A Simple Refinement Language for CASL [⋆]

Till Mossakowski[1], Donald Sannella[2], and Andrzej Tarlecki[3]

[1] BISS, Department of Computer Science, University of Bremen
[2] LFCS, School of Informatics, University of Edinburgh, Edinburgh, UK
[3] Institute of Informatics, Warsaw University and
Institute of Computer Science, PAS, Warsaw, Poland

Abstract. We extend CASL architectural specifications with a simple refinement language that allows the formalization of developments as refinement trees. The essence of the extension is to allow refinements of unit specifications in CASL architectural specifications.

1 Introduction

The standard development paradigm of algebraic specification [1] postulates that the development begins with a formal *requirement specification* SP_0 (extracted from a software project's informal requirements) that fixes only expected properties but ideally says nothing about implementation issues; this is to be followed by a number of *refinement* steps that fix more and more details of the design, until a specification SP_n is obtained that is detailed enough that its conversion into a program P is relatively straightforward:

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n \cdots\!> P$$

Actually, this picture is too simple in practice: for complex software systems, it is necessary to reduce complexity by introducing branching points into the chain of refinement steps, so that the resulting implementation tasks can be resolved independently, e.g. by different developers. CASL architectural specifications [3, 8] have been designed for this purpose, based on the insight that structuring of implementations is different from structuring specifications [9].

However, CASL architectural specifications allow for the specification of individual branching points only. In this work, we extend CASL with a simple refinement language that adds the means to formalize whole developments in the form of *refinement trees.*

As it stands, this is not a formal proposal for an extension to CASL and some of the details of syntax etc. are rather tentative. It is intended as a basis for further discussion and experimentation, that may eventually lead to such a proposal.

[⋆] This work has been partially supported by KBN grant 7T11C 002 21, European projects AGILE IST-2001-32747 (AT) and MRG IST-2001-33149 (DS), British–Polish Research Partnership Programme (DS, AT), and German DFG project KR 1191/5-2 (TM).

The paper is organized as follows. Section 2 recalls Casl, Sect. 3 introduces simple refinements and Sect. 4 branching refinements. These are related to constructor implementations in Sect. 5, which leads to the question of how programs are modelled in Casl. This is addressed in Sect. 6. Sections 7 and 8 describe the syntax and semantics of our proposed refinement language; to facilitate understanding, we first deal with a simpler version in Sect. 7 before exposing the full complexity of the proposed refinement language and its semantics in Sect. 8. Finally, Sect. 9 sketches a larger example and Sect. 10 concludes the paper.

2 Casl Preliminaries

Casl [2, 8] consists of several major *layers*, which are quite independent and may be understood and used separately:

Basic specifications are written in many-sorted first-order logic, extended by subsorting, partial functions and induction axioms for datatypes. Indeed, the details are quite irrelevant here as long as a basic specification determines a *signature* together with a set of *axioms*. The semantics of a basic specification is then given by the signature and the class of all *models* that satisfy the axioms. Formally, $[\![\langle\Sigma, \Phi\rangle]\!] = \{M \in \mathbf{Mod}(\Sigma) \mid M \models \Phi\}$, where $\mathbf{Mod}(\Sigma)$ is the class of all Casl models over the signature Σ.

Structured specifications allow specifications to be built from basic specifications by the use of translation, reduction, union, and extension, as well as generic (parametrized) and named specifications; semantics of structured specifications is given in terms of signatures and model classes, as for basic specifications.

Architectural specifications describe the structure of an implementation by defining how it may be constructed out of software components (*units*) that satisfy given specifications. These *unit specifications* describe self-contained units (models, as above), or generic units (corresponding to parametrized programs) mapping such models to other models.

Casl admits a clean separation of the layer of basic specifications from the other layers. Any logic can be used in the basic specification layer, as long as it is formalized as an *institution* [10]. The semantics of the other layers is defined for an arbitrary institution. The architectural specification layer is also independent of the details of the features used for building structured specifications.

3 Simple Refinements

The simplest form of refinement is just model class inclusion. Consider the following standard specification of monoids:

spec Monoid =
  sort *Elem*
  ops $0 : Elem$;
    $\_\_ + \_\_ : Elem \times Elem \to Elem$, *assoc*, *unit 0*

This specification is rather loose. It can be refined in many different ways, e.g. into the natural numbers. We first specify the natural numbers inductively in terms of zero and successor, then define addition, and finally hide the successor operation:

spec NATWITHSUC =
  free type $Nat ::= 0 \mid suc(Nat)$
  op $\_\_ + \_\_ : Nat \times Nat \rightarrow Nat, unit\ 0$
    $\forall x, y : Num \bullet x + suc(y) = suc(x + y)$

spec NAT =
  NATWITHSUC **hide** *suc*

The refinement between the two specifications can now be stated as follows:

refinement R1 =
  MONOID **refined via** $Elem \mapsto Nat$ **to** NAT

Correctness of this refinement means that given any NAT-model, its reduct along $Elem \mapsto Nat$ yields a MONOID-model, formally

$$M|_{\sigma} \in [\![\text{MONOID}]\!] \text{ for each } M \in [\![\text{NAT}]\!]$$

where σ maps *Elem* to *Nat* (σ is generated from the symbol map $Elem \mapsto Nat$ in a straightforward way, see [8]). Of course, this just states that the natural numbers with addition form a monoid, or, in other words, that σ : MONOID $\rightarrow$ NAT is a specification morphism. Specification morphisms arise in CASL already as *views*, used for instantiating generic specifications. For that application it is useful to allow them to be generic themselves, and this leads to certain complications. Here specification morphisms are used for a different purpose where the complications of generic views are irrelevant and distracting.

The specification NAT can be taken as a realisation of the natural numbers, but quite an inefficient one. It is far more efficient to use a binary representation (++ is binary addition with carry):

spec NATBIN =
  generated type $Bin ::= 0 \mid 1 \mid \_\_0(Bin) \mid \_\_1(Bin)$
  ops $\_\_ + \_\_ , \_\_ ++ \_\_ : Bin \times Bin \rightarrow Bin$
  $\forall x, y : Bin$

$\bullet\ 0\ 0 = 0$	$\bullet\ 0\ 1 = 1$
$\bullet\ \neg\ (0 = 1)$	$\bullet\ x\ 0 = y\ 0 \Rightarrow x = y$
$\bullet\ \neg\ (x\ 0 = y\ 1)$	$\bullet\ x\ 1 = y\ 1 \Rightarrow x = y$
$\bullet\ 0 + 0 = 0$	$\bullet\ 0 ++ 0 = 1$
$\bullet\ x\ 0 + y\ 0 = (x + y)\ 0$	$\bullet\ x\ 0 ++ y\ 0 = (x + y)\ 1$
$\bullet\ x\ 0 + y\ 1 = (x + y)\ 1$	$\bullet\ x\ 0 ++ y\ 1 = (x ++ y)\ 0$
$\bullet\ x\ 1 + y\ 0 = (x + y)\ 1$	$\bullet\ x\ 1 ++ y\ 0 = (x ++ y)\ 0$
$\bullet\ x\ 1 + y\ 1 = (x ++ y)\ 0$	$\bullet\ x\ 1 ++ y\ 1 = (x ++ y)\ 1$

We now have a further refinement:

refinement R2 =
    NAT **refined via** *Nat* ↦ *Bin* **to** NATBIN

Note that it is quite typical that the target specification of the refinement adds auxiliary operations, which are forgotten by reducing along the signature morphism.

