Architectural specifications in CASL*

Michel Bidoit[1] Donald Sannella[2] Andrzej Tarlecki[3]

[1] Laboratoire Spécification et Vérification, CNRS & ENS de Cachan, France
[2] Laboratory for Foundations of Computer Science, University of Edinburgh, UK
[3] Institute of Informatics, Warsaw University and Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland.

Abstract. One of the most novel features of CASL, the Common Algebraic Specification Language, is the provision of so-called *architectural specifications* for describing the modular structure of software systems. A brief discussion of refinement of CASL specifications provides the setting for a presentation of the rationale behind architectural specifications. This is followed by some details of the features provided in CASL for architectural specifications, hints concerning their semantics, and simple results justifying their usefulness in the development process.

1 Introduction

A common feature of present-day algebraic specification languages (see e.g. [SW83], [EM85], [GH93], [CoFI96], [SW98]) is the provision of *specification-building operations* [BG77] for building large specifications in a structured fashion from smaller and simpler ones. Less usual in specification languages are features for describing the modular structure of software systems under development. This paper is about the facilities for this that are provided in CASL, the new *Common Algebraic Specification Language* [CoFI98b] that has been developed under the auspices of the Common Framework Initiative [Mos97], [CoFI98a] in an attempt to create a focal point for future joint work on algebraic specifications and a platform for exploitation of past and present work on methodology, support tools, etc.

Following earlier practical experiences [FJ90], [FAC92] and foundational work [Bid88], [ST89], [SST92], [BH93], we argue that mechanisms for structuring specifications are not the same as and cannot suffice for describing the modular structure of software under development. CASL therefore provides a separate kind of specifications, so-called *architectural specifications*, for this purpose. An architectural specification consists of a list of *unit declarations*, indicating the component modules required with specifications for each of them, together with a *unit term* that describes the way in which these modules are to be combined. Such architectural specifications are aimed at the "implementation" modular structure of the system rather than at the "interaction" relationships between modules in the sense of [AG97] (the latter to be considered when specifications of "reactive" modules are introduced in a CASL extension, cf. [FL97]).

* This is a revised and expanded version of [BST99].

The aim of this paper is to present motivation, intuition and technicalities related to this concept. We provide some basic information about CASL in Sect. 2, discuss the development of programs from specifications by stepwise refinement in Sect. 3 and then introduce architectural specifications in Sect. 4. We stress there how generic components arise naturally from the desire to allow separate but related modules to be developed independently. The semantics and correctness aspects of architectural specifications with the simplest ways of combining modules are discussed in Sects. 5, 6 and 7. Further operators for combining modules are presented in Sect. 8. The development process in the presence of architectural specifications is briefly discussed in Sect. 9.

Architectural specifications are presented in the context of CASL. However, the overall ideas if not all the technicalities are applicable in any specification and development framework, as we explain in Sect. 10. We also venture there briefly into more advanced features of architectural specification and development, bringing in ideas of *behavioural refinement*.

2 CASL preliminaries

CASL is a formalism to describe CASL *structures*: many-sorted algebras with subsorts, partial operations and predicates. Structures are classified by *signatures*, which give *sort* names (with their subsorting relation), partial/total *operation* names, and *predicate* names, together with *profiles* of operations and predicates. For each signature Σ, the class of all Σ-structures is denoted $\mathbf{Mod}[\Sigma]$.

The basic level of CASL includes *declarations* to introduce components of signatures and *axioms* to give properties of structures that are to be considered as *models* of a specification. The logic used to write the axioms is essentially first-order logic (so, with quantification and the usual logical connectives) built over *atomic formulae* which include strong and existential equalities, definedness formulae and predicate applications, with generation constraints added as special, non-first-order sentences. A basic CASL specification SP amounts to a definition of a signature Σ and a set of axioms Φ. It denotes the class $[\![SP]\!] \subseteq \mathbf{Mod}[\Sigma]$ of its *models*, which are those Σ-structures that *satisfy* all the axioms in Φ: $[\![SP]\!] = \{A \in \mathbf{Mod}[\Sigma] \mid A \models \Phi\}$.

Apart from basic specifications as above, CASL provides ways of building complex specifications out of simpler ones by means of various *structuring constructs*. These include translation, hiding, union, and both free and loose forms of extension. *Generic specifications* and their *instantiations* with pushout-style semantics [EM85] are also provided. Structured specifications built using these constructs can be given a compositional semantics where each specification SP determines a signature $Sig[SP]$ and a class $[\![SP]\!] \subseteq \mathbf{Mod}[Sig[SP]]$ of models. We say that SP is *consistent* if $[\![SP]\!]$ is non-empty.

2.1 Example

Here is a sequence of definitions of CASL specifications. We intersperse them with comments to clarify the meaning of particular CASL constructs and notations.

The example is small but it is not contrived in the sense that the way in which the specifications build upon one another seems quite natural.

spec MONOID =
  sort *Thing*
  ops $null : Thing$;
    $o : Thing \times Thing \rightarrow Thing$, **assoc**, **unit** *null*
end

This is the usual specification of a monoid with a sort of elements, a constant, and a binary operation that is associative and has the constant as a neutral element.

spec NUM =
  sort *Num*
  ops $0 : Num$;
    $succ : Num \rightarrow Num$
end

A signature for natural numbers — a starting point for further specifications.

spec ADDNUM =
  { NUM **then op** $plus : Num \times Num \rightarrow Num$
    vars $x, y : Num$
    axiom $plus(x, succ(y)) = succ(plus(x, y))$ }
  and
  { MONOID **with** $Thing \mapsto Num$, $null \mapsto 0$, $o \mapsto plus$ }

This enriches NUM by a binary operation and then further requires that *Num* with *0* and *plus* form a monoid. The union of specifications is employed here to re-use a specification (MONOID) introduced earlier. Since everything is extremely simple we could as well incorporate the requirements from MONOID directly into the axioms for *plus*, but in more complex cases this could be a lot of work and moreover, some of the conceptual structure of the specification would be lost.

spec ORDNUM = NUM **then pred** $\_\_ < \_\_ : Num \times Num$
  axiom $\forall x : Num \bullet 0 < succ(x)$

Another extension of NUM by a loosely specified binary predicate.

spec CODENUM =
  ADDNUM **and** ORDNUM
  then op $code : Num \rightarrow Num$
    axiom $\forall x : Num \bullet 0 < code(x)$

In CODENUM we put both previous extensions of NUM together and then add a unary operation on *Num* with another simple axiom.

spec ELEM = **sort** *Elem* **end**

$$
\begin{array}{l}
\textbf{spec } \textsc{PartContainer } [\textsc{Elem}] = \\
\quad \textbf{generated type} \;\; \mathit{Cont} ::= \mathit{empty} \mid \mathit{add}(\mathit{Elem}; \mathit{Cont})? \\
\quad \textbf{pred} \quad \mathit{addable} : \mathit{Elem} \times \mathit{Cont} \\
\quad \textbf{vars} \quad x, y : \mathit{Elem};\ C : \mathit{Cont} \\
\quad \textbf{axiom} \quad \mathit{def}\ \mathit{add}(x, C) \Leftrightarrow \mathit{addable}(x, C) \\
\quad \textbf{pred} \quad \_\_ \in \_\_ : \mathit{Elem} \times \mathit{Cont} \\
\quad \textbf{axioms} \quad \neg\, (x \in \mathit{empty}); \\
\qquad\qquad (x \in \mathit{add}(y, C) \Leftrightarrow x = y \vee x \in C)\ \mathit{if}\ \mathit{addable}(y, C) \\
\textbf{end}
\end{array}
$$

This is a generic (in ELEM) specification of "partial containers", which introduces a datatype *Cont* generated by a constant *empty* and a partial constructor *add* that adds an element to a container. An element x may be added to a container C if and only if $\mathit{addable}(x, C)$ is satisfied. But *addable* is left unspecified at this stage. The usual membership predicate is provided as well.

$$
\begin{array}{l}
\textbf{spec } \textsc{PartNumCont} = \\
\quad \textsc{PartContainer}[\textsc{CodeNum}\ \textbf{fit}\ \mathit{Elem} \mapsto \mathit{Num}]
\end{array}
$$

We instantiate the above generic specification to CODENUM, with an appropriate fitting of the parameter. The result contains all the operations and predicates of CODENUM together with those added by PARTCONTAINER with the profiles of the latter adjusted accordingly.

$$
\begin{array}{l}
\textbf{spec } \textsc{UniqueNumCont} = \\
\quad \textsc{PartNumCont} \\
\quad \textbf{then}\ \ \textbf{vars} \quad x : \mathit{Num};\ C : \mathit{Cont} \\
\qquad\qquad \textbf{axiom} \quad \mathit{addable}(x, C) \Leftrightarrow \neg\, (x \in C) \wedge \neg\, (\mathit{code}(x) \in C)
\end{array}
$$

Finally, we constrain the addability condition, requiring that a number is addable to a container if and only if neither it nor its code are already included there.

3 Program development and refinement

The intended use of CASL, as of any such specification formalism, is to specify programs. Each CASL specification should determine a class of programs that correctly realize the specified requirements. To fit this into the formal view of CASL specifications, programs must be written in a programming language having a semantics which assigns[1] to each program its *denotation* as a CASL structure. Then each program P determines a signature $\mathit{Sig}[P]$ and a structure $[\![P]\!] \in \mathbf{Mod}[\mathit{Sig}[P]]$. The denotation $[\![SP]\!]$ of a specification SP is a description of its admissible realizations: a program P is a *(correct) realization* of SP if $\mathit{Sig}[P] = \mathit{Sig}[SP]$ and $[\![P]\!] \in [\![SP]\!]$.

