# 8

# Specifying the Architecture of Implementations

*Architectural specifications impose structure on implementations, whereas specification-building operations only structure the text of specifications.*

As explained in the previous chapters, the specification of a complex system may be fairly large and should be structured into coherent, easy to grasp, pieces. CASL provides a number of specification-building operations to achieve this, as detailed in Chap. 6. Moreover, generic specifications, described in Chap. 7, provide pieces of specification that are easy to reuse in different contexts, where they can be adapted as desired by instantiating them.

Specification-building operations and generic specifications are useful *to structure the text of the specification* of the system under consideration. However, the models of a structured specification have no more structure than do those of a flat, unstructured, specification. Indeed, most examples given in the previous chapters could have been structured differently, with the same meaning (i.e., with the same models). Structured specifications are usually adequate at the requirements stage, where the focus is on the expected overall properties of the system under consideration.

In contrast, the aim of architectural specifications is *to prescribe the intended architecture of the implementation* of the system. Architectural specifications provide the means for specifying the various *components* from which the system will be built, and describing how these components are to be assembled to provide an implementation of the system of interest. At the same time, they allow the task of implementing a system to be split into independent, clearly-specified sub-tasks. Thus, architectural specifications are essential at the design stage, where the focus is on how to factor the implementation of the system into components.

The aim of this chapter is to discuss and illustrate both the role of architectural specifications and how to express them in CASL.

M. Bidoit and P.D. Mosses: CASL User Manual, LNCS 2900, pp. 93–109, 2004.

The idea underlying architectural specifications is that eventually in the process of systematic development of modular software from specifications, components are implemented as software modules in some chosen programming language. However, this step is beyond the scope of specification formalisms, so in CASL and in this chapter we identify components with models (and with functions from models to models, in the case of generic components). The modular structure of the software under development, as described by an architectural specification, is therefore captured here simply as an explicit, structural way to build CASL models.

*The examples in this chapter are artificially simple.*

Architectural specifications, and more generally component-oriented approaches, are intended for relatively large systems. In this chapter, however, we have to rely on simple small examples to illustrate and explain CASL architectural specification concepts and constructs. After reading this chapter, the reader is encouraged to study Chap. 13, which provides realistic examples of the use of architectural specifications. A more detailed account of the rationale behind architectural specifications in the context of formal software development by stepwise refinement can be found in [11].

The following structured specifications will be referred to later in this chapter when illustrating CASL architectural specifications:

**spec** COLOR = %{ As defined in Chap. 3, p. 37 }%
**spec** NATURAL_ORDER = %{ As defined in Chap. 3, p. 38 }%
**spec** NATURAL_ARITHMETIC = %{ As defined in Chap. 3, p. 38 }%

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

**spec** CONT [ELEM] =
**generated type** $Cont[Elem] ::= empty \mid insert(Elem;\ Cont[Elem])$
**preds** $\_\_is\_empty : Cont[Elem];$
$\_\_is\_in\_\_ : Elem \times Cont[Elem]$
**ops** $choose : Cont[Elem] \to ? Elem;$
$delete : Elem \times Cont[Elem] \to Cont[Elem]$
$\forall e, e' : Elem;\ C : Cont[Elem]$
- $empty\ is\_empty$
- $\neg\ insert(e, C)\ is\_empty$
- $\neg\ e\ is\_in\ empty$
- $e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$
- $def\ choose(C) \Leftrightarrow \neg\ C\ is\_empty$
- $def\ choose(C) \Rightarrow choose(C)\ is\_in\ C$
- $e\ is\_in\ delete(e', C) \Leftrightarrow (e\ is\_in\ C \wedge \neg(e = e'))$

**end**

**spec** CONT_DIFF [ELEM] =
CONT [ELEM]
**then** **op** *diff* : $Cont[Elem] \times Cont[Elem] \to Cont[Elem]$
$\forall e : Elem;\ C, C' : Cont[Elem]$
• $e\ is\_in\ diff(C, C') \Leftrightarrow (e\ is\_in\ C \wedge \neg(e\ is\_in\ C'))$
**end**