The two refinements can be combined into a *chain* of refinements:

refinement R3 =
    MONOID **refined via** *Elem* ↦ *Nat* **to**
        NAT **refined via** *Nat* ↦ *Bin* **to** NATBIN

which can be depicted as follows:

$$\textsc{Monoid} \overset{\sigma}{\rightsquigarrow} \textsc{Nat} \overset{\theta}{\rightsquigarrow} \textsc{NatBin}$$

Here, σ and θ are the specification morphisms associated to the refinements, and the correctness conditions for the individual refinements guarantee that the chain is also a correct refinement in the following sense:

$$M|_{\sigma;\theta} = (M|_\theta)|_\sigma \in [\![\textsc{Monoid}]\!] \text{ for each } M \in [\![\textsc{NatBin}]\!]$$

If we want to save some typing, we can also write:

refinement R3′ =
    MONOID **refined via** *Elem* ↦ *Nat* **to** R2

or equivalently

refinement R3″ =
    MONOID **refined via** *Elem* ↦ *Nat* **to** NAT **then** R2

which can be rewritten as

refinement R3‴ = R1 **then** R2

4 Branching Refinements

Suppose that we want to implement not only NAT, but NATWITHSUC, i.e. also the successor function. Now, while the presence of the successor function enables an easy *specification* of the natural numbers, it may be a little distracting in achieving an efficient *implementation.* So we can help the implementor and impose (via a CASL architectural specification) that the natural numbers should be implemented with addition, and the successor function should only be implemented afterwards, *in terms of* addition:

arch spec ADDITION_FIRST =
    units N: NAT;
        M: **{ op** $suc(n : Nat) : Nat = n + 1$ **} given** N
    result M

We thus have chosen to split the implementation of NatWithSuc into two independent subtasks: the implementation of Nat, and the implementation of a generic program, that given *any* Nat-model will realise the successor function on top of it. The generic program is then applied once to the implementation N of Nat. A version making this genericity explicit is the following:

> **arch spec** Addition_First_Generic =
>   **units** N : Nat;
> F : Nat $\rightarrow$ **{ op** $suc(n : Nat) : Nat = n + 1$ **}**;
> $M = F[N]$
>   **result** M

Here, F is a generic program unit, that is, a parametrized program, that may be applied to any program unit matching its parameter specification, not only to N. The specification

Nat $\rightarrow$ **{ op** $suc(n : Nat) : Nat = n + 1$ **}**

is a so-called (generic) *unit specification.* It denotes the class of all functions F mapping Nat-models to models of

Nat **then { op** $suc(n : Nat) : Nat = n + 1$ **}**

in such a way that the argument model (unit) is preserved, i.e. $F(N)|_{\text{Nat}} = N$ for any Nat-model N.

The term $F[N]$ is a *unit term* (in this case: a unit application) computing a unit out of the (generic and non-generic) units introduced earlier. $M = F[N]$ is a *unit definition* which defines the unit M to be exactly the (value of the) unit term $F[N]$. The unit M is then used as the *result* unit term. In general, the result unit may be given by an arbitrary unit term involving the units declared or defined within the architectural specification. If the result unit is itself to be generic, the unit term has to preceded by a λ-abstraction (this is one form of *unit expression*).

We can express that Addition_First is a refinement of NatWithSuc as follows:

> **refinement** R4 =
>   NatWithSuc **refined to arch spec** Addition_First

This time, we have left out the signature morphism, since it is just the identity. Since the refined specification is an architectural specification, we use the keywords **arch spec** before the refined specification.

If we want to combine this design decision with the decision to implement Nat with NatBin, we can write a refinement directly after the specification of the unit in question:

> **arch spec** Addition_First_With_Bin =
>   **units** N : Nat **refined via** $Nat \mapsto Bin$ **to** NatBin;
> F : Nat $\rightarrow$ **{ op** $suc(n : Nat) : Nat = n + 1$ **}**;
> $M = F[N]$
>   **result** M

or, more briefly, using the refinement of NAT into NATBIN named R2 above:

> **arch spec** ADDITION_FIRST_WITH_BIN$'$ =
>   **units** N : R2;
> F : NAT $\rightarrow$ **{ op** $suc(n : Nat) : Nat = n + 1$ **}**;
> $M = F[N]$
>   **result** M

5 Refinement: Constructor Implementations

Semantically, all the types of refinements introduced so far can be seen as *constructor implementations* in the sense of [15]. Constructor implementations are written as

$$SP \underset{\kappa}{\rightsquigarrow\!\!>} SP'$$

Here, a constructor κ is a function mapping models to models; formally $\kappa : \mathbf{Mod}(Sig[SP']) \rightarrow \mathbf{Mod}(Sig[SP])$. Such a constructor implementation is *correct* if

$$\text{for all } A' \in [\![SP']\!],\ \kappa(A') \in [\![SP]\!].$$

In our proposed refinement language, constructors are induced by specification morphisms $\sigma \colon SP \longrightarrow SP'$, that is, signature morphisms from $Sig[SP]$ to $Sig[SP']$ with $[\![SP']\!]|_\sigma \subseteq [\![SP]\!]$. The constructor is just the reduct functor induced by σ, and correctness is equivalent to σ being a specification morphism.

Constructors correspond to *generic program modules* in programming languages, such as generic packages in Ada or functors in Extended ML:

```
functor K(X:SP'):SP = ... code ...
```

In the framework of [15], a specification is implemented via a sequence of refinement steps, until ultimately the empty specification EMPTY is reached:

$$SP_0 \underset{\kappa_1}{\rightsquigarrow\!\!>} SP_1 \underset{\kappa_2}{\rightsquigarrow\!\!>} \cdots \underset{\kappa_n}{\rightsquigarrow\!\!>} SP_n = \textsc{Empty}$$

If all these steps are correct, the combination of the constructors (starting with the trivial model *empty* of the EMPTY specification) yields a model of the original specification SP_0:

$$\kappa_1(\kappa_2(\cdots(\kappa_n(\mathit{empty}))\cdots)) \in [\![SP_0]\!]$$

Architectural specifications introduce *branching*: a specification may be refined to *several* specifications, which requires n-ary constructors:

$$SP \underset{\kappa}{\rightsquigarrow\!\!>} \begin{cases} SP_1 \\ \vdots \\ SP_n \end{cases}$$

As expected, correctness here means that

$$\text{for all } A_1 \in [\![SP_1]\!], \ldots, A_n \in [\![SP_n]\!],\ \kappa(A_1, \ldots, A_n) \in [\![SP]\!].$$

The corresponding Casl architectural specification is written

arch spec
  units U_1: SP_1
    ...
    U_n: SP_n
  result UT

where UT is a unit term describing the constructor κ, which may involve the unit names $U_1, \ldots, U_n$. Unit terms are built by renaming of units, hiding parts thereof, amalgamation of units, application of generic units to arguments, as well as local unit definitions (introducing local names for unit terms). The requirements imposed by the semantics on the result unit terms ensure that the induced constructors are always defined for the relevant argument units.

Analogously to the linear situation, once we have a tree of correct refinement steps with leaves being empty specifications, as follows:

$$SP \underset{\kappa}{\rightsquigarrow} \begin{cases} SP_1 \underset{\kappa_1}{\rightsquigarrow} \textsc{Empty} \\ \vdots \\ SP_n \underset{\kappa_n}{\rightsquigarrow} \begin{cases} SP_{n1} \underset{\kappa_{n1}}{\rightsquigarrow} \left\{ SP_{n11} \underset{\kappa_{n11}}{\rightsquigarrow} \textsc{Empty} \right. \\ \ldots \\ SP_{nm} \underset{\kappa_{nm}}{\rightsquigarrow} \textsc{Empty} \end{cases} \end{cases}$$

we can construct a model of the original requirement specification by successively applying the constructors, starting with the trivial model *empty*:

$$\begin{array}{l} \kappa(\,\kappa_1(\mathit{empty}), \\ \quad \ldots \\ \quad \kappa_n(\,\kappa_{n1}(\kappa_{n11}(\mathit{empty})), \\ \qquad \ldots \\ \qquad \kappa_{nm}(\mathit{empty}))) \qquad \in [\![SP]\!] \end{array}$$

Note that this whole section applies not only to the refinement of ordinary program units (models), but also to generic units (functions from models to models). We will come back to this later.