[1] This may be rather indirect, and in general involves a non-trivial abstraction step. It has not yet been attempted for any real programming language.

In an idealized view of program development, we start with an initial loose requirements specification SP_0 and refine it step by step until some easily-realizable specification SP_{last} is obtained:

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_{last}$$

Stepwise refinement only makes sense if the above chain of refinements guarantees that any correct realization of SP_{last} is also a correct realization of SP_0: for any P, if $[\![P]\!] \in [\![SP_{last}]\!]$ then $[\![P]\!] \in [\![SP_0]\!]$. This is ensured by the definition of refinement: for any SP and SP' with the same signature, we define

$$SP \rightsquigarrow SP' \iff [\![SP']\!] \subseteq [\![SP]\!].$$

The construction of a program to realize SP_{last} is outside the scope of CASL. CASL provides means for building specifications only; in this sense it is not a "wide-spectrum" language [BW82]. Furthermore, there is no construct in CASL to explicitly express refinement between specifications. All this is a part of the meta-level, though firmly based on the formal semantics of CASL specifications.

A more satisfactory model of refinement allows for modular decomposition of a given development task into several tasks by refining a specification to a sequence of specifications, each to be further refined independently. (Of course, a development may branch more than once, giving a tree structure.)

$$SP \rightsquigarrow BR \begin{cases} SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_{1,last} \\ \vdots \\ SP_n \rightsquigarrow \cdots \rightsquigarrow SP_{n,last} \end{cases}$$

Once we have realizations $P_1, \ldots, P_n$ of the specifications $SP_{1,last}, \ldots, SP_{n,last}$, we should be able to put them together with no extra effort to obtain a realization of SP. So for each such branching point we need an operation to combine arbitrary realizations of $SP_1, \ldots, SP_n$ into a realization of SP. This may be thought of as a linking procedure $LINK_{BR}$ attached to the branching point BR, where for any $P_1, \ldots, P_n$ realizing $SP_1, \ldots, SP_n$, $LINK_{BR}(P_1, \ldots, P_n)$ realizes SP: if $[\![P_1]\!] \in [\![SP_1]\!], \ldots, [\![P_n]\!] \in [\![SP_n]\!]$ then $[\![LINK_{BR}(P_1, \ldots, P_n)]\!] \in [\![SP]\!]$.

Crucially, this means that whenever we want to replace a realization P_i of a component specification SP_i with a new realization P_i' (still of SP_i), all we need to do is to "re-link" it with realizations of the other component specifications, with no need to modify them in any way. $LINK_{BR}(P_1, \ldots, P_i', \ldots, P_n)$ is guaranteed to be a correct realization of SP, just as $LINK_{BR}(P_1, \ldots, P_i, \ldots, P_n)$ was. In other words, the only interaction between the components happens via $LINK_{BR}$, so the components may be developed entirely independently from each other.

The nature of $LINK_{BR}$ depends on the nature of the programs considered. They could be for instance just "program texts" in some programming language like Pascal (in which case $LINK_{BR}$ may be a simple textual operation, say, regrouping the declarations and definitions provided by the component programs) or actual pieces of compiled code (in which case $LINK_{BR}$ would really be linking in the usual sense of the word). Our preferred view is that the programming language in use has reasonably powerful and flexible modularization facilities, such as those in Standard ML [Pau96] or Ada [Ada94]. Then $P_1, \ldots, P_n$ are program modules (structures in Standard ML, packages in Ada) and $LINK_{BR}$ is a module expression or a *generic module* with formal parameters for which the actual modules $P_1, \ldots, P_n$ may be substituted. Note that if we later replace a module P_i by P'_i as above, "recompilation" of $LINK_{BR}(P_1, \ldots, P'_i, \ldots, P_n)$ might be required but in no case will it be necessary to modify the other modules.

One might expect that BR above is just a *specification-building operation* OP (or a specification construct expressible in CASL), and branching could be viewed as "ordinary" refinement $SP \rightsquigarrow OP(SP_1, \ldots, SP_n)$. Further refinement of $OP(SP_1, \ldots, SP_n)$ might then consist of separate refinements for $SP_1, \ldots, SP_n$ as above. Then we need at least that OP is "monotonic" with respect to inclusion of model classes.[2] Then the following "refinement rule" is sound:

$$\frac{SP_1 \rightsquigarrow SP'_n \qquad \cdots \qquad SP_n \rightsquigarrow SP'_n}{OP(SP_1, \ldots, SP_n) \rightsquigarrow OP(SP'_1, \ldots, SP'_n)}$$

This view is indeed possible provided that the specification-building operation OP is *constructive* in the following sense: for any realizations $P_1, \ldots, P_n$ of $SP_1, \ldots, SP_n$, we must be able to construct a realization $LINK_{OP}(P_1, \ldots, P_n)$ of $OP(SP_1, \ldots, SP_n)$. In that case, $OP(SP_1, \ldots, SP_n)$ will be consistent whenever $SP_1, \ldots, SP_n$ are. However, simple examples show that some standard specification-building operations (like the union of specifications) do not have this property. It follows that refining SP to $OP(SP_1, \ldots, SP_n)$, where OP is an arbitrary specification-building operation, does not ensure that we can provide a realization of SP even when given realizations of $SP_1, \ldots, SP_n$. (See [HN94] for a different approach to this problem.)

Another problem with the refinement step $SP \rightsquigarrow OP(SP_1, \ldots, SP_n)$ is that it does not explicitly indicate that subsequent refinement is to proceed by independently refining each of $SP_1, \ldots, SP_n$, so preserving the structure imposed by the operation OP. The structure of the specification $OP(SP_1, \ldots, SP_n)$ in no way prescribes the structure of the final program. And this is necessarily so: while preserving this structure in the subsequent development is convenient when it is natural to do so, refinements that break this structure must also be allowed. Otherwise, at very early stages of the development process we would have to fix the final structure of the resulting program: any decision about structuring a specification would amount to a decision about the structure of the final program. This is hardly practical, as the aims of structuring specifications in the early development phases (and at the requirements engineering phase) are quite

[2] The specification-building operations we use here, hence all derived specification constructs, are monotonic, as are most of the constructs of CASL and other specification languages. The few exceptions — like imposing the requirement of freeness — can be viewed as operations which add "constraints" to specifications rather than as fully-fledged specification-building operations.

distinct from those of structuring final programs. Simple examples are mentioned below, cf. [FJ90].

On the other hand, at certain stages of program development we need to fix the structure of the system under development: the design of the architecture of the system is often among the most important design decisions in the development process. In CASL, this is the role of *architectural specifications*, see Sect. 4.

3.1 Example

Consider the task of realizing UNIQUENUMCONT from Sect. 2.1. Its structure does not provide useful guidance to the structure of its realization. For instance, there would be obvious trouble with the use of union in ADDNUM: an attempt to implement ADDNUM "structurally", by providing independent realizations for NUM (with an appropriate extension by *plus*) and for MONOID (with appropriate renaming), would succeed only by pure chance!

Furthermore, the last extension of PARTNUMCONT by an axiom for *addable* cannot be a directive to first realize PARTNUMCONT and then somehow miraculously ensure that the predicate *addable* does indeed satisfy the axiom. After all, realizing PARTNUMCONT means, among other things, choosing a realization for *addable*. One might change this specification, so that a realization of PARTNUMCONT would be required for any choice of *addable* — but this would be quite a different specification with quite a different structure. Moreover, it would not enable the implementor to take advantage of the fact that the axiom for *addable* ensures that an element need never be added to a container more than once.

We might re-structure the above specification instead by introducing some new "constructive" compositions or exposing some existing ones. For instance:

spec UNIQUECONTAINER [CODENUM] =
  PARTCONTAINER[CODENUM **fit** $Elem \mapsto Num$]
  then **vars** $x : Num;\ C : Cont$
    axiom $addable(x, C) \Leftrightarrow \neg\,(x \in C) \wedge \neg\,(code(x) \in C)$

spec UNIQUENUMCONT' = UNIQUECONTAINER[CODENUM]

Then we have that UNIQUENUMCONT $\rightsquigarrow$ UNIQUENUMCONT' (in fact, the two specifications are equivalent) and the instantiation in the latter specification is "constructive", which indicates a possible split of further development to a part where a realization of CODENUM is developed and another part where UNIQUECONTAINER is implemented. See Sect. 4.1 below for details.

4 Architectural specifications

The conclusion from Sect. 3 is that we need to distinguish carefully between two kinds of structuring mechanisms needed in the specification and development process.

On one hand we need the standard mechanisms to structure specifications to facilitate their construction, reading, understanding and re-use. These are provided by the specification-building operations of CASL, disregarding whether these operations are "constructive" or not. In general, their use should not be viewed as fixing the shape of the development tree or as determining the modular structure of the final program. On the other hand, at a certain stage of program development we need to design the structure of the final program, and consider these decisions binding in the subsequent development process. Such a design is given by refining a specification to a "constructive" combination of specified components. The essence here is not so much the use of a constructive specification-building operation, as rather some specific construction (linking procedure) that builds a realization of the original specification once given realizations of the component specifications.