**spec** REQ = CONT_DIFF [NATURAL_ORDER]

**spec** FLAT_REQ =
**free type** $Nat ::= 0 \mid suc(Nat)$
**pred** $\_\_ < \_\_ : Nat \times Nat$
**generated type** $Cont[Nat] ::= empty \mid insert(Nat;\ Cont[Nat])$
**preds** $\_\_is\_empty : Cont[Nat]$;
$\_\_is\_in\_\_ : Nat \times Cont[Nat]$
**ops** $choose : Cont[Nat] \to ? Nat$;
$delete : Nat \times Cont[Nat] \to Cont[Nat]$;
$diff : Cont[Nat] \times Cont[Nat] \to Cont[Nat]$
$\forall e, e' : Nat;\ C, C' : Cont[Nat]$
• $0 < suc(e)$
• $\neg(e < 0)$
• $suc(e) < suc(e') \Leftrightarrow e < e'$
• $empty\ is\_empty$
• $\neg\ insert(e, C)\ is\_empty$
• $\neg\ e\ is\_in\ empty$
• $e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$
• $def\ choose(C) \Leftrightarrow \neg\ C\ is\_empty$
• $def\ choose(C) \Rightarrow choose(C)\ is\_in\ C$
• $e\ is\_in\ delete(e', C) \Leftrightarrow (e\ is\_in\ C \wedge \neg(e = e'))$
• $e\ is\_in\ diff(C, C') \Leftrightarrow (e\ is\_in\ C \wedge \neg(e\ is\_in\ C'))$
**end**

## 8.1 Architectural Specifications

Let's assume in the following that REQ describes our requirements about the system to be implemented. First, note that both REQ and FLAT_REQ have the same models, which illustrates our point about the fact that the CASL specification-building operations are merely facilities to structure the text of specifications into coherent units.

*An architectural specification consists of a list of unit declarations, specifying the required components, and a result part, indicating how they are to be combined.*

**arch spec** SYSTEM =
**units** $N$ : NATURAL_ORDER;
$C$ : CONT [NATURAL_ORDER] **given** $N$;
$D$ : CONT_DIFF [NATURAL_ORDER] **given** $C$
**result** $D$

The SYSTEM architectural specification is intended to prescribe a specific architecture for implementing the system specified by REQ.

The first part, introduced by the keyword **units**, indicates that we require the implementation of our system to be made of three components $N$, $C$, and $D$. The second part, introduced by the keyword **result**, indicates that the component $D$ provides the desired implementation.

Each component is provided with its specification. The line:

$N$ : NATURAL_ORDER

declares a component $N$ specified by NATURAL_ORDER, which means simply that $N$ should be a model of NATURAL_ORDER.

The line:

$C$ : CONT [NATURAL_ORDER] **given** $N$

declares a component $C$ which, given the previously declared component $N$, provides a model of CONT [NATURAL_ORDER]. It is essential to understand that the component $C$ must *expand* the assumed component $N$ into a model of CONT [NATURAL_ORDER], which means that $C$ reduced to the signature of NATURAL_ORDER must be equal to $N$. This property reflects the fact that a software module is supposed to use what it is given exactly as supplied, without altering it.

Similarly, the line:

$D$ : CONT_DIFF [NATURAL_ORDER] **given** $C$

declares a component $D$ which, given the component $C$, expands it into a model of CONT_DIFF [NATURAL_ORDER].

The final result is therefore simply $D$. (More complex examples of result expressions will be illustrated in examples below.)

As in the rest of CASL, visibility is linear in architectural specifications, meaning that any component must be declared before being used (e.g., the component $N$ should be declared before being referred to by '**given** $N$' in the declaration of the component $C$ in the architectural specification SYSTEM).

Component names (such as $N$, $C$, and $D$ in SYSTEM) are *local* to the architectural specification where they are declared, and are not visible outside it.

> *There can be several distinct architectural choices for the same requirements specification.*

**arch spec** SYSTEM_1 =
**units** $N$ : NATURAL_ORDER;
$CD$ : CONT_DIFF [NATURAL_ORDER] **given** $N$
**result** $CD$

The architectural specifications SYSTEM and SYSTEM_1 both provide models of REQ. However, the former insists on an implementation made of three components, while the latter insists on an implementation made of two components. Thus the architectural specification SYSTEM_1 corresponds to a different architectural choice for implementing our REQ specification. Of course, further design for implementing the component $CD$ of SYSTEM_1 may lead to splitting this implementation task exactly as in SYSTEM above. However, there are also other possibilities, including for instance an architectural design where we would split the task of implementing $CD$ into two different tasks, one for implementing containers with all their operations (including *diff*) except *delete*, the other for implementing *delete* by means of *diff* and other operations.

