification languages - is genericity. Schemas, notably definitions of data types, can be parameterized. On instantiation of such a schema, one has to supply an actual parameter (type) which satisfies the obligations defined within the generic schema. This principle, which is supported by many approaches - in section 3 we defined the notion of type-compatibility - has been highly elaborated in Goguen's type expressions (Theories) [GOGU 84] in OBJ and FOOPS [GOGU 90]. It is also directly incorporated in his integration-/specification language LIL [GOGU 86].

However, it should be said that the above is not to postulate a contradiction between relational and other algebraic specifications. Amongst other things, their common principle is that they can be split into a description of the signature and a body which defines the permissible "mappings" within the space defined by the signature. Thus, if one looks only at the signature, one works at the level defined above as the syntactic one. If the bodies of the specifications are also taken into account, one reaches the semantic level. It is only a question of how to access a software repository in the most efficient way whether one will use the predicate structure (and hence: predicate based search) or any other approach. We conjecture, that due to the wealth of knowledge in automatic reasoning, search based on the subsumption of predicates has its advantages.

Remains the pragmatic level. We consider this strictly a necessity stemming from the bottom up approach we want to support. Hence, it would be foreign to top down methods, to support such concepts. More surprising might be that only few authors favouring the object oriented paradigm have so far seen the necessity for this level [MILI 90]. Is this due to the fact, that such global notions run too cross to object oriented sermons of total locality?

6. Conclusion

It has been shown, that integrating systems from reusable components requires a diffent methodological approach, - and hence also different specification technology - than building systems from scratch. It has been argued, that a layered approach towards system specification will be needed to satisfy these aims.

The approach advocated has been demonstrated in the context of relational (software) specification. It permits to describe components in a purely functional way, focussing exclusively on the services needed/provided. Further, it allows for starting out with rather loose specifications which will be made more solid during the process of system development.

Traditionally, it would have been considered awkward, if system development starts out from such a weak specification that the builder will essentially define the system. However, under cost considerations, we should give requestors some handle, to clearly state what is mandatory and what is optional - i.e. to allow for "intentionally left open". The things "intentionally left open" at the initial project stages might be tightly specified as soon as more details are known of the rest of the system. We claim that this not only is a more natural approach - which comes close to what is nowadays already an acknowledged methodology: prototyping. It is also a mandatory approach if software reuse will have to become a success.

References

[BASI 90] Basili V.: "Viewing Maintenance as Reuse-Oriented Software Development"; IEEE-Software, Vol. 7/1, Jan. 1990, pp. 19 - 25.

[BENC 90] Ben Cherifa A., Boudriga N., Mili A., Mittermeir R.T., Rossak W.: "A Formal Specification Structure for Software Reuse", submitted for publication.

[BIGG 87] Biggerstaff T.J., Richter Ch.: "Reusability Framework, Assessment, and Directions", IEEE Software, Vol. 4/2, March 87, pp.41 - 49.

[BIGG 89] Biggerstaff T.J., Perlis A.J. (eds).: "Software Reusability", Vol. I and Vol. II, Addison Wesley and acm press, 1989.

[BOUD 90] Boudriga, N., Elloumi F., Mili A.: "The Lattice of Specifications"; SIAM Conference on Discrete Mathematics, Atlanta, Georgia, June 1990.

[CALD 91] Caldiera, G. , Basili V.: "Identifying and Qualifiying Reusable Software Components". IEEE Computer, Vol. 24/2, Feb. 1991, pp. 61-69.

[CURT 89] Curtis B: "What You Have to Understand: This Isn't The Way. We Develop Software At Our Company", invited speach delivered at 11th ICSE, Pittsburgh, May 1989

[FREE 87] Freeman P. (ed): "Software Reusability", IEEE tutorial, IEEE-CS press, 1987.

[GOGU 84] Goguen J.A.: "Parameterized Programming"; IEEE Trans. on Software Engineering, Vol. SE-10/5, Sept. 1984, pp. 528 - 543.

[GOGU 86] Goguen J.A.: "Reusing and Interconnecting Software Components"; IEEE Computer, Vol. 19/2, Feb. 1986, pp. 16 - 28.

[GOGU 90] Goguen J.A., and Wolfram D.: "On Types and FOOPS"; Proc. IFIP WG 2.6, DS-4 Conf, Windermere '90, North Holland, 1990.

[KAPP 89] Kappel G., Vitek J., Nierstrasz O., Gibbs S., Junod B., Stadelmann M.: "An Object-Based Visual Scripting Environment"; in: Tschritzis D. (ed.): "Object Oriented Development", C.U.I., Univ. de Geneve, 1989, pp. 123 - 142.