6 Programs in Casl

One problem with the approach described so far is that the constructors provided by specification morphisms and architectural specifications in Casl do not suffice for implementing specifications. In a sense, these constructors only provide means to *combine* or *modify* existing program units—but there is no way to build program units *from scratch.* That is, Casl lacks a notion of *program.*

An obvious way out of this situation is to add more unit operations that can be used for unit terms (or unit expressions) in architectural specifications.

Concerning construction of datatypes, one could add a simple version of free extension, giving a model of a datatype that is determined uniquely up to isomorphism, and that corresponds to an algebraic datatype in a functional programming language. For the construction of operations on top of these datatypes, one could use reducts along derived signature morphisms. Derived signature morphisms may map an operation to a term or to a recursive definition by means of equations, like function definitions in a functional programming language. See [16, Chap. 4] for a more detailed account of this approach.

Note that this approach is necessarily no longer institution independent. The details of the kind of free extensions that actually correspond to datatypes in a programming language depend both on the institution and the programming language at hand. The same remark holds for the definition of derived signature morphisms.

An alternative is to approximate the institution independent essence of programs by considering *monomorphic specifications*. A unit specification is monomorphic if the result specification is a monomorphic extension of the argument specifications. This means that it is a construction that is unique up to isomorphism. Ultimately, monomorphic unit specifications need to be translated to (parametrized) programs in some programming language. As above, this process obviously depends on both the institution and the programming language in question. The difference is that the specification language itself remains institution independent, since the translation to a programming language is not part of the specification language.

In some cases it is possible to perform the translation automatically, for unit specifications that obey certain syntactic restrictions. For functional programming languages such as Haskell and ML, one would require that all sorts are given as free types, and all functions are defined by means of recursive equations in such a way that termination is provable. Indeed, the translation of a parametrized program then provides a construction that is unique, not only unique up to isomorphism. See [14] for details, and [6] for a translation of a subset of Casl to OCAML. Using free extensions, it is also possible to capture partial recursive functions, see [6, 14]. Moreover, with Haskell (and its type class Eq) as target language, generated types with explicitly given equality can also be used. For ML and Haskell, there is also a direct correspondence at the level of Casl unit terms, see Fig. 1.

For other programming languages, the translation between monomorphic specifications and programs might be much less straightforward. In general, it may be necessary to translate manually, and prove that the resulting program is a correct realization of the specification. There may also be a mismatch between the constructs that are available for combining modules in the programming language and the constructs that Casl provides for combining unit terms. Then, one possibility would be to view unit terms in architectural specifications as prescriptions for the composition and transformation of the component units, and carry these out manually using the constructs that the programming language provides. (This may be automated by devising operations on program

Casl	ML	Haskell
non-generic unit	structure	module
generic unit	functor	multi-parameter type class in a module
monomorphic unit specification with free types and recursive definitions	structure with datatypes and recursive definitions	module with datatypes and recursive definitions
unit application	functor application	type class instantiation
unit amalgamation	combination of structures	combination of modules
unit hiding	restriction to subsignature	hiding
unit renaming	redefinition	redefinition
architectural specification	structure/functor using other structures/functors	module using other modules

Fig. 1. Unit term constructs in ML and Haskell

texts corresponding to unit term constructs.) Alternatively, one might take the target programming language into account in the refinement process and simply avoid in unit terms any use of the constructs that have no counterpart in the programming language at hand.

With this approach, the use of a parametrized program κ in a constructor implementation $SP \underset{\kappa}{\rightsquigarrow} SP'$ is expressed as

> **arch spec**
>   **unit** K : $SP' \to SP$ **refined to** *USP*
>   **result** K

where *USP* is a monomorphic specification of κ from which the corresponding parameterized program may be obtained directly. Such a constructor can also be used in the context of another refinement. For example, the refinement

$$SP \underset{\kappa}{\rightsquigarrow} SP' \underset{\kappa'}{\rightsquigarrow} SP''$$

is expressed as

```
refinement R5 =
  SP refined to
    arch spec
      units
        K : SP' → SP refined to USP
        A' : SP' refined to arch spec
                              units
                                K' : SP'' → SP' refined to USP'
                                A'' : SP''
                              result K'(A'')
      result K(A')
```

where *USP* and *USP'* are monomorphic specifications of κ and κ', respectively.

7 Refinements in CASL

Let us now come to a more systematic treatment of the refinement language that we propose.

The grammar below extends the grammar for the concrete syntax of CASL given in the CASL Reference Manual [8]. The new parts of the grammar are marked in *italics*, while removed parts are ~~crossed out~~. The details here are formulated in terms of concrete syntax, in contrast to [8], to make the presentation more accessible to readers who are not intimately familiar with the details of the CASL design. A corresponding abstract syntax is given in the appendix.

The central notion of the refinement language is specification refinement. These can take various forms, all of which have already been discussed along with concrete examples above.

```
SPEC-REF    ::= SPEC-NAME
              | UNIT-SPEC
              | UNIT-SPEC refined via SYMB-MAP-ITEMS* to SPEC-REF
              | UNIT-SPEC refined to SPEC-REF
              | arch spec ARCH-SPEC
              | SPEC-REF then SPEC-REF
```

Like ordinary specifications and unit specifications, specification refinements can be named:

```
SPEC-REF-DEFN ::= refinement SPEC-NAME = SPEC-REF end
```

Here, the notation end stands for optional **end**.

The syntax of declarations of units within architectural specifications is relaxed: arbitrary specification refinements are allowed, not only unit specifications:

```
UNIT-DECL     ::= UNIT-NAME : SPEC-REF
```

To avoid additional complexity but mainly for methodological reasons, we leave out refinements of unit specifications with imports (the "**given**" clause in ADDITION_FIRST in Sect. 4). See Sect. 10 for justification and discussion.

Finally, since we allow for coercion of architectural specifications to specification refinements, there is no need for coercing them to unit specifications:

```
UNIT-SPEC   ::= SPEC
              | SPEC *...* SPEC -> SPEC
              ~~| arch spec ARCH-SPEC~~
              | closed UNIT-SPEC
```

As with the rest of CASL, the semantics is given in two parts: the static semantics and the model semantics. In the semantics below, we ignore global environments which store the meanings of global names; consequently, we also omit the case of named specification refinements. Details, which are straightforward, follow a similar pattern as in [8, III:5 and III:6].

The static semantics of specification refinements is given in Fig. 2. The judgements are of the form $\vdash SPR \rhd (U\Sigma, U\Sigma')$. Here, $U\Sigma$ is the unit signature for

$$\frac{\vdash \mathit{USP} \rhd U\Sigma}{\vdash \mathit{USP} \text{ qua } \mathtt{SPEC\text{-}REF} \rhd (U\Sigma, U\Sigma)} \qquad \frac{\begin{array}{c}\vdash \mathit{USP} \rhd U\Sigma = (\Sigma_1, \ldots, \Sigma_n \to \Sigma) \\ \vdash \mathit{SI} \rhd \sigma : \Sigma \to \Sigma' \\ \vdash \mathit{SPR} \rhd (U\Sigma' = (\Sigma_1, \ldots, \Sigma_n \to \Sigma'), U\Sigma'')\end{array}}{\vdash \mathit{USP} \textbf{ refined via } \mathit{SI} \textbf{ to } \mathit{SPR} \rhd (U\Sigma, U\Sigma'')}$$

$$\frac{\begin{array}{c}\vdash \mathit{USP} \rhd U\Sigma \\ \vdash \mathit{SPR} \rhd (U\Sigma, U\Sigma'')\end{array}}{\vdash \mathit{USP} \textbf{ refined to } \mathit{SPR} \rhd (U\Sigma, U\Sigma'')}$$

$$\frac{\vdash \mathit{ASP} \rhd C_s, U\Sigma}{\vdash \mathtt{arch\ spec}\ \mathit{ASP} \rhd (U\Sigma, \bot)} \qquad \frac{\vdash \mathit{SPR}_1 \rhd (U\Sigma, U\Sigma') \qquad \vdash \mathit{SPR}_2 \rhd (U\Sigma', U\Sigma'')}{\vdash \mathit{SPR}_1 \textbf{ then } \mathit{SPR}_2 \rhd (U\Sigma, U\Sigma'')}$$

Fig. 2. Static semantics of specification refinements

units of the specification being refined and $U\Sigma'$ is the unit signature for units of the specification after refinement. A unit signature consists of a tuple of argument signatures (which is empty and may be omitted for non-generic units) and a result signature. Further details can be found in [8, III:5]. For instance, we have

$$\vdash \text{Monoid } \textbf{refined via } \mathit{Elem} \mapsto \mathit{Nat} \textbf{ to } \text{Nat} \rhd (\Sigma_{\text{Monoid}}, \Sigma_{\text{Nat}}),$$

where Σ_{Monoid} and Σ_{Nat} are the signatures of Monoid and Nat respectively. Since so far we don't allow for further refinement of architectural specifications, only of their units, if *SPR* is an architectural specification we mark this by putting $U\Sigma' = \bot$.