> *Each unit declaration listed in an architectural specification corresponds to a separate implementation task.*

For instance, in the architectural specification SYSTEM, the task of providing a component $D$ expanding $C$ and implementing CONT_DIFF [NATURAL_ORDER] is independent from the tasks of providing implementations $N$ of NATURAL_ORDER and $C$ of CONT [NATURAL_ORDER] **given** $N$. Hence, when providing the component $D$, one cannot make any further assumption on how the component $C$ is (or will be) implemented, besides what is expressly ensured by its specification.

To understand this, let us consider again the requirements specification REQ (or its variant FLAT_REQ). Among its models, there is one where containers are implemented by sorted lists (in increasing order, without repetitions), and in this model we can choose to implement *diff* by the following algorithm:

$$\begin{aligned} \mathit{diff}(L, L') = {} & \mathit{nil}\ \mathit{when}\ L = \mathit{nil} \\ & \mathit{else}\ L\ \mathit{when}\ L' = \mathit{nil} \\ & \mathit{else}\ \mathit{insert}(\mathit{head}(L), \mathit{diff}(\mathit{tail}(L), L'))\ \mathit{when}\ \mathit{head}(L) < \mathit{head}(L') \\ & \mathit{else}\ \mathit{diff}(\mathit{tail}(L), \mathit{tail}(L'))\ \mathit{when}\ \mathit{head}(L) = \mathit{head}(L') \\ & \mathit{else}\ \mathit{diff}(L, \mathit{tail}(L')) \end{aligned}$$

In this model, however, we rely on knowledge about the implementation of containers to decide how to implement *diff* – which is fine, since both are simultaneously implemented in the same component. In contrast, in the architectural specification SYSTEM, we request that containers are to be implemented in the component $C$ while *diff* is to be provided by a separate component $D$. Imposing that the component $D$ can be developed independently of the component $C$ means that for $D$ it is no longer possible to implement *diff* as sketched above, since this specific implementation choice may not be compatible with an independently chosen realization for $C$ (where containers may be implemented by bags, for instance). Hence an implementation of *diff* in the component $D$ can only rely on the operations provided by $C$ (e.g., *choose* and *delete*); this may turn out to be less efficient for some particular realization of $C$, but should be compatible with any independently chosen realization for $C$ (bags, for instance). In the case of the architectural specification SYSTEM_1, since both containers and the *diff* operation are implemented in the same component $CD$, we can of course decide to implement containers by ordered lists without repetitions and *diff* as sketched above.

Thus the component $D$ should expand *any* given implementation $C$ of CONT [NATURAL_ORDER] and provide an implementation of CONT_DIFF [NATURAL_ORDER], which is tantamount to providing a *generic* implementation $G$ of CONT_DIFF [NATURAL_ORDER] which takes the particular implementation of CONT [NATURAL_ORDER] as a parameter to be expanded. Then we obtain $D$ by simply applying $G$ to $C$.

Genericity here arises from the independence of the developments of $C$ and $D$, rather than from the desire to build multiple implementations of CONT_DIFF [NATURAL_ORDER] using different implementations of CONT [NATURAL_ORDER]. This is reflected by the fact that $G$ is left implicit in the architectural specification SYSTEM.

> *A unit can be implemented only if its specification is a conservative extension of the specifications of its given units.*

For instance, the component $D$ can exist only if the specification CONT_DIFF [NATURAL_ORDER] is a conservative extension of CONT [NATURAL_ORDER], i.e., if any model of the latter specification can be expanded into a model of the former one, which is indeed the case here. Similarly, the component $C$ can exist since CONT [NATURAL_ORDER] is a conservative extension of NATURAL_ORDER.

Consider now the following variant of CONT_DIFF [NATURAL_ORDER] and the associated variant of the architectural specification SYSTEM.

**spec** CONT_DIFF_1 =
CONT [NATURAL_ORDER]
**then** **op** *diff* : $Cont[Nat] \times Cont[Nat] \to Cont[Nat]$
$\forall x, y : Nat;\ C, C' : Cont[Nat]$
- $diff(C, empty) = C$
- $diff(empty, C') = empty$
- $diff(insert(x, C), insert(y, C')) =$
  $insert(x, diff(C, insert(y, C')))$ *when* $x < y$
  *else* $diff(C, C')$ *when* $x = y$
  *else* $diff(insert(x, C), C')$
- $x$ *is_in* $diff(C, C') \Leftrightarrow (x$ *is_in* $C \wedge \neg(x$ *is_in* $C'))$

**end**

**arch spec** INCONSISTENT =
**units** $N$ : NATURAL_ORDER;
$C$ : CONT [NATURAL_ORDER] **given** $N$;
$D$ : CONT_DIFF_1 **given** $C$
**result** $D$

