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 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 discussion of refinement of CASL specifications provides the setting for a presentation of the rationale behind architectural specifications. This is followed by 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 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 practical experiences [FJ90] and foundational work [Bid88], [ST89], [SST92], [BH93], we argue that mechanisms to structure specifications 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).

The aim of this paper is to present motivation, intuition and technicalities related to this concept. We provide some 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. The semantics and correctness issues of architectural specifications are discussed in Sects. 5, 6 and 7. The development process in the presence of architectural specifications is briefly discussed in Sect. 8.

Even though we present architectural specifications in the context of CASL, the ideas apply in any specification and development framework, as we mention in Sect. 9. We also briefly mention there the issue 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 built over *atomic formulae* which include strong and existential equalities, definedness formulae and predicate applications. 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: $[\![SP]\!] = \{A \in \mathbf{Mod}[\Sigma] \mid A \models \Phi\}$.

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.

2.1 Example

Here is a sequence of definitions of CASL specifications.

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

spec ADDNUM = NUM **then** **op** $plus : Num \times Num \to Num$
  vars $x, y : Num$
  axiom $plus(x, succ(y)) = succ(plus(x, y))$

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

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

We start with a signature for natural numbers, and then extend it in two ways: by a binary operation with a simple axiom and by a loosely specified binary predicate. In CODENUM we put both extensions together and then add a unary operation on *Num* with another simple axiom.

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

spec PARTCONTAINER [ELEM] =
  generated type $Cont ::= empty \mid add(Elem; Cont)?$
  pred $addable : Elem \times Cont$
  vars $x, y : Elem;\ C : Cont$
  axiom $def\ add(x, C) \Leftrightarrow addable(x, C)$
  pred $\_\_ \in \_\_ : Elem \times Cont$
  axioms $\neg\ (x \in empty);$
  $(x \in add(y, C) \Leftrightarrow x = y \vee x \in C)\ if\ addable(y, C)$
end

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 $addable(x, C)$ is satisfied. But *addable* is left unspecified at this stage. The usual membership predicate is provided as well.

spec PARTNUMCONT =
  PARTCONTAINER[CODENUM **fit** $Elem \mapsto Num$]

We instantiate PARTCONTAINER to CODENUM, with an appropriate fitting of the parameter. The result contains all the components of CODENUM together with those added by PARTCONTAINER with their profiles adjusted accordingly.

spec UNIQUENUMCONT =
  PARTNUMCONT
  then **vars** $x : Num;\ C : Cont$
  axiom $addable(x, C) \Leftrightarrow \neg\ (x \in C) \wedge \neg\ (code(x) \in C)$

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 is to specify programs. Each CASL specification should determine a class of programs that realize the specified requirements. It follows that programs must be written in a language having a semantics which assigns[1] to each program its *denotation* as a CASL structure. Then each program P determines a signature $Sig[P]$ and a structure $[\![P]\!] \in \mathbf{Mod}[Sig[P]]$. The denotation $[\![SP]\!]$ of a specification SP is a description of its admissible realizations: a program P is a *(correct) realization* of SP if $Sig[P] = Sig[SP]$ and $[\![P]\!] \in [\![SP]\!]$.

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:

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

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

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

The construction of a program to realize SP_{last} is outside the scope of CASL. 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.)

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

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 $[\![LINK_{BR}(P_1, \ldots, P_n)]\!] \in [\![SP]\!]$.

The nature of $LINK_{BR}$ depends on the nature of the programs considered. Our preferred view is that the programming language in use has reasonably powerful and flexible modularization facilities, such as those in Standard ML or Ada. 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* on its own) with formal parameters for which the actual modules $P_1, \ldots, P_n$ may be

[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.

substituted. Note that if we later replace a module P_i by another realization P'_i of SP_i, "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" w.r.t. inclusion of model classes.[2] This view is indeed possible provided that the specification-building operation OP is *constructive*: 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 $OP(SP_1, \ldots, SP_n)$. However, simple examples show that some standard specification-building operations (like the union of specifications) do not have this property. (See [HN92] 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 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, the last extension of PARTNUMCONT by an axiom for *addable* cannot be a directive to first realize PARTNUMCONT and then somehow miraculously ensure

[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.

that the predicate *addable* does indeed satisfy the axiom. 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 there are two different 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. 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* can be used to refine ordinary specifications, whether structured or not, explicitly introducing branching into the development process and structure into the final program:

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

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 $U_1, \ldots, U_n$ correctly realizing 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:

$$\begin{array}{ll} \textbf{units} & U_1 : SP_1; \\ & U_2 : SP_2 \ \textbf{given} \ U_1; \\ & \ldots \\ & U_n : SP_n \ \textbf{given} \ U_{n-1} \\ \textbf{result} & U_n \end{array}$$

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:

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

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 except that F is anonymous there. This shows how to explain architectural specifications involving "**given**" by translation to architectural specifications involving 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.