The latter structuring facility, although quite standard in modular programming languages, is rarely explicitly provided in specification formalisms. In many approaches, the structure of the specification is regarded as determining the structure of the final program, examples like those in Sect. 3.1 notwithstanding, see e.g. [GB80], [MA91]. Or else *ad hoc* informal mechanisms are used to indicate that a certain part of the structure of a specification (given by a constructive specification-building operation) is to remain fixed throughout the rest of the development. We consider this unsatisfactory and likely to be confusing. Therefore CASL provides an explicit notation whereby one specifies the components required together with a way to combine them to build the resulting program. Such *architectural specifications* (an alternative terminology is *organizational specifications* [GHW82]) can be used to refine ordinary specifications, whether structured or not, explicitly introducing branching into the development process and structure into the final program:

$$SP \leadsto \boxed{BR \begin{cases} SP_1 \\ \vdots \\ SP_n \end{cases}}$$

The corresponding architectural specification is written as follows:

$$\begin{array}{ll} \textbf{units} & U_1 : SP_1; \\ & \ldots \\ & U_n : SP_n \\ \textbf{result} & LINK_{BR}(U_1, \ldots, U_n) \end{array}$$

Notice that we provide names for program *units* to be implemented according to the component specifications given, and we give a "linking procedure" $LINK_{BR}$ to combine these units rather than an operation to combine their specifications. The component specifications $SP_1, \ldots, SP_n$ are ordinary CASL specifications. The "linking procedure" $LINK_{BR}(U_1, ..., U_n)$ is just a *unit term* that might involve the units named $U_1, \ldots, U_n$. It builds a new unit when given actual units $U_1, \ldots, U_n$ that correctly realize the specifications $SP_1, \ldots, SP_n$.

Typically $SP_1, \ldots, SP_n$ (and so, units that realize them) will contain shared parts, or some of them will rely on others. For instance, we might start by implementing some simple specification SP_1. Then, given an implementation U_1 of SP_1, build an implementation U_2 of some "larger" specification SP_2 using U_1, etc. The last stage is to build an implementation U_n of SP_n using U_{n-1}, and the final result is U_n. The corresponding architectural specification is:

units $U_1 : SP_1$;
$U_2 : SP_2$ **given** U_1;
...
$U_n : SP_n$ **given** U_{n-1}
result U_n

Here, the "linking procedure" U_n is trivial since all of the linking was done when we used U_{j-1} to build U_j for $1 < j \leq n$. Of course, this is just the simplest case. In particular, it does not cover multiple dependencies (where a unit might use several other units), sharing between various units in a more flexible way than just having each unit use the previous one, or reusability (whereby a unit may be used more than once). Still, it illustrates the idea of splitting a development task into subtasks, clearly indicating their interfaces and the flow of information between them. In the extreme, such a split may be done step by step, each time splitting the work into just two parts:

$SP \leadsto$ $\boxed{\begin{array}{ll} \textbf{units} & U_1 : SP_1; \\ & U_2 : SP\ \textbf{given}\ U_1 \\ \textbf{result} & U_2 \end{array}}$

The task of providing a realization U_1 for SP_1 is *independent* from the task of providing a realization U_2 for SP using U_1. It follows that no properties of U_1 may be exploited in the development of U_2 other than those explicitly ensured by the specification SP_1. This requires a realization of SP for *any* realization of SP_1, which is tantamount to requiring a *generic* realization F of SP which takes the particular realization of SP_1 as parameter. Then we obtain U_2 by simply feeding U_1 to F.

Genericity here arises from the independence of the developments of U_1 and U_2, rather than from the desire to build multiple realizations of SP using different realizations of SP_1. This is reflected in the fact that F is not named in the architectural specification above. If it is desired to indicate the potential for re-use explicitly, we may give F "first-class" status as a so-called *generic unit* with a specification $SP_1 \rightarrow SP$ which indicates that it will realize SP when given a realization of SP_1:

units $U_1 : SP_1$;
$F : SP_1 \rightarrow SP$;
$U_2 = F[U_1]$
result U_2

Here, $U_2 = F[U_1]$ is a so-called *unit definition*.

The earlier specification is equivalent to this version with the sole exception that F is anonymous there. This shows how to explain architectural specifications involving "**given**" by translation to architectural specifications involving explicit generic units. A key insight is the use of genericity to control the flow of information between developments of independent units, as well as for multiple instantiation. Despite this, it seems useful to retain both notations as they convey different pragmatic intuitions.

In this specification (and in all those arising from translation of specifications involving "**given**") the generic unit F is instantiated only once, but in general it may be applied to more than one argument, as demonstrated in Sect. 8.3.

In programming languages with sufficiently powerful modularisation facilities, generic units correspond to some form of generic modules (functors in Standard ML, generic packages in Ada, etc.). This is in contrast to units (or simply: programs) like U_1, realizing ordinary structured specifications, which correspond to "closed" modules (structures in Standard ML, non-generic packages in Ada, etc.). The components of such a closed unit are available for use in any program that imports it. The only way to use generic units is to first instantiate them, otherwise their components are not ready for use.

Generic unit specifications correspond to functor headings in Extended ML [ST89] and to a restricted form of Π-specifications in [SST92], cf. Spectral [KS91]. On the other hand, generic unit specifications and generic specifications coincide in ACT ONE [EM85], which the above discussion argues is inappropriate.

4.1 Example

Recall the specifications built in Sect. 2.1 and the further comments on them in Sect. 3.1. We ended up there with a specification

spec UniqueNumCont' = UniqueContainer[CodeNum]

which suggests a way of decomposing the task of implementing UniqueNumCont. This may be turned into a design decision by refining this specification to an architectural specification that captures the decomposition meant here:

arch spec UCNum =
  units N : CodeNum;
    UCN : UniqueNumCont' **given** N
  result UCN

Then UniqueNumCont $\rightsquigarrow$ UCNum (this becomes fully formal only when the semantics of architectural specifications is given more precisely below).

We might, however, be a bit more clever in our design and require a realization of containers with the specified "uniqueness" property for arbitrary elements equipped with the operations that allow one to express this property. For instance:[3]

spec TRANSELEM =
  sort *Elem*
  op $transform : Elem \rightarrow Elem$
end

spec ABSTRACTUNIQUECONT =
  PARTCONTAINER[TRANSELEM]
  then **vars** $x : Elem;\ C : Cont$
    axiom $addable(x, C) \Leftrightarrow \neg (x \in C) \wedge \neg (transform(x) \in C)$

arch spec ABSTRACTUCNUM =
  units N : CODENUM;
    AUC : TRANSELEM $\rightarrow$ ABSTRACTUNIQUECONT
  result $AUC[N$ **fit** $Elem \mapsto Num,\ transform \mapsto code]$

We still have UNIQUENUMCONT $\rightsquigarrow$ ABSTRACTUCNUM.

The required generic unit AUC here is more abstract and more general than the "anonymous" unit to build UCN as required in UCNUM. AUC has to work for arbitrary structures fitting the abstract TRANSELEM specification; it could be re-used in the future for arguments other than N.

The anonymous generic unit in UCNUM is required to work only for structures fitting the considerably richer specification CODENUM. This might make life easier for its implementor (the extra structure can be used in the implementation) but also makes the unit less general.

It is up to the system designer to choose whether to follow the "more general" or "more specific" line of design and so choose between ABSTRACTUCNUM and UCNUM (or some yet different architectural specification) as a refinement for UNIQUENUMCONT. The key point is that an architectural specification may be given to present an architecture for the system in a prescriptive way.

5 Semantics of unit specifications

To provide a formal framework covering the above ideas as well as more advanced aspects of architectural specifications, we will now take a closer look at the underlying semantics of generic units and their specifications.

Consider a unit specification of the form $SP' \rightarrow SP$, and let Σ' and Σ be the respective signatures of SP' and SP. In CASL, SP is implicitly viewed as an extension of SP'. Therefore, without loss of generality, we assume that in each specification of the form $SP' \rightarrow SP$, SP extends SP', that is: $\Sigma' \subseteq \Sigma$ and $[\![SP]\!]|_{\Sigma'} \subseteq [\![SP']\!]$.

[3] The reader is kindly asked to rely on her/his intuition and the obvious analogy with the instantiation of generic specifications to grasp the meaning of instantiation of generic units with non-trivial fitting of arguments. Details will be given in Sect. 8.3.