The specification CONT_DIFF_1 is consistent (has some models, for instance sorted lists, in increasing order, without repetitions), but is not a conservative extension of CONT [NATURAL_ORDER] (since, for instance, a model of CONT [NATURAL_ORDER] where containers are realized by arbitrary lists, possibly with repetitions, cannot be expanded into a model of CONT_DIFF_1 – in that case, the last two axioms are contradictory). As a consequence, in the architectural specification INCONSISTENT, the specification of the component $D$ is inconsistent, since no component can expand *all* implementations $C$ of CONT [NATURAL_ORDER] into models of CONT_DIFF_1. The architectural specification INCONSISTENT is therefore itself inconsistent.

To summarize, architectural specifications not only prescribe the intended architecture of the implementation of the system, but they also ensure that the specified components can be developed independently of each other (which imposes a certain degree of genericity for these components).

## 8.2 Generic Components

*Genericity of components can be made explicit in architectural specifications.*

**arch spec** SYSTEM_G =
**units** $N$ : NATURAL_ORDER;
$F$ : NATURAL_ORDER → CONT [NATURAL_ORDER];
$G$ : CONT [NATURAL_ORDER] → CONT_DIFF [NATURAL_ORDER]
**result** $G[F[N]]$

The architectural specification SYSTEM_G is a variant of SYSTEM; here we choose to specify the second and third components as explicit *generic components*.

The line:

$F$ : NATURAL_ORDER → CONT [NATURAL_ORDER]

declares a generic component $F$. Given any component implementing (i.e., model of) NATURAL_ORDER, $F$ should expand it into an implementation of CONT [NATURAL_ORDER]. The models of the generic-component specification NATURAL_ORDER → CONT [NATURAL_ORDER] are functions that map any model of NATURAL_ORDER to a model of CONT [NATURAL_ORDER]. These functions are required to be persistent, meaning that the result model expands the argument model.

The third component $G$ is also specified as a generic component: given any implementation of CONT [NATURAL_ORDER], $G$ should expand it into an implementation of CONT_DIFF [NATURAL_ORDER].

Hence the whole system is obtained by the composition of applications $G[F[N]]$, as described in the result part. In CASL, such combinations of components are called *unit terms*. (More complex examples of unit terms will be illustrated in examples below.)

The component $C$ of SYSTEM corresponds to the application $F[N]$ in SYSTEM_G, and similarly the component $D$ in SYSTEM corresponds to $G[C]$, i.e., to $G[F[N]]$ in SYSTEM_G.

The models of a specification of the form *SP1* → *SP2* are generic components *GC* that should always expand their argument into a model of the target specification. This only makes sense as long as the signature of the target specification contains the signature of *SP1*. This is why in CASL, *SP2* is always considered as an implicit extension of *SP1*, and *SP1* → *SP2* abbreviates *SP1* → **{** *SP1* **then** *SP2* **}**.[1] Moreover, since the generic component *GC*

[1] When *SP2* is already defined as an extension of *SP1*, as it is the case for instance here for CONT_DIFF [NATURAL_ORDER], *SP2* is equivalent to *SP1* **then** *SP2*.

should expand *any* model of *SP1*, the specification *SP1* → *SP2* is *consistent* (i.e., has some models) if and only if the specification *SP1* **then** *SP2* is a conservative extension of *SP1*. Forgetting this fact is a potential source of inconsistent specifications of generic components in architectural specifications. For instance, the specification CONT [NATURAL_ORDER] → CONT_DIFF_1 is inconsistent, for the reasons explained at the end of the previous section.

> *A generic component may be applied to an argument richer than required by its specification.*

**arch spec** SYSTEM_A =
**units** *NA* : NATURAL_ARITHMETIC;
*F* : NATURAL_ORDER → CONT [NATURAL_ORDER];
*G* : CONT [NATURAL_ORDER] → CONT_DIFF [NATURAL_ORDER]
**result** *G* [*F* [*NA*]]

The above architectural specification SYSTEM_A is a variant of SYSTEM_G. Here we require a component *NA* implementing the specification NATURAL_ARITHMETIC, instead of a component *N* implementing NATURAL_ORDER as in SYSTEM_G (perhaps because we know that such a component is already available in some collection of previously-implemented components.)

The generic component *F* requires a component fulfilling the specification NATURAL_ORDER, but can of course be applied to a richer argument, as in *F* [*NA*]. A similar reasoning applies to *G*.