Generic unit specifications correspond to functor headings in Extended ML [ST89] and to a restricted form of Π-specifications in [SST92], cf. Spectral [KS91]. 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 indicates 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.

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.

5 Semantics of unit specifications

Consider a unit specification of the form $SP' \rightarrow SP$. In CASL, SP is implicitly viewed as an extension of SP'. We therefore assume that in each specification of the form $SP' \rightarrow SP$, SP extends SP', that is: $Sig[SP'] \subseteq Sig[SP]$ and $[\![SP]\!]|_{Sig[SP']} \subseteq [\![SP']\!]$.

To realize the specification $SP' \rightarrow SP$, we should provide a "program fragment" ΔP for $SP \setminus SP'$ that extends any realization P' of SP' to a realization $\Delta P(P')$ of SP. For all programs P' such that $[\![P']\!] \in [\![SP']\!]$, $\Delta P(P')$ must be a program that extends P' and realizes SP. Hence, semantically ΔP determines a function $[\![\Delta P]\!] \colon [\![SP']\!] \rightarrow [\![SP]\!]$ that "preserves" its argument. Consequently:

$$[\![SP' \rightarrow SP]\!] = \{F \colon [\![SP']\!] \rightarrow [\![SP]\!] \mid \text{for all } A' \in [\![SP']\!], F(A')|_{Sig[SP']} = A'\}$$

This view of program fragments as 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

[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.

on realizations of more than one (sub-)specification. Specifications of multiply-dependent units will have the form $SP_1 \times \ldots \times SP_n \rightarrow SP$. As with singly-dependent units, we assume that SP extends each of $SP_1, \ldots, SP_n$ (or equivalently, their union). We then have:

$$
\begin{aligned}
\llbracket SP_1 \times \ldots \times SP_n \rightarrow SP \rrbracket = \{F \colon \llbracket SP_1 \times \ldots \times SP_n \rrbracket \to \llbracket SP' \rrbracket \mid \\
\text{for all } \langle A_1, \ldots, A_n \rangle \in \llbracket SP_1 \times \ldots \times SP_n \rrbracket, \\
F(A_1, \ldots, A_n)\big|_{Sig[SP_i]} = A_i, \text{ for } i = 1, \ldots, n\}
\end{aligned}
$$

We have not yet defined $\llbracket SP_1 \times \ldots \times SP_n \rrbracket$. In general, not all tuples $\langle A_1, \ldots, A_n \rangle$ of structures $A_1 \in \llbracket SP_1 \rrbracket, \ldots, A_n \in \llbracket SP_n \rrbracket$ can be extended to structures in $\llbracket SP \rrbracket$: 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 $1 \leq i, j \leq n$. 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 define:

$$
\begin{aligned}
&\llbracket SP_1 \times \ldots \times SP_n \rrbracket = \\
&\qquad \{\langle A_1, \ldots, A_n \rangle \in \mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n] \mid A_1 \in \llbracket SP_1 \rrbracket, \ldots, A_n \in \llbracket SP_n \rrbracket\}
\end{aligned}
$$

6 Sharing and well-formedness

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 share their commonly-named parts. In a programming language like Standard ML, this is a part of the "type discipline" and the required sharing is (type-)checked statically.

Consider the following simple example:

$$
\begin{aligned}
&\textbf{spec } SP_0 = \textbf{sort } s \textbf{ end} \\
&\textbf{spec } SP_a = \textbf{sort } s \textbf{ op } a : s \textbf{ end} \\
&\textbf{spec } SP_b = \textbf{sort } s \textbf{ op } a, b : s \textbf{ end} \\
&\textbf{spec } SP_c = \textbf{sort } s \textbf{ op } a, c : s \textbf{ end} \\
&\textbf{spec } SP_d = \textbf{sort } s \textbf{ op } a, b, c, d : s \textbf{ axiom } d = b \ \vee \ d = c \textbf{ end}
\end{aligned}
$$

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:

$$
\begin{array}{ll}
\textbf{units} & U_b : SP_b; \\
& U_c : SP_c; \\
& F_d : SP_b \times SP_c \to SP_d
\end{array}
$$

the instantiation $F_d[U_b, U_c]$ cannot be allowed, since we have no way to ensure that the units U_b and U_c do indeed share s and a. On the other hand, consider the following unit declarations:

$$
\begin{array}{ll}
\textbf{units} & U_a : SP_a; \\
& F_b : SP_a \to SP_b; \\
& F_c : SP_a \to SP_c; \\
& F_d : SP_b \times SP_c \to SP_d
\end{array}
$$

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.