As indicated above, to realize the specification $SP' \to SP$, we should provide a "program fragment" ΔP for $SP \setminus SP'$ that extends any realization P' of SP' to a realization P of SP, which we will write as $\Delta P(P')$. The basic semantic property required is that for all programs P' such that $[\![P']\!] \in [\![SP']\!]$, $\Delta P(P')$ is a program that extends P' and realizes SP (semantically: $[\![\Delta P(P')]\!]|_{\Sigma'} = [\![P']\!]$ and $[\![\Delta P(P')]\!] \in [\![SP]\!]$). This amounts to requiring ΔP to determine a partial function[4] $[\![\Delta P]\!]: \mathbf{Mod}[\Sigma'] \to ?\ \mathbf{Mod}[\Sigma]$ that "preserves" its argument whenever it is defined, is defined on (at least) all structures in $[\![SP']\!]$,[5] and yields a result in $[\![SP]\!]$ when applied to a structure in $[\![SP']\!]$. Consequently:

$$\begin{aligned} [\![SP' \to SP]\!] = \{F : \mathbf{Mod}[\Sigma'] \to ?\ \mathbf{Mod}[\Sigma] \mid \\ &\text{for all } A' \in \mathrm{Dom}(F), F(A')|_{\Sigma'} = A', \\ &\text{for all } A' \in [\![SP']\!], F(A') \text{ is defined and } F(A') \in [\![SP]\!]\} \end{aligned}$$

This definition can easily be restated in a form closer to the definition of the semantics of specifications in Sect. 2. First, we can generalize the notion of CASL structures to generic structures as follows:

$$\begin{aligned} \mathbf{Mod}[\Sigma' \to \Sigma] = \{F : \mathbf{Mod}[\Sigma'] \to ?\ \mathbf{Mod}[\Sigma] \mid \\ &\text{for all } A' \in \mathrm{Dom}(F), F(A')|_{\Sigma'} = A'\} \end{aligned}$$

Then $[\![SP' \to SP]\!]$ can equivalently be defined by:

$$\begin{aligned} [\![SP' \to SP]\!] = \{F \in \mathbf{Mod}[\Sigma' \to \Sigma] \mid \\ &\text{for all } A' \in [\![SP']\!], F(A') \text{ is defined and } F(A') \in [\![SP]\!]\} \end{aligned}$$

Note that this set will be empty if there is some model of SP' that cannot be extended to a model of SP; then we say that $SP' \to SP$ is *inconsistent*.

This semantic view of program fragments as partial functions naturally leads to further generalisations. The most obvious one is to admit multi-argument functions, providing for the possibility that the realization of some specification might depend on realizations of more than one (sub-)specification. Specifications of multiply-dependent units will have the form $SP_1 \times \ldots \times SP_n \to SP$. As with singly-dependent units, we assume that SP extends each of $SP_1, \ldots, SP_n$ (or equivalently, their union). Let $\Sigma_1, \ldots, \Sigma_n$ and Σ be the respective signatures of $SP_1, \ldots, SP_n$ and SP. We then have:

$$\begin{aligned} [\![SP_1 \times \ldots \times SP_n \to SP]\!] = \\ \{F \in \mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n \to \Sigma] \mid \\ &\text{for all } \langle A_1, \ldots, A_n \rangle \in [\![SP_1 \times \ldots \times SP_n]\!], \\ &F(A_1, \ldots, A_n) \text{ is defined and } F(A_1, \ldots, A_n) \in [\![SP]\!]\} \end{aligned}$$

where $\mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n \to \Sigma]$ and $[\![SP_1 \times \ldots \times SP_n]\!]$ are defined as explained below.

[4] As in CASL, $X \to ?\ Y$ denotes the set of partial functions from X to Y.

[5] Intuitively, $\Delta P(P')$ is "statically" well-formed as soon as P' has the right signature, but needs to be defined only for arguments that realize SP'.

In general, not all tuples $\langle A_1, \ldots, A_n \rangle$ of structures $A_1 \in [\![SP_1]\!], \ldots, A_n \in [\![SP_n]\!]$ can be extended to structures in $[\![SP]\!]$: if a symbol in SP is inherited from one or more of $SP_1, \ldots, SP_n$, then its interpretation in the resulting structure must be the same as in each corresponding argument structure. So, if such a symbol occurs in several arguments then it is impossible to expand a tuple of arguments to a result unless all of the relevant arguments interpret this symbol in the same way.

A tuple $\langle A_1, \ldots, A_n \rangle$ of structures $A_1 \in \mathbf{Mod}[\Sigma_1], \ldots, A_n \in \mathbf{Mod}[\Sigma_n]$ is *compatible* if any symbol that occurs in both Σ_i and Σ_j is interpreted in the same way in A_i and A_j, for $1 \leq i, j \leq n$.[6] Compatible tuples $\langle A_1, \ldots, A_n \rangle$ are in bijective correspondence with structures $A \in \mathbf{Mod}[\Sigma_1 \cup \ldots \cup \Sigma_n]$ over the union of the signatures $\Sigma_1, \ldots, \Sigma_n$. Namely, each structure $A \in \mathbf{Mod}[\Sigma_1 \cup \ldots \cup \Sigma_n]$ corresponds to the compatible tuple $\langle A|_{\Sigma_1}, \ldots, A|_{\Sigma_n} \rangle$. On the other hand, given a compatible tuple $\langle A_1, \ldots, A_n \rangle$, there exists a unique structure $A \in \mathbf{Mod}[\Sigma_1 \cup \ldots \cup \Sigma_n]$ such that $A_1 = A|_{\Sigma_1}, \ldots, A_n = A|_{\Sigma_n}$ — we will call A the *amalgamation* of $\langle A_1, \ldots, A_n \rangle$, and write it as $A_1 \oplus \ldots \oplus A_n$. Then we take $\mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n]$ to be the class of all compatible tuples of structures from $\mathbf{Mod}[\Sigma_1], \ldots, \mathbf{Mod}[\Sigma_n]$, respectively, and use this to define the semantics of tuples of specifications:

$$\begin{aligned}
&\mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n] = \{\langle A|_{\Sigma_1}, \ldots, A|_{\Sigma_n} \rangle \mid A \in \mathbf{Mod}[\Sigma_1 \cup \ldots \cup \Sigma_n]\} \\
&[\![SP_1 \times \ldots \times SP_n]\!] = \\
&\qquad \{\langle A_1, \ldots, A_n \rangle \in \mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n] \mid A_1 \in [\![SP_1]\!], \ldots, A_n \in [\![SP_n]\!]\}
\end{aligned}$$

Given this, $\mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n \to \Sigma]$ is defined as follows:

$$\begin{aligned}
\mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n \to \Sigma] = \{F\colon & \mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n] \ \to?\ \mathbf{Mod}[\Sigma] \mid \\
& \text{for all } \langle A_1, \ldots, A_n \rangle \in \mathrm{Dom}(F), \\
& F(A_1, \ldots, A_n)|_{\Sigma_i} = A_i, \ \text{for } i = 1, \ldots, n\}
\end{aligned}$$

6 Sharing and well-formedness

In spite of their somewhat technical motivation, the definitions at the end of the previous section convey important methodological concepts. Namely, we now have a way to require that a number of units (fed to a unit dependent on them) *share* some of their parts. Even though they might be developed independently, certain parts of the argument units must be identical. In CASL, this requirement is imposed by the use of the same names in argument signatures for symbols which are to be shared between the argument units. An application of a generic unit to a tuple of arguments is *well-formed* only if the arguments do indeed share their commonly-named parts. In a programming language like Standard ML,

[6] This is correct for signatures without subsorts (i.e., with a discrete subsort ordering). In the presence of non-trivial subsorts, the precise conditions for compatibility must be more carefully stated, see [CoFI99].

this is a part of the "type discipline" and the required sharing is (type-)checked statically.

Consider the following simple example:

spec SP_0 = **sort** s **end**
spec SP_a = **sort** s **op** $a : s$ **end**
spec SP_b = **sort** s **op** $a, b : s$ **end**
spec SP_c = **sort** s **op** $a, c : s$ **end**
spec SP_d = **sort** s **op** $a, b, c, d : s$ **axiom** $d = b \lor d = c$ **end**

Then the generic unit specification $SP_b \times SP_c \to SP_d$ imposes a constraint on the arguments for the generic unit: they are required to share a common realization of the sort s and constant a. Consequently, given the following unit declarations:

units $U_b : SP_b$;
$U_c : SP_c$;
$F_d : SP_b \times SP_c \to SP_d$

the instantiation $F_d[U_b, U_c]$ cannot be allowed, since there is nothing that ensures that the units U_b and U_c do indeed share s and a. It is easy to provide units U_b and U_c that realize SP_b and SP_c respectively without fulfilling this sharing requirement. On the other hand, consider the following unit declarations:

units $U_a : SP_a$;
$F_b : SP_a \to SP_b$;
$F_c : SP_a \to SP_c$;
$F_d : SP_b \times SP_c \to SP_d$

The unit term $F_d[F_b[U_a], F_c[U_a]]$ is well-formed in the context of these declarations. The required sharing between the two arguments for F_d, namely between $F_b[U_a]$ and $F_c[U_a]$, is ensured. In both $F_b[U_a]$ and $F_c[U_a]$ the sort s and constant a come from U_a, and so must be the same. This follows simply from the fact that generic units expand their arguments, preserving them without any modification in the result.

The situation becomes a bit less clear if components of instantiations of generic units are involved. For instance, consider:

units $U_0 : SP_0$;
$F_a : SP_0 \to SP_a$

and declarations of F_b, F_c, F_d as above. Is $F_d[F_b[F_a[U_0]], F_c[F_a[U_0]]]$ well-formed? One might expect so: the sort s in the two arguments for F_d can be traced to the same unit U_0, and the constant a to the two occurrences of $F_a[U_0]$. Here, the sharing of s does not raise any objections (it just requires the "tracing procedure" to search through a slightly longer chain of instantiations). But the argument that the two occurrences of $F_a[U_0]$ share the constant a cannot be carried too far. In general, to decide if two instantiations of F_a, say $F_a[U_0]$ and $F_a[U_0']$, share the constant a, we would have to check if the two argument units U_0 and U_0' are identical. Clearly, this is too complicated for static analysis, even if in trivial cases it can be seen to hold immediately, as above. Moreover, in some programming languages with flexible modularisation facilities the new items introduced by instantiation of generic modules are distinct for each such instantiation. For instance, functors (generic modules) in Standard ML have such a "generative" semantics: each time a functor is instantiated to an argument, the new types it builds are generated anew and are kept distinct from those built by other instantiations, even if the arguments were the same each time. A similar phenomenon occurs with Ada generic packages.

