Till Mossakowski

CASL Reference Manual

Part V: Refinement

December 14, 2003

Springer
Berlin Heidelberg New York
Hong Kong London
Milan Paris Tokyo

Contents

1 Introduction

In this part of the volume, we introduce a simple refinement language that is built on top of CASL. The material in this part is more speculative than that in the other parts. It is meant as a starting point for more complex refinement notions and development methodologies.

1.1 The Algebraic Development Paradigm

The standard development paradigm of algebraic specification [AKKB99] postulates that the development begins with a formal *requirement specification* (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, so that one finally arrives at what is often termed the *design specification*. The last refinement step then results in an actual *implementation* in a programming language, see Fig. 1.1.

$$\begin{tikzcd} \text{Requirement specification } SP_0 \arrow[dr] \\ & SP_1 \arrow[dr] \\ & & \cdots \arrow[dr] \\ & & & SP_{n-1} \arrow[dr] \\ & & & & \text{Design specification } SP_n \arrow[dr] \\ & & & & & \text{Program} \end{tikzcd}$$

Fig. 1.1. Stepwise refinement

1.2 Constructor Refinement

Casl includes both structured specifications (allowing for combining specifications) and architectural specifications (prescribing how implementation units are linked together). However, the issue of refinement between these specifications has been deliberately excluded from Casl, leaving room for several refinement languages built on top of Casl, corresponding to different methodologies. A refinement calculus for architectural specifications has been developed in [BST02].

Casl's *views* express some aspect of refinement, namely that when refining a specification more and more in the development process, the model class becomes smaller and smaller by making more and more design decisions (until a monomorphic design specification or program is reached). However, Casl's views are not expressive enough for refinement (they are primarily a means for naming fitting morphisms for parameterized specifications). This is because there are more aspects of refinement than just model class inclusion. One central issue here is so-called *constructor refinement* [ST88]. Constructor refinement means that a specification SP_1 is refined to a specification SP_2 with the help of a construction κ on SP_2-models. The refinement condition is then

$$\kappa(\mathbf{Mod}(SP_2)) \subseteq \mathbf{Mod}(SP_1)$$

Constructor refinement arises in two forms. The first form is specific to the particular specification logic, and includes the basic constructions for writing implementation units that can be found in programming languages, e.g. enumeration types, algebraic datatypes (that is, free types) and recursive definitions of operations. In specification languages, this can be modeled via *derived signature morphisms*, that may, for example, map sorts to datatype definitions and operations to terms. Since the details are institution-specific, we adopt a simple solution here: an institution-specific constructor is just given by a unit specification whose result specification is a monomorphic extension of the argument specifications. Due to monomorphicity, the looseness of the unit specification is eliminated, and (up to isomorphism) only one unit (parametric or not) is specified. In the Casl logic, this covers the usual datatypes and recursive definitions. It even covers a bit too much (like non-recursive operations); hence, further restrictions should be developed for particular institutions. For Casl, the syntactic criteria for monomorphic and definitional extensions given in Sect. (((log-log-sec:conservativity))) in connection with the Casl proof calculus provide a starting point.

The second form of constructor refinement is entirely logic independent and concerns the building of larger implementation units out of smaller ones: the task of implementing the larger unit can be decomposed into several independent subtasks consisting of the implemenation of the smaller units. This is done using Casl architectural specifications, where the smaller units are declared with their (ordinary structured) specification, and the larger unit is constructed with a unit term out of the smaller ones. The declared units can then be refined seperately. Moreover, different units can be refined in different ways (even if they happed to be declared with the same specification). This means that the structure of an architectural refinement must match that of the architectural specification being refined.

1.3 Outlook: Behavioural Refinement

Often, a refined specification does satisfy the initial requirements not 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 only differing in their “junk” entries (that is, those beyond the top pointer) exhibit the same behaviour in terms of the stack operations. Hence, they correspond to the same abstract stack and should be treated as being the same for the purpose of the refinement. This can be achieved in several ways. A simple way is to allow derived signature morphisms to map the equality symbol to any binary relation (with the semantics that the target unit is quotiented by the induced congruence relation). This can be expressed in the simple refinement language presented here by providing a monomorphic unit specification that specifies the congruence and the quotient explicity. A more elaborate way is to use observational equivalences between models, which are usually induced by sets of observable sorts [ST87]. Here, both the congruence and the quotient need not be given explicitly, but are rather constructed using observational equivalence.