The model semantics of specification refinements is given in Fig. 3. The judgements are of the form $\vdash \mathit{SPR} \Rightarrow \mathcal{R}$. If $\vdash \mathit{SPR} \rhd (U\Sigma, U\Sigma')$ then $\mathcal{R}$ is a class of pairs (U, U') such that U and U' are units over unit signatures $U\Sigma$ and $U\Sigma'$ respectively and $\mathcal{R}^{-1}$ is a partial function mapping $U\Sigma'$-units to $U\Sigma$-units. $\mathcal{R}^{-1}$ is the constructor involved in the refinement and its domain is the class of models of the specification after refinement. A unit is either a model (when it is non-generic) or a unit function, mapping compatible tuples of argument models to result models. Further details can be found in [8, III:5]. For instance, we have

$$\vdash \text{Monoid } \textbf{refined via } \mathit{Elem} \mapsto \mathit{Nat} \textbf{ to } \text{Nat} \Rightarrow \{(N|_\sigma, N) \mid N \in [\![\text{Nat}]\!]\}$$

where σ maps *Elem* to *Nat*. Again, this takes a special form when *SPR* is an architectural specification: the second component of each pair is then $\bot$.

Both static semantics and model semantics rules rely on the semantics of unit specifications [8, III:5], symbol mappings [8, III:4] and architectural specifications [8, III:5]. We just recall that the static semantics of an architectural specification consists of a static unit context (which we ignore here, but see Sect. 8) and a result unit signature. An architectural model consists of a unit environment (again, ignored here, but see Sect. 8) and a result unit. Note that the semantics of architectural specifications has to be adjusted as well, since its unit declarations may now involve arbitrary specification refinements rather than only unit specifications. Luckily, going from the semantics of the former to the plain Casl semantics of the latter is very easy here. In the static semantics,

$$\frac{\vdash \mathit{USP} \Rightarrow \mathcal{U}}{\vdash \mathit{USP}\ \text{qua}\ \texttt{SPEC-REF} \Rightarrow \{(U,U) \mid U \in \mathcal{U}\}}$$

$$\frac{\begin{array}{c}\vdash \mathit{USP} \Rightarrow \mathcal{U} \qquad \vdash \mathit{SPR} \Rightarrow \mathcal{R}\\ U' \in \mathcal{U},\ \text{for all } (U',U'') \in \mathcal{R}\end{array}}{\vdash \mathit{USP}\ \textbf{refined to}\ \mathit{SPR} \Rightarrow \mathcal{R}}$$

$$\frac{\begin{array}{c}\vdash \mathit{USP} \Rightarrow \mathcal{U} \qquad \vdash \mathit{SI} \rhd \sigma : \Sigma \to \Sigma' \qquad \vdash \mathit{SPR} \Rightarrow \mathcal{R}\\ U'|_\sigma \in \mathcal{U},\ \text{for all } (U',U'') \in \mathcal{R}\\ \mathcal{R}' = \{(U'|_\sigma, U'') \mid (U',U'') \in \mathcal{R}\}\end{array}}{\vdash \mathit{USP}\ \textbf{refined via}\ \mathit{SI}\ \textbf{to}\ \mathit{SPR} \Rightarrow \mathcal{R}'}$$

$$\frac{\vdash \mathit{ASP} \Rightarrow \mathcal{AM}}{\vdash \texttt{arch spec}\ \mathit{ASP} \Rightarrow \{(U,\bot) \mid (E,U) \in \mathcal{AM}\}}$$

$$\frac{\begin{array}{c}\vdash \mathit{SPR}_1 \Rightarrow \mathcal{R}_1 \qquad \vdash \mathit{SPR}_2 \Rightarrow \mathcal{R}_2\\ \text{for all } (U',U'') \in \mathcal{R}_2, (U,U') \in \mathcal{R}_1 \text{ for some } U\\ \mathcal{R} = \{(U,U'') \mid (U,U') \in \mathcal{R}_1, (U',U'') \in \mathcal{R}_2 \text{ for some } U'\}\end{array}}{\vdash \mathit{SPR}_1\ \textbf{then}\ \mathit{SPR}_2 \Rightarrow \mathcal{R}}$$

Fig. 3. Model semantics of specification refinements

from the semantics of specification refinements we just take the first component (the unit signature of the specification being refined). In the model semantics, we project the semantics of specification refinements onto the first component, thus taking the class of all units that may arise as results of the construction involved in the refinement.

The semantics of specification refinements relies on the simplifying assumption that the parameter specifications of generic unit specifications do not change under refinement. This allows us to freely use the reduct notation $U|_\sigma$, even when U is a generic unit; in this case, the notation denotes the unit function obtained by reducing the result via σ after applying U. In practice, this restriction is not troublesome, since we always can write an architectural specification that adjusts the parameter specification as required. Namely, given unit specifications $SP \to SP'$ and $SP_1 \to SP'_1$ with a specification morphism $\sigma : SP_1 \to SP$, the following is a correct specification refinement[4]

$SP \to SP'$ **refined via** τ **to arch spec**
  unit $F{:}SP_1 \to SP'_1$
  result $\lambda X{:}SP.F[X$ **fit** $\sigma]$

[4] Assuming that all symbols shared between SP'_1 and SP originate in SP_1, as imposed by CASL rules for application of generic units.

where τ is a specification morphism from SP_1 to the pushout specification $SP'_1 \oplus SP$ in the following diagram:

$$\begin{tikzcd}
SP_1 \arrow[r] \arrow[d, "\sigma"] & SP'_1 \arrow[d, dotted] \\
SP \arrow[r, dotted] \arrow[dr] & SP'_1 \oplus SP \\
& SP' \arrow[u, "\tau"']
\end{tikzcd}$$

In the terminology of [11], (σ, τ) is a *first-order morphism* from $SP \to SP'$ to $SP_1 \to SP'_1$.

A crucial property of development trees as are now captured by architectural specifications with specification refinements is that adding correct refinements to unit specifications in an architectural specification, and thus expanding the development tree by additional refinement steps at its leaves, preserves correctness of the entire development. In particular, the semantics of architectural specifications with new correct refinements remains well-defined. This holds by [3, Thm. 2] (a technical assumption necessary there holds trivially in the absence of imports).