Therefore, for safety, we assume that new symbols introduced by a generic unit are not shared between its instantiations, even when its arguments are the same in each case. (For programming languages with "applicative" rather than generative modules, this treatment is sound albeit marginally more awkward than necessary.) Auxiliary unit definitions may be used in CASL to avoid repetition of unit instantiation. For instance, we can rewrite the previous example:

$$
\begin{array}{ll}
\textbf{units} & U_0 : SP_0; \\
& F_a : SP_0 \rightarrow SP_a; \\
& U_a' = F_a[U_0]; \\
& F_b : SP_a \rightarrow SP_b; \\
& F_c : SP_a \rightarrow SP_c; \\
& F_d : SP_b \times SP_c \rightarrow SP_d
\end{array}
$$

In the context of the above unit declarations and definitions, $F_d[F_b[U_a'], F_c[U_a']]$ is well-formed and captures the intention behind $F_d[\,F_b[F_a[U_0]],\, F_c[F_a[U_0]]\,]$. An alternative way to present this example is to make the definition of U_a' local to the unit instantiation:

$$\textbf{local}\ U_a' = F_a[U_0]\ \textbf{within}\ F_d[\,F_b[U_a'],\ F_c[U_a']\,]$$

This is legal in the context of the previous unit declarations.

To sum up: in the context of a sequence of unit declarations and definitions, symbols in two units *share* if they can be traced to a common symbol in a non-generic unit. The "tracing procedure" can be broken down according to the constructs available for forming unit terms. For applications of generic units to arguments, symbols in the result are new if they do not occur in the argument signatures. Otherwise they can be traced to the same symbols in the arguments (and, transitively, to the symbols those can be traced to). The symbols of a declared unit can be traced only to themselves. The symbols of a defined unit may be traced according to the definitional term for the unit. So, for instance, using the specifications above, consider:

$$
\begin{array}{ll}
\textbf{units} & U_0 : SP_0; \\
& F_a : SP_0 \rightarrow SP_a; \\
& U_a' = F_a[U_0]; \\
& G : SP_0 \times SP_a \rightarrow SP_b
\end{array}
$$

The term $G[U_0, U_a']$ is well-formed since the sort s that the two arguments are required to share can in both cases be traced to the sort s in U_0.

7 Semantics of unit terms

As indicated above, an architectural specification comprises a sequence of unit declarations and unit definitions followed by a unit term which shows how the named units can be put together to build the result. Obviously, it is not possible to put together units in completely arbitrary ways; they must fit together properly, as in modular programming languages. Then given an *environment* which maps the declared unit names to particular (possibly generic) structures, the result term denotes a structure.

The static analysis of unit terms, with sharing analysis etc., is just the beginning of checking their correctness. The most crucial step is to check that when a unit (or tuple of units) is fed to a generic unit then the interfaces match, making sure that the requirements imposed on the parameter(s) of the generic unit by its specification are fulfilled by the argument (tuple). To take a simple example:

$$\begin{array}{ll}\textbf{units} & U : SP; \\ & F : SP' \;\to\; SP''\end{array}$$

Can we now feed the unit U to the generic unit F? Or in other words: is the unit term $F[U]$ *correct*? In order for it to be well-formed, the signatures of U and of the argument of F must coincide: $Sig[SP] = Sig[SP']$. And if F were multiply-dependent with symbols in common between different arguments, then sharing would also have to be checked. But also, F is required to work only for arguments that realize SP', including the requirements imposed by any axioms SP' may contain. So, for $F[U]$ to be correct, we must make sure that what we know about U is sufficient to establish what is required of the argument for F. Clearly, everything we know about U is recorded in SP — no other information is available. Even later on, when the unit U has been developed, the whole point of its declaration here — which decomposes the development task into developing U and F separately — is to limit the knowledge about U at this level to what is provided by SP. So, what we know about the unit U is that it denotes a structure in $[\![SP]\!]$. The argument of F is required to denote a structure in $[\![SP']\!]$. Consequently, the term $F[U]$ is correct provided $[\![SP]\!] \subseteq [\![SP']\!]$.

We have used different words to describe different aspects of "good" unit terms. *Well-formedness* is a static property, expected to be decidable so that it can be checked automatically. To check whether a unit term is well-formed we need information about the signatures of the units available (a reference to a non-available unit is not well-formed, of course) as well as sharing information about them (this is trivial for declared units but non-trivial for defined units). In such a context, well-formedness of a term is determined as sketched in Sect. 6. *Correctness* requires verification: it is not decidable in general. To check whether a unit term is correct we need full semantic information about the units that make it up, as explained below.

The last example was perhaps misleadingly simple: the argument U of F came equipped with an explicit specification (SP) that provided all the information that was available about U. In general, the argument may be more complex than this, and still we have to be able to gather all the information about it that is available. So, for instance, what do we know about $F[U]$ (in the context of the above unit declarations — of course assuming that $Sig[SP] = Sig[SP']$ and $[\![SP]\!] \subseteq [\![SP']\!]$)? Clearly, we know that the result realizes the specification SP''. Is this all? Not quite: we also know that U, and hence the reduct of $F[U]$ to $Sig[SP]$, realizes SP, which may carry more information than SP' does.

Given an environment ρ which maps unit names to particular (possibly generic) structures, a unit term T denotes a structure $[\![T]\!]_\rho$, defined inductively as follows:

- If T is a unit name U then $[\![T]\!]_\rho = \rho(U)$.
- If T is an instantiation $F[T_1, \ldots, T_n]$ where F is an n-ary generic unit and $T_1, \ldots, T_n$ are unit terms, then $[\![F[T_1, \ldots, T_n]]\!]_\rho = \rho(F)([\![T_1]\!]_\rho, \ldots, [\![T_n]\!]_\rho)$.

Some unit terms will not denote. A trivial reason for this might be the application of a generic unit to the wrong number of arguments, or to arguments with wrong signatures, or the use of an unbound unit name. Less trivially, there might be an attempt to apply a generic unit to a non-compatible tuple of structures. These cases cannot arise if the term is well-formed in the sense discussed above. Finally, a term will not denote if it involves application of a generic unit to a structure outside its domain; this cannot happen if the term is correct.

Correctness is defined in a *context* γ where unit names are associated with specifications rather than with particular structures realizing those specifications.[7] We say that an environment ρ *matches* a context γ if they bind the same unit names and for each unit name U in their domain, the structure $\rho(U)$ realizes the specification $\gamma(U)$: $\rho(U) \in [\![\gamma(U)]\!]$.[8] For any unit term T that is well-formed in the context γ, we write $[T]_\gamma$ for the class of all structures $[\![T]\!]_\rho$ that T denotes in environments ρ that match γ. Intuitively, $[T]_\gamma$ captures the properties of the unit built by T using unit declarations and definitions that determine γ.

Correctness of a well-formed unit term is defined by induction on its structure as follows:

- A unit name U is correct. (By well-formedness, U is declared in γ.) It follows that $[U]_\gamma = [\![\gamma(U)]\!]$.
- An instantiation $F[T_1, \ldots, T_n]$ is correct, where $\gamma(F)$ is $SP_1 \times \ldots \times SP_n \rightarrow SP$, if $T_1, \ldots, T_n$ are so and $[T_1]_\gamma \subseteq [\![SP_1]\!], \ldots, [T_n]_\gamma \subseteq [\![SP_n]\!]$. It follows that

$$[F[T_1, \ldots, T_n]]_\gamma = \{A \in [\![SP]\!] \mid A\big|_{Sig[SP_1]} \in [T_1]_\gamma, \ldots, A\big|_{Sig[SP_n]} \in [T_n]_\gamma\}.$$

[7] The context carries all semantic information about available units. It is convenient to think of the information about a unit as taking the form of a unit specification, even though the formal semantics of CASL uses slightly more complex semantic objects here. The specifications of declared units are given directly in the declarations. For defined units, the semantic information (together with the relevant sharing information) is determined according to the semantics of the unit term in the definition, as explained below.

[8] Moreover, the units in ρ share the components indicated by the sharing information in the context γ.

This omits the use of defined units in unit terms, treated in the obvious way: information about these units is extracted from their definitional terms and stored in the context as well. Further constructs for unit terms are discussed in the next section.

The above statements defining the correctness of unit terms also provide a more direct way to compute $[T]_\gamma$, without referring to the class of all environments that match γ. This can be proved by induction on the structure of unit terms, and can be used to directly calculate the ensured properties of T, and to validate its correctness.

Theorem 1. *Let γ be a context and let T be a unit term that is well-formed and correct in γ. Then for any environment ρ that matches γ, $[\![T]\!]_\rho$ is defined (and $[\![T]\!]_\rho \in [T]_\gamma$).*

This means that once we have finished the development process and so have provided realizations of each of the units declared, a correct result term will successfully combine these realizations to give a structure. Moreover, this structure satisfies the properties we can calculate directly from the architectural specification. Correctness of the result term of an architectural specification can be checked before realizations of its component units are provided. No *a posteriori* checking is necessary to ensure that independent successful developments of the components will fit together to give a correct result.