More generally, a generic component can be applied to any component (or to any unit term) that can be reduced along some morphism to an argument of the required 'type' (i.e., to a model of the required specification). When necessary, a fitting symbol map can be used to describe the correspondence between the symbols provided by the argument and those expected by the generic component. We do not detail here the technicalities related to these fitting symbol maps, since they are quite similar to those used in instantiations of generic specifications and the notations are the same.

As a last remark, note that, similarly to what happens when instantiating a generic specification by an argument specification, when a generic component is applied to an argument richer than required, the extra symbols are kept in the result. Hence the result of the architectural specification SYSTEM_A above contains also the interpretations of the arithmetic and ordering operations on natural numbers, as they are provided by the component *NA*. This means in particular that the implementations described by SYSTEM_A have a larger signature than the ones described by SYSTEM_G.

*Specifications of components can be named for further reuse.*

**unit spec** CONT_COMP = ELEM → CONT [ELEM]

**unit spec** DIFF_COMP = CONT [ELEM] → CONT_DIFF [ELEM]

**arch spec** SYSTEM_G1 =
**units** $N$ : NATURAL_ORDER;
$F$ : CONT_COMP;
$G$ : DIFF_COMP
**result** $G[F[N]]$

In the above example, we give the name CONT_COMP to the specification (of generic components) ELEM → CONT [ELEM]. Similarly, we give the name DIFF_COMP to the specification CONT [ELEM] → CONT_DIFF [ELEM]. Then both named specifications can be reused in the architectural specification SYSTEM_G1 which is similar to the architectural specification SYSTEM_G.

In the architectural specification SYSTEM_G1, we use again the fact that the generic component $F$ can be applied to richer arguments than models of ELEM (and similarly for $G$). Since ELEM is more general (has more models) than NATURAL_ORDER, there are potentially fewer possibilities for implementing the generic component specified by CONT_COMP (which should be compatible with any model of ELEM) than there are for implementing the generic component specified by NATURAL_ORDER → CONT [NATURAL_ORDER] (which only needs to be compatible with models of NATURAL_ORDER; a similar argument holds for DIFF_COMP). As a consequence, the architectural specifications SYSTEM_G and SYSTEM_G1 do not describe the same implementations of the requirements specification REQ.

*Both named and unnamed specifications can be used to specify components.*

**unit spec** DIFF_COMP_1 =
CONT [ELEM] → { **op** *diff* : $Cont[Elem] \times Cont[Elem] \to Cont[Elem]$
$\forall e : Elem;\ C, C' : Cont[Elem]$
• $e$ *is_in* $diff(C, C') \Leftrightarrow$
$(e$ *is_in* $C \wedge \neg(e$ *is_in* $C'))$ }

So far we have always used named (structured) specifications to specify components. unnamed specifications can be used as well, as illustrated by the above variant DIFF_COMP_1 of DIFF_COMP. Here, for the sake of the example, we directly specify the *diff* operation instead of referring to the

named specification CONT_DIFF. Remember that in a specification of a generic component of the form *SP1* $\rightarrow$ *SP2*, *SP2* is always considered as an implicit extension of *SP1*, which explains why the above example is well-formed.

> *Specifications of generic components should not be confused with generic specifications.*

Generic specifications naturally give rise to specifications of generic components, which can be named for later reuse, as illustrated above by CONT_COMP. However, the reader should not confuse a generic specification (which is nothing other than a piece of specification that can easily be adapted by instantiation) with the corresponding specification of a generic component: the latter cannot be instantiated, it is the specified *generic component* which gets *applied* to suitable components.

Conservative extensions of the form '**spec** *SP2* = *SP1* **then** *SP*' also naturally give rise to specifications of generic components of the form *SP1* $\rightarrow$ *SP2*, as illustrated by DIFF_COMP above.

> *A generic component may be applied more than once in the same architectural specification.*

**arch spec** OTHER_SYSTEM =
**units** $N$ : NATURAL_ORDER;
$C$ : COLOR;
$F$ : CONT_COMP
**result** $F\,[N]$ **and** $F\,[C$ **fit** $Elem \mapsto RGB]$

The above architectural specification requires a component $N$ specified by NATURAL_ORDER, a component $C$ specified by COLOR, and a generic component $F$ specified by CONT_COMP. Then, as described by the result part, the desired system is obtained by applying $F$ to $N$ and applying $F$ to $C$ (in this case, an explicit fitting symbol map is necessary, since COLOR exports two sorts *RGB* and *CMYK*). Finally both application results are combined, which is expressed by '**and**'.