2

The Refinement Language

Allow also for refinements without symbol maps, leading to a trivial signature morphism

This section introduces the abstract and concrete syntax of refinements, and describes their intended interpretation, based on CASL structured and architectural specifications.

2.1 Refinement Definitions

A refinement definition may be written into a CASL library like a specification or view definition (although strictly speaking, it does not belong to CASL proper).

```
REF-DEFN ::= ref-defn REF-NAME REFINEMENT
LIB-ITEM ::= ... | REF-DEFN
```

An refinement definition is written:

refinement RN : R
end

where the terminating '**end**' keyword is optional.

It defines the name RN to refer to the refinement R, extending the global environment (which must not already include a definition for RN).

```
REF-NAME ::= SIMPLE-ID
```

A refinement name `REF-NAME` is normally displayed in a SMALL-CAPS font, and input in mixed upper and lower case.

2.2 Refinements

A refinement is either simple, which means that a structured or unit specification is being refined, by mapping units to units (where the units are non-parameterized in the case of structured specifications, and parameterized in the case of unit specification). The other possibility is an architectural refinement, which means that all the declared units of an architectural specification are refined (by just refining their specifications).

```
REFINEMENT ::= simple-refinement REF-TYPE REF-BODY
             | arch-refinement ARCH-SPEC UNIT-REFINEMENT*
```

A simple refinement of `REF-TYPE` RT, using the construction given by the `REF-BODY` RB, is written:

$RT = RB$

It is well-formed only if the construction associated to RB, when applied to a unit of the target of RT, delivers a unit of the source of RT.

If the refinement body is empty, the simple refinement is written

RT

An architectural refinement refines the units of an architectural specification ASP, using a list $UR_1; \ldots; UR_n$ of `UNIT-REFINEMENT`s (each of the latter corresponds to a target specification and a refinement body). It is written:

ASP **to units** $UR_1; \ldots; UR_n$

2.3 Refinement Types

```
REF-TYPE ::= simple-ref-type SPEC SPEC
           | arch-ref-type SPEC ARCH-SPEC
           | unit-ref-type UNIT-SPEC UNIT-SPEC
           | arch-unit-ref-type UNIT-SPEC ARCH-SPEC
```

A refinement type `REF-TYPE` is written

SP_1 **to** SP_2

It denotes the type of refinements of units of type SP_1 to those of type SP_2 (possibly using a construction taking SP_2-models to SP_1-models). Here, the model class for `SPEC` and `UNIT-SPEC` is the standard one for structured and unit specifications, whereas the model class of an `ARCH-SPEC` consists of all interpretations of unit terms that are possible when assinging models to the declared units in a way compatible with the declarations.

Source and target type of a refinement type must be compatible in the sense that both either denote classes of unparameterized units, or both classes of parameterized units with the same number of parameters.

2.4 Refinement Bodies

Given a refinement type, a refinement body `REF-BODY` specifies the way in which units of the source are constructed out of units of the target. If parameterized units are refined, the refinement body links the result specification of the parameterized units, not their argument specifications — the latter are required to have identical signatures. Moreover, each sequence of compatible models in the domain of the source parameterized unit is required to be in the domain of the target parameterized unit. The construction on the results of parameterized units is then extended to whole parameterized units by leaving the argument units just as they are.

```
REF-BODY ::= simple-mor SYMB-MAP-ITEMS
           | named-ref REF-NAME
           | named-view VIEW-NAME FIT-ARG*
           | compose REF-NAME REF-NAME
```

A refinement body `REF-BODY` can either be a signature morphism, given by a symbol map (`SYMB-MAP-ITEMS`). In this case the associated construction is just taking reducts along the signature morphism. (An empty refinement body corresponds to an empty symbol map.)

Furthermore, a refinement body can also be a reference `REF-NAME` to a previously-defined refinement, or a reference `VIEW-NAME` to a previously-defined view (where in the case of a parameterized views, appropriate fitting arguments have to be provided, cf. Sect. **??**). Finally, a composition of two (named) refinements is written

RN_1 **then** $RN2$

It corresponds to the composition of the associated constructions.