8 Other operators

Apart from the direct use of declared units and instantiation of generic units with actual arguments, a number of other constructs to build units are useful and are typically provided in some form in programming languages with advanced modularisation facilities. In some sense, none of these produces a new unit; they are used to "customize" what we have already defined, for instance to fit it to a required signature.

Each of these constructs, except for generic unit expressions (Sect. 8.4), relates directly to one of the specification-structuring constructs in CASL. For instance, amalgamation of units relates to union of specifications. To draw attention to this relationship, the syntax is deliberately the same. Nevertheless, it is crucial not to confuse the two levels, as was explained in Sect. 3.

8.1 Amalgamation

We need a way of putting together already developed units, to build a larger unit that contains all of their components. The semantic counterpart of this operation is amalgamation. Given unit terms $T_1, \ldots, T_n$, their amalgamation is denoted by T_1 **and** $\cdots$ **and** T_n. Consider the following example, where NUM is as in Sect. 2.1.

spec CHAR =
  sort *Char*
  ops $a, b, c : Char$
end
spec NUMANDCHAR = NUM **and** CHAR
arch spec SPLIT =
  units N : NUM;
    C : CHAR
  result N **and** C

In the above, SPLIT describes one natural way to realize the specification NUMANDCHAR, by simply realizing its two totally independent parts separately, and then putting the two units realizing these two parts together:

NUMANDCHAR $\rightsquigarrow$ SPLIT

Just as when feeding a number of required arguments to a generic unit, when amalgamating a number of units we must make sure that they share components having common names. Here is another trivial example:

spec NUM_23 = NUM **then preds** *divisible_2* : *Num*;
    divisible_3 : *Num*
spec NUM_2 = NUM **then pred** *divisible_2* : *Num*
spec NUM_3 = NUM **then pred** *divisible_3* : *Num*
arch spec SPLIT_23 =
  units N : NUM;
    F_2 : NUM $\rightarrow$ NUM_2;
    F_3 : NUM $\rightarrow$ NUM_3
  result $F_2[N]$ **and** $F_3[N]$

Given the above, NUM_23 $\rightsquigarrow$ SPLIT_23 . However, had we attempted:

arch spec SPLIT?_23 =
  units N_2 : NUM_2;
    N_3 : NUM_3
  result N_2 **and** N_3

then N_2 **and** N_3 would not be a well-formed unit term, since we have not ensured that the realization of NUM is shared between N_2 and N_3.

More formally: in a context γ, given well-formed unit terms $T_1, \ldots, T_n$, their amalgamation T_1 **and** $\cdots$ **and** T_n is a well-formed unit term over the signature that is the union of the signatures of $T_1, \ldots, T_n$, provided that each common symbol in the signatures of T_i and T_j is shared between T_i and T_j (i.e. can be traced in both T_i and T_j to the same symbol in a declared non-generic unit), for $1 \leq i < j \leq n$.[9] If this is the case, then for any environment ρ matching γ,

$$[\![T_1 \textbf{ and } \cdots \textbf{ and } T_n]\!]_\rho = [\![T_1]\!]_\rho \oplus \cdots \oplus [\![T_n]\!]_\rho$$

[9] As mentioned in footnote 6, compatibility conditions for amalgamation must be more carefully stated in the presence of subsorts, see [CoFI99].

It follows that

$$[T_1 \textbf{ and } \ldots \textbf{ and } T_n]_\gamma = \{A_1 \oplus \ldots \oplus A_n \mid A_1 \in [T_1]_\gamma, \ldots, A_n \in [T_n]_\gamma, \\ \langle A_1, \ldots, A_n \rangle \text{ is compatible}\}$$

A well-formed amalgamation T_1 **and** $\cdots$ **and** T_n is correct whenever each of $T_1, \ldots, T_n$ is correct.

The sharing requirement ensures compatibility of any structures $A_1 \in [T_1]_\gamma, \ldots, A_n \in [T_n]_\gamma$ that result from the developments of units described in the context γ. In other words: for any environment ρ that matches γ, $[\![T_1]\!]_\rho, \ldots, [\![T_n]\!]_\rho$ are compatible.

For instance, recall the above example SPLIT_23. In the unit term describing the result there, once N, F_2 and F_3 are bound to specific structures resp. generic structures in an environment ρ, then $[\![F_2[N]]\!]_\rho$ and $[\![F_3[N]]\!]_\rho$ are compatible. This holds even though there may be structures in $[F_2[N]]_\gamma$ and $[F_3[N]]_\gamma$ respectively that are not compatible with each other (where γ is the context determined by the unit declarations in SPLIT_23). But this is normal: even in $[N]_\gamma$ there are structures that are not compatible with each other, while clearly once a specific structure is bound to N in ρ, then $[\![N]\!]_\rho$ is a structure that is compatible with itself.

The amalgamation construct is in some sense redundant. Given specifications SP_1 and SP_2, the specification $SP_1 \times SP_2 \to \{SP_1 \textbf{ and } SP_2\}$ unambiguously specifies a generic unit which produces the amalgamation of any two (compatible) arguments. So, instead of adding syntax for amalgamation, we could simply specify the amalgamation units as needed. However, we feel that this would not be an appropriate simplification: it might mislead the reader into thinking that such a specification carries non-trivial implementation requirements.

8.2 Reduct and renaming

Another construct which seems necessary is that of *reduct*. It allows the user to design realizations that contain some auxiliary components not to be exported for use by clients. For example:

spec SP = **sort** s **end**
spec SP_{ab} = **sort** s **op** $a, b : s$ **end**
spec SP_{bc} = **sort** s **op** $b, c : s$ **end**
arch spec SP' =
  units $S : SP$;
    $F_{ab} : SP \to SP_{ab}$;
    $F_{bc} : SP \to SP_{bc}$
  result { $F_{ab}[S]$ **hide** a } **and** { $F_{bc}[S]$ **reveal** s, c }

In the result term of the architectural specification SP' we have used two forms of reduct, which we want to provide here just as in CASL structured specifications. The first, as used in "$F_{ab}[S]$ **hide** a", lists the symbols to be hidden. The well-formedness conditions simply require that the hidden symbols are actually there.

The semantics is simple as well: the reduct, as defined for CASL structures, is taken. Tracing symbol names is trivial: the symbols remaining in the reduct are traced as they are in the unit to which hiding is applied. The second form of hiding (i.e. **reveal**) is similar but dual.

Similarly as in CASL structured specifications, we can also rename components of units. The well-formedness conditions are a little more complicated: not only must the renamed symbols be in the signature of the unit, but if we rename two different symbols to the same name, they must share in the unit term to which we apply the renaming.[10] Then, each new symbol is traced to its origin in the obvious way. The renamed unit denotes a structure over the signature that is the target of the renaming, where the interpretation of each symbol is given by the interpretation of its original name in the original structure. As in CASL structured specifications, revealing and renaming may be combined.

Given well-formedness, hiding and renaming do not impose any additional correctness conditions.

8.3 Instantiation with fitting morphisms

Hiding and renaming may be used to adjust the names of unit components to whatever is required. For instance, consider the following specification:

spec STACKNUM =
  sorts *Num*, *Stack*
  ops *0* : *Num*;
    succ : *Num* $\rightarrow$ *Num*;
    empty : *Stack*;
    push : *Num* $\times$ *Stack* $\rightarrow$ *Stack*;
    pop : *Stack* $\rightarrow?$ *Stack*;
    top : *Stack* $\rightarrow?$ *Num*
  axioms ...
end

A natural refinement of this is to the following architectural specification:

spec ELEM = **sort** *Elem* **end**
spec STACKELEM =
  sorts *Elem*, *Stack*
  ops *empty* : *Stack*;
    push : *Elem* $\times$ *Stack* $\rightarrow$ *Stack*;
    pop : *Stack* $\rightarrow?$ *Stack*;
    top : *Stack* $\rightarrow?$ *Elem*
  axioms ...
end

[10] Again, the precise conditions are a little more complex in the presence of subsorts, see [CoFI99].

arch spec STACKOFNUM =
units N : NUM;
ST : ELEM $\rightarrow$ STACKELEM
result $\{$ $ST[N$ **reveal** $Num \mapsto Elem]$ **with** $Elem \mapsto Num$ $\}$ **and** N

Neutral names are used in the parameter for ST to indicate its potential re-usability. It is easy to come up with situations in which one would instantiate this unit in a number of different ways. But then each application to a particular unit will involve renaming to make the names match, with another renaming after application to recover the original names as above. Furthermore, since ST does not depend on anything other than the sort of elements (*Num*), the extra operations available in NUM are absent in the result of the application. So if we want to be able to push *0* onto the empty stack (say) in the resulting unit, we need to put back these additional operations, as above.