Apart from '**free**', all specification-building operations for structured specifications have natural counterparts at the level of components, which are expressed using the same keywords.[2] The reader should remember that specification-building operations work with specifications defining classes of

[2] The situation is however a bit different with specification extensions, which lead to specifications of generic components, as explained above, or to specifications of components expanding a given component, as illustrated in the previous section.

models (e.g., union of specifications, denoted by '**and**'), while in architectural specifications we work with individual models (corresponding to components, as is the case here in OTHER_SYSTEM where '**and**' is used to combine the two components $F\,[N]$ and $F\,[C$ **fit** $Elem \mapsto RGB]$ ).

Hence renaming and hiding also have natural counterparts at the level of components. For instance, remember that the implementations described by SYSTEM_A have a larger signature than the implementations described by SYSTEM_G. It is however easy to modify the result part of SYSTEM_A if what we really want are implementations with the same signature as the implementations described by SYSTEM_G: one has just to hide the extra symbols resulting from the component *NA* as follows:

**result** $G\,[F\,[NA]]$ **hide** *1*, $\_ + \_$, $\_ * \_$

or:

**result** $G\,[F\,[NA$ **hide** *1*, $\_ + \_$, $\_ * \_]]$

Symbol maps used in renaming and hiding at the level of components follow the same rules as symbol maps used in renaming and hiding at the level of structured specifications (see Chap. 6).

*Several applications of the same generic component is different from applications of several generic components with similar specifications.*

**arch spec** OTHER_SYSTEM_1 =
**units** $N$ : NATURAL_ORDER;
$C$ : COLOR;
$FN$ : NATURAL_ORDER → CONT [NATURAL_ORDER];
$FC$ : COLOR → CONT [COLOR **fit** $Elem \mapsto RGB$ ]
**result** $FN\,[N]$ **and** $FC\,[C]$

The above architectural specification OTHER_SYSTEM_1 is a variant of OTHER_SYSTEM. However, in OTHER_SYSTEM, we insist on choosing one implementation for containers in the generic component $F$, and then we apply it twice, first to a component $N$ implementing NATURAL_ORDER, and then to a component $C$ implementing COLOR. In contrast, in OTHER_SYSTEM_1, we may choose two different implementations for containers, one for containers of natural numbers in the component $FN$ and another one for containers of colors in the component $FC$.

The architectural specifications OTHER_SYSTEM and OTHER_SYSTEM_1 are therefore similar but clearly different. Neither is better than the other: each corresponds to a different architectural decision, and selecting one rather than the other is a matter of architectural design. Components that are more widely reusable tend to have less efficient implementations, in general. (Here

the fact that *RGB* has only three values might be exploited in *FC* to give a more space-efficient representation of containers than is possible for *FN*.)

*Generic components may have more than one argument.*

**unit spec** SET_COMP = ELEM → GENERATED_SET [ELEM]

**spec** CONT2SET [ELEM] =
CONT [ELEM] **and** GENERATED_SET [ELEM]
**then** **op** *elements_of __* : *Cont*[*Elem*] → *Set*
∀*e* : *Elem*; *C* : *Cont*[*Elem*]
• *elements_of empty* = *empty*
• *elements_of insert*(*e*, *C*) = {*e*} ∪ *elements_of C*
**end**

**arch spec** ARCH_CONT2SET_NAT =
**units** *N* : NATURAL_ORDER;
*C* : CONT_COMP;
*S* : SET_COMP;
*F* : CONT [ELEM] × GENERATED_SET [ELEM]
→ CONT2SET [ELEM]
**result** *F* [*C* [*N*]] [*S* [*N*]]

The architectural specification ARCH_CONT2SET_NAT requires a component *N* implementing NATURAL_ORDER, a generic component *C* implementing CONT_COMP, i.e., containers, and a generic component *S* implementing SET_COMP, i.e., sets. Then it further requires a generic component *F* that, given *any pair of compatible* models *X* of CONT [ELEM] and *Y* of GENERATED_SET [ELEM], expands them into a model of CONT2SET [ELEM].

Models *X* and *Y* are said to be *compatible* if they share a common interpretation for all symbols they have in common. Here the only symbol they have in common is the sort *Elem*, so the compatibility condition means that *X* and *Y* have the same carrier set for *Elem*. Compatibility is a natural condition, since it is obviously necessary that *X* and *Y* have a common interpretation of their common symbols, otherwise they cannot be both expanded to the same more complex component.