[MATS 83] Matsumoto Y.: A Software Factory: "An Overall Approach to Software Production", Proc. of the ITT Workshop on Reusability in Programming, Newport, RI, Sept. 83 (reprinted in [FREE 87]).

[MEYE 87] Meyer B.: "Reusability: The Case of Object-oriented Design"; IEEE Software, Vol. 4/2, March 87, pp. 50.64.

[MILI 83] Mili A.:, "A Relational Approach to the Design of Deterministic Programs"; Acta Informatica, Vol. 20/4, 1983, pp. 315 - 328.

[MILI 86a] Mili A., Desharnais J., Gagne J.R.: "Formal Models of Stepwise Refinement of Programs" acm Computing Surveys, Vol 18/3, Sept. 1986, pp. 2431 - 2476.

[MILI 86b] Mili A., Wang X.-Y., Yu Q.: "Specifiation Methodology: An Integrated Relational Approach"; Software - Practice and Experience, Vol. 16/11, Nov. 86, pp. 1003 -1030.

[MILI 87] Mili A., Desharnais J., Mili F.: "Relational Heuristics for the Design of Deterministic Programs", Acta Informatica, Vol. 24/3, 1987, pp. 239 - 276.

[MILI 89] Mili A., Boudriga N., Mili F.: "Towards Structured Specifying", Ellis Horwood Publ., 1989.

[MILI 90] Mili H., Sibert J., Intrator Y.: "An Object Oriented Model based on Relations"; Journal of Systems and Software, Vol 12/2, May 1990, pp. 139 - 156.

[MILL 75] Mills H.D.: "The New Math of Computer Programming", cacm, Vol 18/1, Jan 1975, pp. 43 - 48.

[MITT 87a] Mittermeir R. T., Rossak W.: "The Role of Generalization Hierarchies in Requirements Modelling", Proc. Algorithmy '87, High Tatras, CSFR, April 1987.

[MITT 87b] Mittermeir R. T., Oppitz M.: "Software Bases for the Flexible Composition of Application Systems"; IEEE Trans. on Software Engineering, Vol. SE-13/4, April 1987, pp. 440 - 460.

[MITT 89a] Mittermeir R. T.: "Normalization of Software to enhaver its Potential for Reuse", TR UBW, 6/89, 1989.

[MITT 89b] Mittermeir R. T.: "Design - Aspects Supporting Software Reuse" in: Dusink L., Hall P.A.V. (eds): "Software Re-use, Utrecht 1989", Springer, 1991, pp. 115 - 119.

[MITT 90] Mittermeir R. T., Rossak W.: "Reusability"; in: Ng P., Yeh R.T.: "Modern Software Engineering", van Nostrand, 1990, pp. 205 - 235.

[NIER 90] Nierstrasz O., Dami L., de Mey V., Stadelmann M., Tsichritzis D.,Vitek J.: "Visual Scripting Towards Interactive Construction of Object-Oriented Applications"; in: Tsichritzis D. (ed.): "Object Management", C.U.I., Univ. de Geneve, 1990, pp. 315 - 331.

[NIER 91] Nierstrasz Ol, Tsichritzis D., de Mey V., Stadelmann M.: "Objects + Scripts = Applications"; in Tsichritzis D. (ed): "Object Composition", C.U.I., Univ. de Geneve, 1991, pp. 11 - 29.

[POWE 90] Power L. R.:"Post-Facto Integration Technology: New Disciplinye for Old Practice", Proc. 1st ICSI, Morristown, New York, Apr. 1990, pp. 4 - 13

[PRIE 87] Prieto-Diaz R., Freeman P.: "Classifying Software for Reusability", IEEE Software, Vol. 4/1, Jan. 87, pp. 6 - 16.

[ROSS 89] Rossak W., Mittermeir R. T.: "A dbms Based Archive for Reusable Software Components", Proc. 2ieme J.I. Le Genie Logiciel & ses Applications, Toulouse, Dec. 89, pp. 501 - 516.

[SPIV 89] Spivey J.M.: "The Z Notation - A Reference Manual"; Prentice Hall, 1989.

[TRAC 88] Tracz W. (ed.): "Software Reuse: Emerging Technology", IEEE Tutorial, IEEE-CS press, 1988.

[WEIN 79] Weinberg G. M.: "You say your design's inexact? Try a wiggle", Datamation, Vol. 25/9, Aug. 1979, pp. 146 - 149.

[YOUR 79] Yourdon E., Constantine L. L.: "Structured Design", Prentice-Hall, 1979.