Such a pattern — extracting part of a unit (N above), renaming its components appropriately (**reveal** $Num \mapsto Elem$ above) to feed it as an argument to a generic unit (ST above), and then renaming the components of the result back (**with** $Elem \mapsto Num$ above) to finally combine them with the original unit (**and** N above) — occurs frequently enough that it deserves a special construct. By analogy with instantiation of generic specifications in CASL, we therefore provide application of a generic unit to its argument via a *fitting morphism*. The above example would be written as:

arch spec STACKOFNUM =
units N : NUM;
ST : ELEM $\rightarrow$ STACKELEM
result $ST[N$ **fit** $Elem \mapsto Num]$

The symbol map given here expands to a signature morphism in exactly the same way as in the instantiation of generic specifications. In particular, we allow compound identifiers, which are treated just as they are there. So we can have:

spec STACKELEM' =
sorts $Elem$, $Stack[Elem]$
ops $empty : Stack[Elem]$;
$push : Elem \times Stack[Elem] \rightarrow Stack[Elem]$;
$pop : Stack[Elem] \rightarrow ? Stack[Elem]$;
$top : Stack[Elem] \rightarrow ? Elem$
axioms ...
end
arch spec TWOSTACKS =
units C : CHAR;
N : NUM;
ST : ELEM $\rightarrow$ STACKELEM'
result $ST[C$ **fit** $Elem \mapsto Char]$ **and** $ST[N$ **fit** $Elem \mapsto Num]$

The result unit term of TWOSTACKS builds a unit which includes units for CHAR and NUM as well as two distinct stack sorts $Stack[Char]$ and $Stack[Num]$

together with two separate sets of operations (their names are overloaded, but their profiles are distinct and this is used to distinguish them).

Instantiation via a fitting morphism is just an abbreviation for the expanded form, with an explicit use of reduction, renaming and amalgamation.[11] In particular, such an instantiation will not be well-formed if component names introduced by the generic unit (that is, those that are not part of the parameter) also occur in the actual parameter. Since sharing between such two occurrences of a symbol cannot be ensured,[12] the amalgamation that is implicit in the instantiation cannot be well-formed. This requirement of separation between the body of a generic specification and its actual parameters has been imposed as an extra static condition in the case of structured specifications.

8.4 Generic unit expressions

So far we have not provided any means for building generic units. As usual, we can simply use λ-notation. We restrict to functions on non-generic units, i.e. higher-order generic units are not available. Here is an example:

arch spec DECOMPOSE =
  units $F : SP_0 \to SP_1$;
    $G : SP_1 \to SP$
  result $\lambda X : SP_0 \bullet G[F[X]]$

This builds a unit that realizes the specification $SP_0 \to SP$.

A λ-expression $\lambda X : SP \bullet T$ is correct in a context γ when T is correct in the expansion of γ by $X \mapsto SP$ (and similarly for well-formedness). Given an environment ρ that matches γ,

$$[\![\lambda X : SP \bullet T]\!]_\rho = \{A \mapsto [\![T]\!]_{\rho[X \mapsto A]} \mid A \in [\![SP]\!]\}$$

To show a simple example of the use of this construct, recall the specifications from Sects. 2.1, 3.1 and 4.1. The architectural specification UCNUM contained an anonymous generic unit with the specification CODENUM $\to$ UNIQUENUMCONT'. In Sect. 4.1 we indicated that it may be replaced by a more general unit AUC : TRANSELEM $\to$ ABSTRACTUNIQUECONT (from the architectural specification ABSTRACTUCNUM). More formally, this can be captured by forming the following architectural specification:

arch spec UCNBYAUC =
  unit AUC : TRANSELEM $\to$ ABSTRACTUNIQUECONT
  result λX : CODENUM $\bullet$ $AUC[X$ **fit** $Elem \mapsto Num,\ transform \mapsto code]$

Now, we have the following refinement (see Sect. 9 below for a discussion of refinement between specifications of generic units):

CODENUM $\to$ UNIQUENUMCONT' $\rightsquigarrow$ UCNBYAUC

[11] When the fitting morphism is not injective, the expansion is a bit more involved than indicated here, but the same principle applies.

[12] Unless the symbol in the generic unit originates from an import of a unit that shares with the actual parameter.

9 Refinements of architectural specifications

Section 4 indicated how a specification may be refined to an architectural specification. With the semantic notation introduced in subsequent sections, the related semantic correctness condition can be expressed as follows:

$$SP \rightsquigarrow \boxed{\begin{array}{ll} \mathbf{units} & U_1 : \ldots \\ & \ldots \\ & U_n : \ldots \\ \mathbf{result} & T \end{array}} \iff [T]_\gamma \subseteq [\![SP]\!]$$

where γ is the context built by the declarations of the units $U_1, \ldots, U_n$. This definition views the class $[T]_\gamma$ (of all result units that T might denote in the context of the unit declarations and definitions given) as the visible, "external" meaning of the architectural specification.

What next? That is, how can architectural specifications themselves be refined? Simple: by refining each of the specifications of declared units separately. But what about specifications of generic units? These are of the form $SP_1 \to SP_2$ (omitting the possibility of multi-argument generic units). Clearly, refinement will preserve genericity, so the question is when

$$SP_1{\to}SP_2 \quad \rightsquigarrow \quad SP'_1{\to}SP'_2$$

To begin with, we need the signatures to agree, that is: $Sig[SP_1] = Sig[SP'_1]$ and $Sig[SP_2] = Sig[SP'_2]$. Furthermore, as with specifications of closed structures, we need that every generic unit that realizes $SP'_1{\to}SP'_2$ must correctly realize $SP_1{\to}SP_2$. For consistent specifications $SP'_1{\to}SP'_2$, this amounts to requiring $[\![SP_1]\!] \subseteq [\![SP'_1]\!]$ and $[\![SP'_2 \textbf{ and } SP_1]\!] \subseteq [\![SP_2]\!]$. Notice that the latter condition is slightly weaker than the more obvious $[\![SP'_2]\!] \subseteq [\![SP_2]\!]$ — we can take advantage of the fact that we will only be applying the unit to arguments that realize SP_1. Hence, refinement of specifications of generic units is in fact a special case of refinement as introduced in Sect. 3:

$$SP_1{\to}SP_2 \quad \rightsquigarrow \quad SP'_1{\to}SP'_2 \qquad \iff \qquad [\![SP'_1{\to}SP'_2]\!] \subseteq [\![SP_1{\to}SP_2]\!]$$

since we have modeled generic units as partial functions that are required to be defined on the models of the argument specification, as explained in Sect. 5.

This allows for "linear" development of individual units declared in an architectural specification. To allow further decomposition, we can refine unit specifications to architectural specifications. For closed units this is covered above. Specifications of generic units may be refined to architectural specifications as well. The only difference is that then architectural specifications with generic result units, as introduced in Sect. 8.4, must be used. The semantics of such a generic result unit term defined in the context of some unit declarations yields a class of functions, which must be included in the class of functions denoted by the generic unit specification to be refined.

The overall effect is that we have a development tree, rather than just a sequence of refinement steps. This was indeed the target from the very beginning. Each leaf of such a tree may be developed independently from the others, using the full machinery of further decomposition via architectural design etc. The development subtree beginning at any given node may be replaced by another development tree without affecting the other parts as long as the new development subtree is correct with respect to the specification at its root.

In the discussion above, it is somehow implicit that architectural specifications as such do not need to be refined. Only the specifications of the declared units within an architectural specification are subject to further refinement, since the architectural specification itself is merely a prescription for further separate independent development of these units (and a description of how to combine the resulting individual pieces into the desired result). This is quite satisfactory from a methodological point of view, and indeed no further refinement concept seems necessary to achieve our goals.

However, one can also argue that when the specifications of the units of a given architectural specification are refined, a refinement of this architectural specification is obtained by textually replacing the unit specifications by their respective refinements. This is based on the following theorem:

Theorem 2. *Let* ASN *be the following architectural specification:*

arch spec ASN =
  units $\cdots$
    $U_1 : SP_1$;
    $\cdots$
    $U_2 : SP_2$ **given** T_2;
    $\cdots$
    $F : SP_a \rightarrow SP_r$;
    $\cdots$
  result T

Assume that in ASN *all the implicit generic specifications involved in unit specifications with imports are consistent.*
Let γ be the context built by the unit declarations in ASN *and assume that all the unit terms involved (in particular, the result unit term T) are well-formed and correct in γ.*
Let then ASN' *be the architectural specification obtained from* ASN *by textually replacing (some) unit specifications by their refinements. So, given $SP_1 \rightsquigarrow SP'_1$, $SP_2 \rightsquigarrow SP'_2$ and $SP_a \rightarrow SP_r \rightsquigarrow SP'_a \rightarrow SP'_r$, we have:*

arch spec ASN' =
  units $\cdots$
    $U_1 : SP'_1$;
    $\cdots$
    $U_2 : SP'_2$ **given** T_2;
    $\cdots$

$F : SP'_a \ \rightarrow\ SP'_r;$
$\dots$
result T

Let γ' *be the context built by the unit declarations in* ASN'*. Then all the unit terms involved (in particular, the result unit term* T*) are well-formed and correct in* γ'*. Moreover,* $[T]_{\gamma'} \subseteq [T]_{\gamma}$*, i.e.* ASN $\rightsquigarrow$ ASN'.

The above theorem shows that refinements of entire architectural specifications work as expected, provided all the implicit generic specifications involved in unit specifications with imports are consistent. Indeed this assumption cannot be dropped, as shown by the following counterexample (where we assume INT to be the usual specification of integers, equipped with the predicate *even* specified in the usual way):

arch spec AS =
  units U : { INT **then op** a : *Int* **axiom** $a = 1 \ \vee\ a = 0$ }
    V : { **op** b : *Int* **axiom** $b = a + 1 \ \wedge\ b$ *even* } **given** U
  result V

arch spec AS' =
  units U : { INT **then op** a : *Int* **axiom** $a = 1$ }
    V : { **op** b : *Int* **axiom** $b = a + 1 \ \wedge\ b$ *even* } **given** U
  result V

The architectural specification AS is inconsistent (since V cannot be built for a unit U in which $a = 0$), while AS' is consistent. Thus, even though the specification of the unit U in AS' is a correct refinement of the specification of U in AS, AS' is *not* a semantically correct refinement of AS. It is important to note that the problem here arises from the inconsistency of the implicit generic specification involved in the declaration of V in AS. If V were replaced by an *explicit* generic specification, the consistency condition would not be required since the only refinement of an inconsistent specification, generic or not, is another inconsistent specification.