8 Component refinements

Refinements introduced in Sect. 7 do not allow the user to refine architectural specifications as such. Only refinements for individual units are allowed, and they must be inserted into the architectural specification, directly into unit declarations. Consider for instance the following example from Sect. 4, where a refinement for unit N in the architectural specification ADDITION_FIRST_GENERIC was captured as follows:

arch spec ADDITION_FIRST_WITH_BIN′ =
  units N : R2;
    F : NAT $\to$ **{ op** $suc(n : Nat) : Nat = n + 1$ **}**;
    $M = F[N]$
  result M

This is not very convenient: given an already defined architectural specification (in this case, ADDITION_FIRST_GENERIC), one would like to avoid rewriting it when indicating that specifications of some of the units (N here) are to be refined (using R2 here). Instead, one would rather refer to the architectural specification as given, and indicate refinements that are to follow, in this case writing:

refinement R =
  arch spec ADDITION_FIRST_GENERIC **then** {N **to** R2}

where {N **to** $R2$} is a refinement of an architectural specification having a unit named N. The refinement R then consists of the architectural specification Addition_First_Generic with N further refined according to R2. The need for such syntax is perhaps even more visible in complex examples involving nested refinements, like refinement R5 at the end of Sect. 6, which one would prefer to restructure as follows:

```
refinement R5′ =
    SP refined to arch spec units
                                   K : SP′ → SP
                                   A′ : SP′
                                 result K(A′)
    then {K to USP,
          A′ to arch spec units
                                   K′ : SP″ → SP′
                                   A″ : SP″
                                 result K′(A″)
                then {K′ to USP′}}
```

or even:

```
refinement R5″ =
    SP refined to arch spec units
                                   K : SP′ → SP
                                   A′ : SP′
                                 result K(A′)
    then {K to USP,
          A′ to arch spec units
                                   K′ : SP″ → SP′
                                   A″ : SP″
                                 result K′(A″)}
    then {A′ to {K′ to USP′}}
```

Of course, all the architectural specifications used here, as well as the refinements, would typically be defined earlier and then referred to by their names when refined further.

To capture such possibilities we extend the syntax for refinements introduced in Sect. 7, adding a new form:

```
SPEC-REF ::= ...
           | {UNIT-NAME_1 to SPEC-REF_1, ..., UNIT-NAME_n to SPEC-REF_n}
```

However, this apparently simple change considerably increases the conceptual (and then semantic) complexity here, since in fact we are now dealing with three kinds of refinements:

- *unit specification refinements* which lead from a unit specification to another unit specification;
- *branching specification refinements* which generalise unit refinements by additionally allowing the target specification to be architectural; and

$$\frac{\vdash \mathit{USP} \rhd U\Sigma}{\vdash \mathit{USP}\ \text{qua}\ \mathtt{SPEC\text{-}REF} \rhd (U\Sigma, U\Sigma)} \qquad \frac{\begin{array}{c}\vdash \mathit{USP} \rhd U\Sigma = (\Sigma_1, \ldots, \Sigma_n \to \Sigma) \\ \vdash \mathit{SI} \rhd \sigma : \Sigma \to \Sigma' \\ \vdash \mathit{SPR} \rhd ((\Sigma_1, \ldots, \Sigma_n \to \Sigma'), B\Sigma'')\end{array}}{\vdash \mathit{USP}\ \textbf{refined via}\ \mathit{SI}\ \textbf{to}\ \mathit{SPR} \rhd (U\Sigma, B\Sigma'')}$$

$$\frac{\begin{array}{c}\vdash \mathit{USP} \rhd U\Sigma \\ \vdash \mathit{SPR} \rhd (U\Sigma, B\Sigma'')\end{array}}{\vdash \mathit{USP}\ \textbf{refined to}\ \mathit{SPR} \rhd (U\Sigma, B\Sigma'')}$$

$$\frac{\vdash \mathit{ASP} \rhd \mathit{RstC}, U\Sigma}{\vdash \mathtt{arch\ spec}\ \mathit{ASP} \rhd (U\Sigma, \pi_2(\mathit{RstC}))} \qquad \frac{\begin{array}{c}\vdash \mathit{SPR}_1 \rhd R\Sigma_1 \qquad \vdash \mathit{SPR}_2 \rhd R\Sigma_2 \\ R\Sigma = R\Sigma_1 \,;R\Sigma_2\end{array}}{\vdash \mathit{SPR}_1\ \textbf{then}\ \mathit{SPR}_2 \rhd R\Sigma}$$

$$\frac{\begin{array}{c}\mathit{UN}_1, \ldots, \mathit{UN}_n \text{ are distinct} \\ \vdash \mathit{SPR}_1 \rhd R\Sigma_1 \quad \cdots \quad \vdash \mathit{SPR}_n \rhd R\Sigma_n\end{array}}{\vdash \{\mathit{UN}_1\ \mathtt{to}\ \mathit{SPR}_1\mathtt{,} \ldots \mathtt{,}\ \mathit{UN}_n\ \mathtt{to}\ \mathit{SPR}_n\} \rhd \{\mathit{UN}_1 \mapsto R\Sigma_1, \ldots, \mathit{UN}_n \mapsto R\Sigma_n\}}$$

Fig. 4. Static semantics of extended refinements

– *component specification refinements* which name units whose specifications are to be further refined as indicated.

The corresponding semantic concepts come now in three flavours as well. For the static semantics, we introduce *refinement signatures*, $R\Sigma$, which take one of the following forms:

– *unit refinement signatures* $(U\Sigma, U\Sigma')$ which consist of two unit signatures (this corresponds to the typical case in the static semantics of Sect. 7);
– *branching refinement signatures* $(U\Sigma, B\Sigma')$ which consist of a unit signature $U\Sigma$ and a *branching signature* $B\Sigma'$, which is either a unit signature $U\Sigma'$ (in which case the branching refinement signature is a unit refinement signature) or a *branching static context* BstC', which is in turn a (finite) map assigning branching signatures to unit names. Note that therefore all static contexts as used in the plain Casl semantics [8, III:5] are branching static contexts, but not vice versa;
– *component refinement signatures* which are (finite) maps $\{\mathit{UN}_i \mapsto R\Sigma_i\}_{i \in \mathcal{J}}$ from unit names to refinement signatures. When all $R\Sigma_i$, $i \in \mathcal{J}$, in such a map are branching refinement signatures, we refer to it as a *refined-unit static context.* Any refined-unit static context $\mathit{RstC} = \{\mathit{UN}_i \mapsto (U\Sigma_i, B\Sigma_i)\}_{i \in \mathcal{J}}$ can be naturally coerced to the static context $\pi_1(\mathit{RstC}) = \{\mathit{UN}_i \mapsto U\Sigma_i\}_{i \in \mathcal{J}}$ of the plain Casl semantics, as well as to the branching static context $\pi_2(\mathit{RstC}) = \{\mathit{UN}_i \mapsto B\Sigma_i\}_{i \in \mathcal{J}}$.

New rules for the static semantics of refinements are given in Fig. 4. The first three rules are essentially inherited from Sect. 7, with a minor change to allow for the target signature to be branching. The new rule for architectural refinements allows for their further refinement by replacing the $\perp$ mark by the branching static context that emerges from the semantics of the architectural specification (see the end of this section for a brief discussion of the new semantics for architectural specifications). The rule for individual component refinements is straightforward: it just stores the refinement signatures obtained from the refinements attached to unit names. The extra complexity is hidden in the rule for refinement composition using an auxiliary partial composition operation on refinement signatures. Given refinement signatures $R\Sigma_1$ and $R\Sigma_2$, their *composition* $R\Sigma_1 ; R\Sigma_2$ is defined inductively depending on the form of the first argument:

- $R\Sigma_1 = (U\Sigma, U\Sigma')$: then $R\Sigma_1 ; R\Sigma_2$ is defined only if $R\Sigma_2$ is a branching refinement signature of the form $(U\Sigma', B\Sigma'')$. Then $R\Sigma_1 ; R\Sigma_2 = (U\Sigma, B\Sigma'')$.
- $R\Sigma_1 = (U\Sigma, BstC')$: then $R\Sigma_1 ; R\Sigma_2$ is defined only if $R\Sigma_2$ is a component refinement signature such that $R\Sigma_2$ *matches* $BstC'$, that is, $dom(R\Sigma_2) \subseteq dom(BstC')$ and for each $UN \in dom(R\Sigma_2)$,
  - either $BstC'(UN)$ is a unit signature and then $R\Sigma_2(UN) = (U\Sigma', B\Sigma'')$ with $U\Sigma' = BstC'(UN)$, or
  - $BstC'(UN)$ is a branching static context and then $R\Sigma_2(UN)$ matches $BstC'(UN)$.