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

$$
\begin{array}{ll}
\textbf{units} & U_0 : SP_0; \\
& F_a : SP_0 \to SP_a
\end{array}
$$

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 U_0, and the constant a to the two occurrences of $F_a[U_0]$. 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 (Standard ML, Ada) the new items introduced by instantiation of generic modules are distinct for each such instantiation.

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:

$$
\begin{array}{ll}
\textbf{units} & U_0 : SP_0; \\
& F_a : SP_0 \to SP_a; \\
& U'_a = F_a[U_0]; \\
& F_b : SP_a \to SP_b;
\end{array}
$$

$$F_c : SP_a \rightarrow SP_c;$$
$$F_d : SP_b \times SP_c \rightarrow SP_d$$

In this context, $F_d[F_b[U'_a], F_c[U'_a]]$ is well-formed and captures the intention behind $F_d[\ F_b[F_a[U_0]],\ F_c[F_a[U_0]]\ ]$.

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.

7 Semantics of unit terms

An architectural specification comprises a sequence of unit declarations and 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:

units $U : SP;$
$F : SP' \rightarrow SP''$

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 as well as sharing information about them. 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 available units, as explained below.

The last example was perhaps misleadingly simple: the argument U of F came equipped with an explicit specification 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 $F[U]$, assuming that $Sig[SP] = Sig[SP']$ and $[\![SP]\!] \subseteq [\![SP']\!]$? Clearly, we know that the result realizes 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 cannot happen 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. 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 $\gamma(U)$: $\rho(U) \in [\![\gamma(U)]\!]$.[4] 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 inductively as follows:

- A unit name U is correct. (By well-formedness, U is declared in γ.) It follows that $[U]_\gamma = [\![\gamma(U)]\!]$.

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

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

This omits the use of defined units in unit terms, treated in the obvious way with information about these units extracted from their definitional terms and stored in the context as well. Some further constructs for unit terms (amalgamation, reduct/renaming, pushout-style instantiation using a fitting morphism, local unit definitions, λ-notation for generic units) are available in CASL, but these are not discussed here for lack of space.

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 which 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!

8 Refinements of architectural specifications

Section 4 indicated how a specification may be refined to an architectural specification. Architectural specifications themselves can in turn be refined by refining each of the specifications for its declared units separately. One remaining issue is to define refinements between specifications of generic units:

$$SP_1 \rightarrow SP_2 \quad \rightsquigarrow \quad SP'_1 \rightarrow SP'_2$$

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, we need that every generic unit that realizes $SP'_1 \rightarrow SP'_2$ must correctly realize $SP_1 \rightarrow SP_2$, but allowing for restrictions of mappings between structures to smaller domains. 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 most obvious $[\![SP'_2]\!] \subseteq [\![SP_2]\!]$ — we can take advantage of the fact that we are expected to apply the unit to arguments that realize SP_1.

This allows for linear development of individual units declared in an architectural specification. To allow further decomposition here, 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 with generic result units.

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.

9 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. The semantics of architectural specifications has been sketched as well, but see [CoFI98c] for all the details. This 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. 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 if the original requirements specification has been fulfilled by this design). Moreover, further developments of the units required may proceed independently from each other, 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 — see [Mos98] for details.

One issue which we have omitted above is that of *behavioural implementation* [Sch87,ST89,NOS95,ST97,BH9?]. The idea is that when realizing a specification 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.

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. 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 2. *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

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

[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).

[BH9?] M. Bidoit and R. Hennicker. Modular correctness proofs of behavioural implementations. *Acta Informatica*, to appear (199?).

[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).

[CoFI98c] CoFI Task Group on Semantics. CASL – The CoFI algebraic specification language – Semantics (version 1.0). To appear (1998).

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

[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).

[HN92] R. Hennicker and F. Nickl. A behavioural algebraic framework for modular system design and reuse. *Recent Trends in Data Type Specifications. Proc. 9th Workshop on Specification of Abstract Data Types* ADT'92, Caldes de Mavella, Springer LNCS 785, 220–234.

[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).

[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 (1998).

[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).