The result is then produced by applying *F* to the pair obtained by applying *C* to *N* and *S* to *N*. Here the pair of arguments *C* [*N*] and *S* [*N*] are obviously compatible, since their common symbols (the sort *Nat* equipped with the operations *0* and *suc*) all come from the *same* component *N* which provides their interpretation, which is expanded (hence cannot be modified) in *C* [*N*] and in *S* [*N*], thus compatibility is guaranteed.

*Open systems can be described by architectural specifications using generic unit expressions in the result part.*

**arch spec** ARCH_CONT2SET =
**units** $C$ : CONT_COMP;
$S$ : SET_COMP;
$F$ : CONT [ELEM] × GENERATED_SET [ELEM]
→ CONT2SET [ELEM]
**result** $\lambda X$ : ELEM • $F\,[C\,[X]]\,[S\,[X]]$

**arch spec** ARCH_CONT2SET_USED =
**units** $N$ : NATURAL_ORDER;
$CSF$ : **arch spec** ARCH_CONT2SET
**result** $CSF\,[N]$

So far our example architectural specifications have described 'closed', stand-alone systems where all components necessary to build the desired system were declared in the architectural specification of interest. In CASL, it is however possible to describe 'open' systems, i.e., systems made of some components that would require further components to provide a 'closed' system. This is illustrated by the architectural specification ARCH_CONT2SET which describes a system with a generic component $C$ implementing containers, a generic component $S$ implementing sets, and a generic component $F$ that expands them to provide an implementation of the operation *elements_of*. The result part is therefore a generic structured component, i.e., an 'open' system, which, given any component $X$ implementing ELEM, provides a system built by applying $F$ to the pair made of the applications of $C$ to $X$ and of $S$ to $X$. In CASL, '$\lambda$' is input as '`lambda`'.

As illustrated by ARCH_CONT2SET_USED, we can then describe a 'closed' system made of a component $N$ implementing NATURAL_ORDER, and of an 'open' system *CSF* specified by ARCH_CONT2SET, which is then applied to $N$ in the result part.

## 8.3 Writing Meaningful Architectural Specifications

In the previous sections we have already pointed out potential sources of inconsistent specifications of components. Another issue which deserves some attention when designing an architectural specification is *compatibility* between components (or, more generally, unit terms) that are to be combined together, either by '**and**', or by fitting them to a generic component with multiple arguments.

*When components are to be combined, it is best to check that any shared symbol originates from the same non-generic component.*

**arch spec** ARCH_CONT2SET_NAT_1 =
**units** $N$ : NATURAL_ORDER;
$C$ : CONT_COMP;
$S$ : SET_COMP;
$G$ : **{** CONT [ELEM] **and** GENERATED_SET [ELEM] **}**
$\rightarrow$ CONT2SET [ELEM]
**result** $G\,[C\,[N]$ **and** $S\,[N]$ **fit** $Cont[Elem] \mapsto Cont[Nat]\,]$

The architectural specification ARCH_CONT2SET_NAT_1 is a variant of ARCH_CONT2SET_NAT where, instead of declaring a generic component $F$ with two arguments, we now declare a generic component $G$ with a single argument, which must be a model of the specification **{** CONT [ELEM] **and** GENERATED_SET [ELEM] **}**, obtained as the union of the two (trivially instantiated) specifications of containers and sets.

As a consequence, to obtain the desired system, in the result part we apply the generic component $G$ to the combination (denoted by '**and**') of $C$ applied to $N$ and of $S$ applied to $N$.[3] This combination makes sense only if both $C\,[N]$ and $S\,[N]$ share the same interpretation of their common symbols. Here their common symbols (the sort *Nat* equipped with the operations *0* and *suc*) all come from the *same* component $N$ which provides their interpretation, which is expanded (hence cannot be modified) in $C\,[N]$ and in $S\,[N]$, thus compatibility is guaranteed.

There is a clear analogy here between the application of the generic component $F$ with multiple arguments in ARCH_CONT2SET_NAT and the combination of $C\,[N]$ and $S\,[N]$ in ARCH_CONT2SET_NAT_1: in both cases the result is meaningful because we can trace shared symbols like the sort *Nat* and the operations *0* and *suc* to a single component $N$ introducing them.

Let us emphasize again that compatibility is a natural requirement: since each unit declaration corresponds to a separate implementation task (and hence each unit subterm to an independently developed subsystem), obviously the combination of components or unit terms makes sense only when some compatibility conditions are fulfilled.