  Then $R\Sigma_1 ; R\Sigma_2 = (U\Sigma, BstC'[R\Sigma_2])$, where given any branching static context $BstC'$ and component refinement signature $R\Sigma_2$ that matches $BstC'$, $BstC'[R\Sigma_2]$ modifies $BstC'$ on each $UN \in dom(R\Sigma_2)$ as follows:
  - if $BstC'(UN)$ is a unit signature then $BstC'[R\Sigma_2](UN) = B\Sigma''$ where $R\Sigma_2(UN) = (BstC'(UN), B\Sigma'')$,
  - if $BstC'(UN)$ is a branching static context then $BstC'[R\Sigma_2](UN) = BstC'(UN)[R\Sigma_2(UN)]$.
- $R\Sigma_1$ is a component refinement signature: then $R\Sigma_1 ; R\Sigma_2$ is defined only if $R\Sigma_2$ is a component refinement signature too, and moreover, for all $UN \in dom(R\Sigma_1) \cap dom(R\Sigma_2)$, $R\Sigma_{UN} = R\Sigma_1(UN) ; R\Sigma_2(UN)$ is defined. Then $R\Sigma_1 ; R\Sigma_2$ modifies the (ill-defined) union of $R\Sigma_1$ and $R\Sigma_2$ by putting $(R\Sigma_1 ; R\Sigma_2)(UN) = R\Sigma_{UN}$ for $UN \in dom(R\Sigma_1) \cap dom(R\Sigma_2)$.

The complexity of the model semantics for refinements increases similarly. Given a refinement signature $R\Sigma$, *refinement relations*, $\mathcal{R}$, are classes of *assignments*, R, which take the following forms:

- *unit assignments*, for $R\Sigma = (U\Sigma, U\Sigma')$, are pairs (U, U') of units over unit signatures $U\Sigma$ and $U\Sigma'$, respectively;
- *branching assignments*, for $R\Sigma = (U\Sigma, B\Sigma')$, are pairs (U, BM'), where U is a unit over the unit signature $U\Sigma$ and BM' is a *branching model* over the branching signature $B\Sigma'$, which is either a unit over $B\Sigma'$ when $B\Sigma'$ is a unit signature (in which case the branching assignment is a unit assignment), or a *branching environment* BE' that fits $B\Sigma'$ when $B\Sigma'$ is a branching static context. Branching environments are (finite) maps assigning branching models to unit names, with the obvious requirements to ensure compatibility with the branching signatures indicated in the corresponding branching static context. Note that therefore all unit environments as used in the plain Casl semantics [8, III:5] are branching environments, but not vice versa.

$$\frac{\vdash USP \Rightarrow \mathcal{U}}{\vdash USP \text{ qua } \texttt{SPEC-REF} \Rightarrow \{(U,U) \mid U \in \mathcal{U}\}}$$

$$\frac{\begin{array}{c}\vdash USP \Rightarrow \mathcal{U} \qquad \vdash SPR \Rightarrow \mathcal{R} \\ U' \in \mathcal{U}, \text{ for all } (U', BM'') \in \mathcal{R}\end{array}}{\vdash USP \textbf{ refined to } SPR \Rightarrow \mathcal{R}}$$

$$\frac{\begin{array}{c}\vdash USP \Rightarrow \mathcal{U} \qquad \vdash SI \rhd \sigma : \Sigma \to \Sigma' \qquad \vdash SPR \Rightarrow \mathcal{R} \\ U'|_\sigma \in \mathcal{U}, \text{ for all } (U', BM'') \in \mathcal{R} \\ \mathcal{R}' = \{(U'|_\sigma, BM'') \mid (U', BM'') \in \mathcal{R}\}\end{array}}{\vdash USP \textbf{ refined via } SI \textbf{ to } SPR \Rightarrow \mathcal{R}'}$$

$$\frac{\vdash ASP \Rightarrow \mathcal{AM}}{\vdash \texttt{arch spec } ASP \Rightarrow \{(U, \pi_2(RE)) \mid (RE, U) \in \mathcal{AM}\}}$$

$$\frac{\vdash SPR_1 \Rightarrow \mathcal{R}_1 \quad \cdots \quad \vdash SPR_n \Rightarrow \mathcal{R}_n}{\begin{array}{r}\vdash \{UN_1 \texttt{ to } SPR_1 \texttt{,} \ldots \texttt{,} UN_n \texttt{ to } SPR_n\} \Rightarrow \{R \mid dom(R) = \{UN_1, \ldots, UN_n\}, \\ R(UN_1) \in \mathcal{R}_1, \ldots, R(UN_n) \in \mathcal{R}_n\}\end{array}}$$

$$\frac{\vdash SPR_1 \Rightarrow \mathcal{R}_1 \qquad \vdash SPR_2 \Rightarrow \mathcal{R}_2 \qquad \mathcal{R} = \mathcal{R}_1 ; \mathcal{R}_2}{\vdash SPR_1 \textbf{ then } SPR_2 \Rightarrow \mathcal{R}}$$

Fig. 5. Model semantics of extended refinements

– *component assignments*, for $R\Sigma = \{UN_i \mapsto R\Sigma_i\}_{i \in \mathcal{J}}$, are (finite) maps $\{UN_i \mapsto R_i\}_{i \in \mathcal{J}}$ from unit names to assignments over the respective refinement signatures. When $R\Sigma$ is a refined-unit static context (and so each R_i, $i \in \mathcal{J}$, is a branching assignment) we refer to $RE = \{UN_i \mapsto (U_i, BM_i)\}_{i \in \mathcal{J}}$ as a *refined-unit environment.* Any such refined-unit environment can be naturally coerced to a unit environment $\pi_1(RE) = \{UN_i \mapsto U_i\}_{i \in \mathcal{J}}$ of the plain CASL semantics, as well as to a branching environment $\pi_2(RE) = \{UN_i \mapsto BM_i\}_{i \in \mathcal{J}}$.

New rules for the model semantics of refinements are given in Fig. 5. As with the static semantics, the non-trivial change is hidden in the rule for refinement composition using the auxiliary partial operation to compose refinement relations. Given two refinement relations $\mathcal{R}_1, \mathcal{R}_2$ over refinement signatures $R\Sigma_1, R\Sigma_2$, respectively, such that the composition $R\Sigma = R\Sigma_1 ; R\Sigma_2$ is defined, the *composition* $\mathcal{R}_1 ; \mathcal{R}_2$ is defined as a refinement relation over $R\Sigma$ as follows:

– $R\Sigma_1 = (U\Sigma, U\Sigma')$, $R\Sigma_2 = (U\Sigma', B\Sigma'')$: then $\mathcal{R}_1 ; \mathcal{R}_2$ is defined only if for all $(U', BM'') \in \mathcal{R}_2$ we have $(U, U') \in \mathcal{R}_1$ for some U. Then

$$\mathcal{R}_1 ; \mathcal{R}_2 = \{(U, BM'') \mid (U, U') \in \mathcal{R}_1, (U', BM'') \in \mathcal{R}_2 \text{ for some } U'\}$$

– $R\Sigma_1 = (U\Sigma, BstC')$ and $R\Sigma_2$ is a component refinement signature that matches $BstC'$: then $\mathcal{R}_1 \,;\mathcal{R}_2$ is defined only if for each $R_2 \in \mathcal{R}_2$ there is $(U, BE') \in \mathcal{R}_1$ such that R_2 *matches* BE', that is, for each $UN \in dom(R_2)$,
  - either $BstC'(UN)$ is a unit signature and then $R_2(UN) = (U'', BM'')$ with $U'' = BE'(UN)$, or
  - $BstC'(UN)$ is a branching static context and then $R_2(UN)$ matches $BE'(UN)$.