2.5 Unit Refinements

```
UNIT-REFINEMENT ::= simple-unit-ref UNIT-NAME SPEC REF-BODY
                  | arch-unit-ref UNIT-NAME ARCH-SPEC REF-BODY
                  | unit-unit-ref UNIT-NAME UNIT-SPEC REF-BODY
```

A unit refinement is written

UN **to** $SP = RB$

If the refinement body RB is empty, it is simply written

UN **to** SP

It declares the unit name UN to be refined to the specification SP, using the construction associated to the refinement body RB. Let ASP be the architectural specification of the enclosing architectural refinement. UN must already have been declared as a (possibly parameterized) unit in ASP. The number of arguments of this unit must coincide with that of the units that are models of SP. Moreover, the enclosing architectural refinement is well-formed only if the construction associated to RB, when applied to a unit of SP, delivers a unit fulfilling the specification associated to UN in ASP.

2.6 Complete refinement trees

A specification in a library is said to have a *textindexcomplete refinement tree*, if it

- either is a unit specification whose result specification is monomorphic over the argument specifications (specific logics may impose further restrictions here in order to ensure that such specifications can be directly implemented in a programming language), or
- it is refined (via a simple of architectural refinement) to specifications having complete refinement trees.

3

Semantics

This chapter provides the semantics of refinements. It is based on the semantics of structured and architectural specifications as given in Part **??**.

3.1 Refinement Concepts

Here, we extend the semantic domains from Part **??**.

$$RN \in \mathit{RefName} = \texttt{SIMPLE-ID}$$

$$\begin{aligned}
R\Sigma = (U\Sigma_s, U\Sigma_t) &\in \mathit{RefSig} = \mathit{UnitSig} \times \mathit{UnitSig} \\
SRef = (\mathcal{U}_s, \mathcal{U}_t, RF) &\in \mathbf{SimpleRef} = \mathbf{UnitSpec} \times \mathbf{UnitSpec} \times (\mathbf{Unit} \rightharpoonup \mathbf{Unit}) \\
R_s &\in \mathit{StaticRCtx} = \mathit{UnitName} \xrightarrow{\text{fin}} \mathit{RefSig} \\
AR &\in \mathbf{ArchRef}(C_s, U\Sigma) = \mathbf{UnitEnv}(C_s) \rightharpoonup \mathbf{ArchMod}(C_s, U\Sigma) \\
&\mathbf{ArchRef} = \textstyle\bigcup_{A\Sigma \in \mathit{ArchSig}} \mathbf{ArchRef} \\
R &\in \mathit{StaticRef} = \mathit{RefSig} \cup \mathit{StaticRCtx} \\
\mathbf{R} &\in \mathbf{Ref} = \mathbf{SimpleRef} \cup \mathbf{ArchRef}
\end{aligned}$$

A *refinenment signature* (to be used for simple refinements) consists of a source and a target unit signature. The parameter signatures of $U\Sigma_1$ and $U\Sigma_2$ of a refinement signature $R\Sigma$ are required to be the same. A *refinenment function* provides the correspondings model semantics. It consists of two unit specifications (one for the source and one for the target of the refinement), plus the actual refinement function. A refinement function $SRef = (\mathcal{U}_s, \mathcal{U}_t, RF)$ is required to actually go from the target of the refinement to the source, that is, $Dom(RF) = \mathcal{U}_t$ and for all $U \in \mathcal{U}_t$, $RF(U) \in \mathcal{U}_s$.

We now come to the corresponding notions for architectural refinements. A *static refinement context* is given by a partial map from unit names (that are intended to coincide with those of an architectural specfification) to refinement signatures (stating how the respective unit is to be refined). Given a static unit context C_s and a unit signature $U\Sigma$, an *architectural refinement* over C_s and $U\Sigma$ is a partial map from unit environements over C_s (giving units for the target specifications of the involved unit specification) to architectural models of $(C_s, U\Sigma)$.

A *static refinement* is either a refinement signature or a static refinement context, and a *refinement* is either a simple refinement or an architectural refinement.