Let us now consider an example where the compatibility condition is violated.

[3] In the application of the generic component $G$ we need an explicit fitting symbol map since otherwise the sort $Cont[Elem]$ can ambiguously be mapped to either $Cont[Nat]$ or $Set$.

**arch spec** WRONG_ARCH_SPEC =
**units** $CN$ : CONT [NATURAL_ORDER];
$SN$ : GENERATED_SET [NATURAL_ORDER];
$F$ : CONT [ELEM] × GENERATED_SET [ELEM]
→ CONT2SET [ELEM]
**result** $F$ [$CN$] [$SN$]

The architectural specification WRONG_ARCH_SPEC is a variant of ARCH_CONT2SET_NAT where, instead of requiring a component $N$ implementing NATURAL_ORDER and two generic components implementing containers and sets respectively, we just require a component $CN$ implementing containers of natural numbers and a component $SN$ implementing sets of natural numbers. However, then the application $F$ [$CN$] [$SN$] makes no sense since there is no way to ensure that the common symbols of $CN$ and $SN$ have the same interpretation. It may indeed be the case that natural numbers are interpreted in some way in $CN$ and in a different way in $SN$, which makes the application of $F$ impossible. (Hence a similar problem would arise if one would use the combination of components '$CN$ **and** $SN$'.)

Let us now consider a more complex example.

**arch spec** BADLY_STRUCTURED_ARCH_SPEC =
**units** $N$ : NATURAL_ORDER;
$A$ : NATURAL_ORDER → NATURAL_ARITHMETIC;
$C$ : CONT_COMP;
$S$ : SET_COMP;
$F$ : CONT [ELEM] × GENERATED_SET [ELEM]
→ CONT2SET [ELEM]
**result** $F$ [$C$ [$A$ [$N$]]] [$S$ [$A$ [$N$]]]

The architectural specification BADLY_STRUCTURED_ARCH_SPEC is a variant of ARCH_CONT2SET_NAT where, in addition to the component $N$ implementing NATURAL_ORDER, we require a generic component $A$ which is used to expand $N$ into an implementation of NATURAL_ARITHMETIC. In the architectural specification ARCH_CONT2SET_NAT, the compatibility condition in the application $F$ [$C$ [$N$]] [$S$ [$N$]] was easy to discharge. Here, in the result unit term $F$ [$C$ [$A$ [$N$]]] [$S$ [$A$ [$N$]]] of BADLY_STRUCTURED_ARCH_SPEC, we apply $F$ to the pair made of $C$ [$A$ [$N$]] and $S$ [$A$ [$N$]]. In this case only a semantic analysis can ensure that these two arguments are compatible, since the common symbols cannot be traced to the same non-generic component, but only to two applications of the same generic component $A$ to similar arguments. (Actually the arguments are just the same here, but in general checking this would require non-trivial semantic reasoning.)

It is advisable to use unit terms where compatibility can be checked by a simple static analysis. CASL provides additional constructs which make it easy to follow this recommendation, as explained below.

*Auxiliary unit definitions or local unit definitions may be used to avoid repetition of generic unit applications.*

```
arch spec WELL_STRUCTURED_ARCH_SPEC =
units N  : NATURAL_ORDER;
      A  : NATURAL_ORDER → NATURAL_ARITHMETIC;
      AN = A[N];
      C  : CONT_COMP;
      S  : SET_COMP;
      F  : CONT[ELEM] × GENERATED_SET[ELEM]
                → CONT2SET[ELEM]
result F[C[AN]][S[AN]]

arch spec ANOTHER_WELL_STRUCTURED_ARCH_SPEC =
units N : NATURAL_ORDER;
      A : NATURAL_ORDER → NATURAL_ARITHMETIC;
      C : CONT_COMP;
      S : SET_COMP;
      F : CONT[ELEM] × GENERATED_SET[ELEM]
                → CONT2SET[ELEM]
result local AN = A[N] within F[C[AN]][S[AN]]
```

The problem illustrated in BADLY_STRUCTURED_ARCH_SPEC can be fixed easily. An auxiliary unit definition may be used to avoid the repetition of generic unit applications, such as '$AN = A[N]$' in WELL_STRUCTURED_ARCH_SPEC. An alternative is to make the definition of $AN$ local to the result unit term, as illustrated in ANOTHER_WELL_STRUCTURED_ARCH_SPEC. In both cases common symbols can be traced to a non-generic unit, and compatibility can be checked by an easy static analysis.