  Then

$$\mathcal{R}_1 \,;\mathcal{R}_2 = \{(U, BE'[R_2]) \mid (U, BE') \in \mathcal{R}_1, R_2 \in \mathcal{R}_2, R_2 \text{ matches } BE'\}$$

  where given any branching environment BE' that fits $BstC'$ and assignment R_2 that matches BE', $BE'[R_2]$ modifies BE' on each $UN \in dom(R_2)$ as follows:
  - if $BstC'(UN)$ is a unit signature then $BE'[R_2](UN) = BM''$ where $R_2(UN) = (BE'(UN), BM'')$;
  - if $BstC'(UN)$ is a branching static context then we put $BE'[R_2](UN) = BE'(UN)[R_2(UN)]$.

– $R\Sigma_1$ and $R\Sigma_2$ are component refinement signatures such that for all $UN \in dom(R\Sigma_1) \cap dom(R\Sigma_2)$, $R\Sigma_{UN} = R\Sigma_1(UN)\,; R\Sigma_2(UN)$ is defined then $\mathcal{R}_1 \,;\mathcal{R}_2$ is defined only if for each $R_2 \in \mathcal{R}_2$ there is $R_1 \in \mathcal{R}_1$ such that R_1 *transfers to* R_2, that is, for each $UN \in dom(R_1) \cap dom(R_2)$,
  - either $R\Sigma_1(UN)$ is a unit refinement signature $(U\Sigma, U\Sigma')$, and then $R_1(UN) = (U, U'_1)$ and $R_2(UN) = (U'_2, BM'')$ with $U'_1 = U'_2$, or
  - $R\Sigma_1(UN)$ is a branching refinement signature $(U\Sigma, BstC')$, and then $R_1(UN) = (U, BE')$ and $R_2(UN)$ is an assignment that matches BE', or
  - $R\Sigma_1(UN)$ is component refinement signature, and then $R_1(UN)$ transfers to $R_2(UN)$.

  Then

$$\mathcal{R}_1 \,;\mathcal{R}_2 = \{R_1 \,; R_2 \mid R_1 \in \mathcal{R}_1, R_2 \in \mathcal{R}_2, R_1 \text{ transfers to } R_2\}$$

  where given any assignments R_1, R_2 over $R\Sigma_1$, $R\Sigma_2$, respectively, such that R_1 transfers to R_2, $R_1 \,; R_2$ is the assignment that modifies the (ill-defined) union of R_1 and R_2 on each $UN \in dom(R_1) \cap dom(R_2)$ as follows:
  - if $R\Sigma_1(UN) = (U\Sigma, U\Sigma')$, $R_1(UN) = (U, U'_1)$ and $R_2(UN) = (U'_2, BM'')$ (hence $U'_1 = U'_2$) then $(R_1 \,; R_2)(UN) = (U, BM'')$;
  - if $R\Sigma_1(UN) = (U\Sigma, BstC')$, $R_1(UN) = (U, BE')$ (hence $R_2(UN)$ is an assignment that matches BE') then $(R_1 \,; R_2)(UN) = (U, BE'[R_2(UN)])$;
  - if $R\Sigma_1(UN)$ is a component refinement signature (hence $R_1(UN)$ and $R_2(UN)$ are component assignments such that $R_1(UN)$ transfers to $R_2(UN)$) then $(R_1 \,; R_2)(UN) = R_1(UN)\,; R_2(UN)$.

We also have to consider the necessary changes to the semantics of architectural specifications in [8, III:5]. Most visibly, as sketched above, we have to modify the semantic concepts for architectural specifications to work with refined-unit static contexts and refined-unit environments rather than unit static contexts and unit environments. This would alter most of the rules only formally. At crucial places, where units are used, the easy modification relies on the π_1 coercions from refined-unit static contexts to static contexts and from refined-unit environments to unit environments for static and model semantics, respectively.

Further straightforward modification concerns the semantics of unit declarations, now with arbitrary specification refinements in place of unit specifications. The new static semantics imposes the restriction that only branching specification refinements (so: no component specification refinements) are allowed here[5], and stores the appropriate branching refinement signature for the declared unit name in the refined-unit static context. Then, the model semantics produces the context that consists of all refined-unit environments that map the declared unit name to a branching assignment in the semantics of the branching specification refinement used in the declaration.

Finally, the semantics of unit definitions involves additional unit refinement signatures and assignments with the $\perp$ mark as the second component to indicate that unit definitions cannot be further refined.

9 The Steam Boiler Example

So far, we have illustrated the refinement language by means of toy examples. A discussion of realistic examples would exceed the space limitations of this paper. However, the CASL User Manual [2, Chap. 13] contains a specification of an industrial case study, namely a steam boiler control system that serves to control the water level in a steam boiler. Reference [2, Sect. 13.10] contains several architectural specifications explaining how to decompose the steam boiler control system into subsystems, using e.g. a specification VALUE for physical values, a specification SBCS_STATE for the specification of the state of the steam boiler, a specification PU_PREDICTION for prediction of the pump behaviour, etc. There is no space here to repeat the details of this example, so we refer the reader to [2, Chap. 13] and only use the additional linguistic features introduced in Sects. 7 and 8 to present specification refinements that formally capture the development described there.

The development in [2, Sect. 13.10] begins by indicating the initial architectural design for the overall requirement specification of the system:

arch spec ARCH_SBCS =
  units P: VALUE $\rightarrow$ PRELIMINARY
    S: PRELIMINARY $\rightarrow$ SBCS_STATE
    A: SBCS_STATE $\rightarrow$ SBCS_ANALYSIS
    C: SBCS_ANALYSIS $\rightarrow$ STEAM_BOILER_CONTROL_SYSTEM
  result $\lambda\, V$: VALUE $\bullet$ $C\,[A\,[S\,[P\,[V]]]]$

[5] The (abstract) syntax of specification refinements may be massaged so that some of the restrictions imposed by the static semantics on the composability and use of specification refinements are incorporated in the (context-free) grammar.

We may record this initial refinement now:

refinement Ref_Sbcs =
  Steam_Boiler_Control_System **to** Arch_Sbcs

In [2, Sect. 13.10], specifications refining the individual units of the above architectural specification are given. We extend the refinement Ref_Sbcs to capture them as well:

refinement Ref_Sbcs′ =
  Ref_Sbcs **then** {*P* **to arch spec** Arch_Preliminary,
                S **to** Unit_Sbcs_State,
                A **to arch spec** Arch_Analysis,
                C **to** Unit_Sbcs_System }

The resulting specification for the unit *S*, Unit_Sbcs_State, is monomorphic:

unit spec Unit_Sbcs_State =
  Preliminary → Sbcs_State_Impl

Development within Casl stops at this point, the last step being the passage to a program in a programming language. This also holds for the component *C*, even though the corresponding unit specification Unit_Sbcs_System is not explicitly provided in [2, Chap. 13].

The architectural specification Arch_Analysis used in the refinement above is given in [2, Sect. 13.10] as follows:

arch spec Arch_Analysis =
  units *FD* : Sbcs_State → Failure_Detection
     PR : Failure_Detection → PU_Prediction
     ME : PU_Prediction → Mode_Evolution [PU_Prediction]
     MTS: Mode_Evolution [PU_Prediction] → Sbcs_Analysis
  result λS : Sbcs_State • *MTS* [*ME* [*PR* [*FD* [*S*]]]]

As remarked in [2, Sect. 13.10], the specifications for the components *ME* and *MTS* are simple enough to be directly implemented, so we stop their development at this point. For the other two units, we record the corresponding refinements from [2, Sect. 13.10]:

refinement Ref_Arch_Analysis =
    {*FD* **to arch spec** Arch_Failure_Detection,
     PR **to arch spec** Arch_Prediction}

Finally, we put the above together and capture the overall development sketched in [2, Sect. 13.10]:

refinement Ref_Sbcs″ =
  Ref_Sbcs′ **then** {A **to** Ref_Arch_Analysis}

10 Conclusion and Future Work

The issue of refinement has been discussed in many specification frameworks, starting with [12] and [13], and some frameworks provide methods for proving correctness of refinements. But this is normally regarded as a "meta-level" issue and specification languages have typically not included syntactic constructs for formally stating such relationships between specifications that are analogous to those presented here for CASL. A notable exception is Specware [17], where specifications (and implementations) are structured using specification diagrams, and refinements correspond to specification morphisms for which syntax is provided. This, together with features for expanding specification diagrams, provides sufficient expressive power to capture our branching specification refinements. A difference is that Specware does not include a distinction between structured specifications and CASL-like architectural specifications, and refinements are required to preserve specification structure.