10 Further comments

We have discussed the issue of designing the structure of a system to be developed from a specification. Our conclusion has been that apart from the usual mechanisms for structuring requirements specifications, we need a separate mechanism to describe the modular structure of the system to be developed. CASL provides this in the form of *architectural specifications*. We presented the basic ideas behind this concept, as well as the full design of architectural specifications in CASL. The semantics of architectural specifications has been sketched as well, but see [CoFI99] for all the details. The level of detail in the presentation was sufficient to state a few basic facts about the semantics, as well as to argue that properties of architectural specifications ensure that the basic goals of their design have been achieved. Namely, architectural specifications make it possible to describe the structure of the system to be developed by listing the units to be built, providing their specifications and indicating the way they are to be combined to form a more complex unit. Units here correspond to generic or non-generic modules, and possibilities to adequately specify the former are provided. Once such an architectural specification is given then its internal correctness can be checked and the ensured properties of the resulting module can be calculated (to check that the original requirements specification has been fulfilled by this design). Moreover, further developments of the units required may proceed independently from each without any need to check that the results are compatible, which brings in all the benefits of modular development.

The above ideas have been presented in the specific context of CASL. However, both the overall idea and the constructs for architectural specifications are largely independent from the details of the underlying CASL logical system. In fact, everything here can be presented in the context of an arbitrary *institution* [GB92] equipped with some extra structure to handle specific presentations of signature morphisms in reducts and renamings and to deal with the issues of sharing between structures when they are amalgamated. Details of a notion of an institution appropriate for the full semantics of institution-independent CASL (or rather, its structured specification and architectural specification mechanisms) are in [Mos98].

One issue which we have omitted above is that of *behavioural implementation* [Sch87], [ST89], [NOS95], [ST97], [BH98]. The idea is that when realizing a specification it is not really necessary to provide a model; it is sufficient to provide a structure that is *behaviourally equivalent* to a model. Intuitively, two structures are *behaviourally equivalent* if they cannot be distinguished by computations involving only the predicates and operations they provide. For CASL structures this can be formally captured by requiring that the two structures satisfy exactly the same definedness sentences and predicate applications — a more general form where equations between terms of some observable sorts are taken into account may be reduced to this by introducing some extra predicate symbols. Generic structures are behaviourally equivalent if they yield behaviourally equivalent structures for each argument.

When using a structure that was built to realize a specification up to behavioural equivalence, it is very convenient to pretend that it actually is a true model of the specification. This is sound provided all the available constructions on structures (hence all the generic units that can be developed) map behaviourally equivalent arguments to behaviourally equivalent results. More precisely: a generic unit is *stable* if for any behaviourally equivalent arguments provided for it via a fitting morphism, the overall results of instantiations of this unit on them are behaviourally equivalent as well. It is important in this formulation that the arguments considered may be built over a larger signature than just the argument signature of the unit — this models the fact that the unit may be used in richer contexts. If all units are stable, it is sufficient to check *local behavioural correctness* of unit terms only: this is defined like correctness in

Sect. 7, but allows the arguments for generic units to fit their formal requirement specifications only up to behavioural equivalence. Then the ensured properties $[T]_\gamma$ of any well-formed and locally behaviourally correct unit term T in a context γ can still be calculated exactly as in Sect. 7, as justified by the following theorem:

Theorem 3. *Let γ be a context and let T be a unit term that is well-formed and locally behaviourally correct in γ. Then for any environment ρ that matches γ up to behavioural equivalence, $[\![T]\!]_\rho$ is in $[T]_\gamma$ up to behavioural equivalence.*

Acknowledgements Our thanks to the whole of CoFI, and in particular to the Language Design Task Group, for many discussions and opportunities to present and improve our ideas on architectural specifications. Thanks to Till Mossakowski for comments on a draft. This work has been partially supported by KBN grant 8 T11C 018 11, the LoSSeD workpackage of CRIT-2 funded by ESPRIT and INCO (AT), a French-Polish project within the CNRS-PAS cooperation programme (MB, AT), and by EPSRC grant GR/K63795, an SOEID/RSE Support Research Fellowship and the FIREworks working group (DS).

References

[Ada94] *Ada Reference Manual: Language and Standard Libraries*, version 6.0. International standard ISO/IEC 8652:1995(E). `http://www.adahome.com/rm95/` (1994).

[AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[BW82] F. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Springer (1982).

[Bid88] M. Bidoit. The stratified loose approach: a generalization of initial and loose semantics. *Selected Papers from the 5th Workshop on Specification of Abstract Data Types*, Gullane. Springer LNCS 332, 1–22 (1988).

[BH93] M. Bidoit and R. Hennicker. A general framework for modular implementations of modular systems. *Proc. 5th Joint Conf. on Theory and Practice of Software Development*, Orsay. Springer LNCS 668, 199–214 (1993).

[BH98] M. Bidoit and R. Hennicker. Modular correctness proofs of behavioural implementations. *Acta Informatica* 35:951–1005 (1998).

[BST99] Architectural specifications in CASL. *Proc. 7th Intl. Conf. on Algebraic Methodology and Software Technology* AMAST'98, Manaus, Brasil. Springer LNCS 1548, 341–357 (1999).

[BG77] R. Burstall and J. Goguen. Putting theories together to make specifications. *Proc. 5th Intl. Joint Conf. on Artificial Intelligence*, Cambridge, 1045–1058 (1977).

[CoFI96] The Common Framework Initiative. Catalogue of existing frameworks. `http://www.brics.dk/Projects/CoFI/Catalogue.html` (1996).

[CoFI98a] The Common Framework Initiative. CoFI: The Common Framework Initiative for algebraic specification and development (WWW pages). `http://www.brics.dk/Projects/CoFI/` (1998).

[CoFI98b] CoFI Task Group on Language Design. CASL – The CoFI algebraic specification language – Summary (version 1.0). `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/` (1998).

[CoFI99] CoFI Task Group on Semantics. CASL – The CoFI algebraic specification language – Semantics (version 1.0). To appear (1999).

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Springer (1985).

[FAC92] *Formal Aspects of Computing* 4(1) (1992). Special issue on module facilities in specification languages.

[FL97] J. Fiadeiro and A. Lopes. Semantics of architectural connectors. *Proc. Colloq. on Formal Approaches in Software Engineering*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Lille. Springer LNCS 1214, 505–519 (1997).

[FJ90] J. Fitzgerald and C. Jones. Modularizing the formal description of a database system. *Proc. VDM'90 Conference*, Kiel. Springer LNCS 428, 198–210 (1990).

[GB80] J. Goguen and R. Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical report CSL-118, SRI International (1980).

[GB92] J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the Assoc. for Computing Machinery* 39:95–146 (1992).

[GH93] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer (1993).

[GHW82] J. Guttag, J. Horning and J. Wing. Some notes on putting formal specifications to productive use. *Science of Computer Programming* 2:53–68 (1982).

[HN94] R. Hennicker and F. Nickl. A behavioural algebraic framework for modular system design and reuse. *Selected Papers from the 9th Workshop on Specification of Abstract Data Types*, Caldes de Malavella. Springer LNCS 785, 220–234 (1994).

[KS91] B. Krieg-Brückner and D. Sannella. Structuring specifications in-the-large and in-the-small: higher-order functions, dependent types and inheritance in SPECTRAL. *Proc. Colloq. on Combining Paradigms for Software Development*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Brighton. Springer LNCS 494, 313–336 (1991).

[MA91] J. Morris and S. Ahmed. Designing and refining specifications with modules. *Proc. 3rd Refinement Workshop*, Hursley Park, 1990. Springer Workshops in Computing, 73–95 (1991).

[Mos98] T. Mossakowski. Institution-independent semantics for CASL-in-the-large. CoFI note S-8 (1998).

[Mos97] P. Mosses. CoFI: The Common Framework Initiative for algebraic specification and development. *Proc. 7th Intl. Joint Conf. on Theory and Practice of Software Development*, Lille. Springer LNCS 1214, 115–137 (1997).

[NOS95] M. Navarro, F. Orejas and A. Sanchez. On the correctness of modular systems. *Theoretical Computer Science* 140:139–177 (1995).

[Pau96] L. Paulson. *ML for the Working Programmer*, 2nd edition. Cambridge Univ. Press (1996).

[SST92] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica* 29:689–736 (1992).

[ST89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. 3rd Joint Conf. on Theory and Practice of Software Development*, Barcelona. Springer LNCS 352, 375–389 (1989).

[ST97] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing* 9:229–269 (1997).

[SW83] D. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, Borgholm. Springer LNCS 158, 413–427 (1983).

[SW98] D. Sannella and M. Wirsing. Specification languages. Chapter 8 of *Algebraic Foundations of Systems Specification* (eds. E. Astesiano, H.-J. Kreowski and B. Krieg-Brückner). Springer, to appear (1999).

[Sch87] O. Schoett. Data Abstraction and the Correctness of Modular Programming. Ph.D. thesis, report CST-42-87, Dept. of Computer Science, Univ. of Edinburgh (1987).