One point of this proposal that requires further work is the treatment of *shared subcomponents*, such as S in the following:

```
arch spec ASP = units S : USP
                      A_1 : arch spec
                              units
                                B_1 : USP'_1
                                ...
                                B_m : USP'_m
                              result ... B_1 ... S ... B_m ...
                      ...
                      A_2 : arch spec
                              units
                                C_1 : USP'_1
                                ...
                                C_p : USP'_p
                              result ... C_1 ... S ... C_p ...
                result ... A_1 ... A_2 ...
```

This requires a relatively straightforward modification to the semantics of CASL architectural specifications to make declared units visible within architectural specifications for further units.

We have not provided a treatment of refinements of unit specifications with imports, as was pointed out in Sect. 7. A formal account of imports would add considerably to the complexity of the semantics, see [8, III:5]. However, they can be regarded as implicit formal parameters which are instantiated only once, as in the specification ADDITION_FIRST_GENERIC. And moreover, this seems to be the appropriate view when refinements are considered. The ultimate target of refinement of such a specification will necessarily involve a parametrized program, and at some point in the refinement process this needs to be made explicit. Thus we regard the lack of treatment of imports as methodologically sound rather than merely a convenient simplification. That said, given modified visibility rules as sketched above, we could allow for refinements from a specification of the form SP **given** UT to an architectural specification of the form

> **arch spec**
>   **units** F: $SP_{par} \to SP'$
>   **result** $F[UT]$

which would be correct provided $UT : SP_{par}$ and $[\![SP]\!] \supseteq [\![SP_{par}$ **and** $SP']\!]$. Notice that here UT typically refers to units from the level of the unit that is specified by SP **given** UT; this is the reason why the modified visibility rules are necessary.

Finally, we have not discussed *behavioural refinement*, corresponding to *abstractor implementations* in [15]. Often, a refined specification does not satisfy the initial requirements literally, but only up to some sort of behavioural equivalence: for example, if stacks are implemented as arrays-with-pointer, then two arrays-with-pointer differing only in their "junk" entries (that is, those that are "above" the pointer) exhibit the same behaviour in terms of the stack operations, and hence correspond to the same abstract stack. This can be taken into account by re-interpreting unit specifications to include models that are behaviourally equivalent to literal models, see [4, 5]; then specification refinements as considered here become behavioural.

Acknowledgments: Our thanks to the anonymous referees, and to Michel Bidoit, whose suggestions encouraged us to make some important improvements.

References

1. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner. *Algebraic Foundations of Systems Specification.* Springer, 1999.
2. M. Bidoit and P.D. Mosses. CASL *User Manual.* LNCS Vol. 2900 (IFIP Series). Springer, 2004.
3. M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273, 2002.
4. M. Bidoit, D. Sannella, and A. Tarlecki. Global development via local observational construction steps. In *Proc. 27th Intl. Symp. on Mathematical Foundations of Computer Science*, LNCS Vol. 2420, pages 1–24. Springer, 2002.
5. M. Bidoit, D. Sannella, and A. Tarlecki. Observational interpretation for CASL specifications. In preparation, 2004.
6. T. Brunet. Génération automatique de code à partir de spécifications formelles. Master's thesis, Université de Poitiers, 2003.
7. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.cofi.info/`.
8. CoFI (The Common Framework Initiative). CASL *Reference Manual.* LNCS Vol. 2960 (IFIP Series). Springer, 2004.
9. J. Fitzgerald and C. Jones. Modularizing the formal description of a database system. In *Proc. VDM'90 Conference*, LNCS Vol. 428. Springer, 1990.

10. J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992.
11. J. Goguen and K. Lin. Morphisms and semantics for higher order parameterized programming. Available from `http://www.cs.ucsd.edu/users/goguen/pps/shom.ps`, August 2002.
12. C.A.R. Hoare. Correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
13. R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Intl. Joint Conf. on Artificial Intelligence*, pages 481–489. British Computer Society, 1971.
14. T. Mossakowski. Two "functional programming" sublanguages of CASL. Note L-9, in [7], March 1998.
15. D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica*, 25:233–281, 1988.
16. D. Sannella and A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development*. Cambridge University Press, 2005, to appear. See `http://homepages.inf.ed.ac.uk/dts/book/index.html`.
17. D. Smith. Designware: Software development by refinement. In *Proc. Conference on Category Theory and Computer Science, CTCS'99*, volume 29 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

A Abstract Syntax for CASL Architectural Specifications

The grammar extends the grammar given in the CASL Reference Manual [8]. The new parts of the grammar are marked in *italics*, while removed parts are ~~crossed out~~.

```
ARCH-SPEC-DEFN   ::= arch-spec-defn ARCH-SPEC-NAME ARCH-SPEC
ARCH-SPEC        ::= basic-arch-spec UNIT-DECL-DEFN+ RESULT-UNIT
                   | ARCH-SPEC-NAME
UNIT-DECL-DEFN   ::= UNIT-DECL | UNIT-DEFN

UNIT-DECL        ::= unit-decl UNIT-NAME _SPEC-REF_ ~~UNIT-IMPORTED~~
~~UNIT-IMPORTED    ::= unit-imported UNIT-TERM*~~
UNIT-DEFN        ::= unit-defn UNIT-NAME UNIT-EXPRESSION

UNIT-SPEC-DEFN   ::= unit-spec-defn SPEC-NAME UNIT-SPEC
UNIT-SPEC        ::= UNIT-TYPE | SPEC-NAME
                   ~~| arch-unit-spec ARCH-SPEC~~
                   | closed-unit-spec UNIT-SPEC
UNIT-TYPE        ::= unit-type SPEC* SPEC

_SPEC-REF-DEFN    ::= ref-unit-spec-defn SPEC-NAME SPEC-REF_
_SPEC-REF         ::= SPEC-NAME_
                   _| unit-spec UNIT-SPEC_
                   _| refinement UNIT-SPEC SYMB-MAP-ITEMS* SPEC-REF_
                   _| arch-unit-spec ARCH-SPEC_
                   _| compose-ref SPEC-REF SPEC-REF_
                   _| component-ref UNIT-REF*_

_UNIT-REF         ::= unit-ref UNIT-NAME SPEC-REF_

RESULT-UNIT      ::= result-unit UNIT-EXPRESSION
UNIT-EXPRESSION  ::= unit-expression UNIT-BINDING* UNIT-TERM
UNIT-BINDING     ::= unit-binding UNIT-NAME UNIT-SPEC
UNIT-TERM        ::= unit-translation UNIT-TERM RENAMING
                   | unit-reduction UNIT-TERM RESTRICTION
                   | amalgamation UNIT-TERM+
                   | local-unit UNIT-DEFN+ UNIT-TERM
                   | unit-appl UNIT-NAME FIT-ARG-UNIT*
FIT-ARG-UNIT     ::= fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*

ARCH-SPEC-NAME   ::= SIMPLE-ID
UNIT-NAME        ::= SIMPLE-ID
```

A `SPEC-NAME` can be a `SPEC-REF` either directly, or indirectly via `UNIT-SPEC`. This ambiguity is solved by looking up the `SPEC-NAME` in the global environment, which is expected to keep information about `UNIT-SPEC`s and `SPEC-REF`s separately.