We also need to add a further component to the global environment, capturing refinement signatures and functions. Actually, for architectural refinements we will need sequences of these. A static global environment Γ_s now is a five-tuple $(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s)$, where

$$\mathcal{T}_s = \mathit{RefName} \xrightarrow{\text{fin}} \mathit{StaticRef}$$

Similarly, a model global environment Γ_m is now a five-tuple $(\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m, \mathcal{R}_m)$, where

$$\mathcal{R}_m = \mathit{RefName} \xrightarrow{\text{fin}} \mathbf{Ref}$$

3.2 Refinement definitions

```
REF-DEFN ::= ref-defn REF-NAME REFINEMENT
LIB-ITEM ::= ... | REF-DEFN
REF-NAME ::= SIMPLE-ID
```

$$\boxed{\Gamma_s \vdash \texttt{REF-DEFN} \rhd \Gamma'_s}$$

$$\boxed{\Gamma_s, \Gamma_m \vdash \texttt{REF-DEFN} \Rightarrow \Gamma'_m}$$

Let Γ_s be $(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s)$, then Γ'_s is $(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}'_s)$ where $\mathcal{G}'_s$ is $\mathcal{R}_s$ extended by an association

$$\texttt{RN} \mapsto R$$

provided that RN is not in the domain of $\mathcal{R}_s$ and $R \in \mathit{StaticRef}$.

Let Γ_m be $(\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m, \mathcal{R}_m)$, then Γ'_m is $(\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m, \mathcal{R}'_m)$, where $\mathcal{R}'_m$ is $\mathcal{R}_m$ extended by an association

$$\texttt{RN} \mapsto \mathbf{R}$$

provided that $\mathcal{R}_m$ does not already contain an association for RN, $\mathbf{R}$ is in $\mathbf{Ref}$ and GS_s, and GS_m are compatible.

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s) \\ \mathtt{RN} \notin Dom(\mathcal{G}_s) \cup Dom(\mathcal{V}_s) \cup Dom(\mathcal{A}_s) \cup Dom(\mathcal{T}_s) \cup Dom(\mathcal{R}_s) \\ \Gamma_s \vdash \mathtt{REFINEMENT} \rhd R \\ \mathcal{R}'_s = \mathcal{R}_s \cup \{\mathtt{RN} \mapsto R\}\end{array}}{\Gamma_s \vdash \mathtt{ref\text{-}defn}\ \mathtt{RN}\ \mathtt{REFINEMENT} \rhd (\mathcal{G}'_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s)}$$

$$\frac{\begin{array}{c}\Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m, \mathcal{R}_m) \\ \mathtt{RN} \notin Dom(\mathcal{G}_m) \cup Dom(\mathcal{V}_m) \cup Dom(\mathcal{A}_m) \cup Dom(\mathcal{T}_m) \cup Dom(\mathcal{R}_m) \\ \Gamma_s, \Gamma_m \vdash \mathtt{REFINEMENT} \Rightarrow \mathbf{R} \\ \mathcal{R}'_m = \mathcal{R}_m \cup \{\mathtt{RN} \mapsto \mathbf{R}\}\end{array}}{\Gamma_s, \Gamma_m \vdash \mathtt{ref\text{-}defn}\ \mathtt{RN}\ \mathtt{REFINEMENT} \Rightarrow (\mathcal{G}'_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m, \mathcal{R}_m)}$$

3.3 Refinements

```
REFINEMENT ::= simple-refinement REF-TYPE REF-BODY
             | arch-refinement ARCH-SPEC UNIT-REFINEMENT*
```

$$\boxed{\Gamma_s \vdash \mathtt{REFINEMENT} \rhd R \qquad \Gamma_s, \Gamma_m \vdash \mathtt{REFINEMENT} \Rightarrow \mathbf{R}}$$

Γ_s and Γ_m are compatible global environments. $R \in \mathit{StaticRef}$ is a static refinement, and $\mathbf{R} \in \mathbf{Ref}$ is a refinement.

$$\frac{\begin{array}{c}\Gamma_s \vdash \mathtt{REF\text{-}TYPE} \rhd R\Sigma \\ R\Sigma, \Gamma_s \vdash \mathtt{REF\text{-}BODY} \rhd R\Sigma\end{array}}{\Gamma_s \vdash \mathtt{simple\text{-}refinement}\ \mathtt{REF\text{-}TYPE}\ \mathtt{REF\text{-}BODY} \rhd R\Sigma}$$

$$\frac{\begin{array}{c}\Gamma_s, \Gamma_m \vdash \mathtt{REF\text{-}TYPE} \Rightarrow (\mathcal{U}_s, \mathcal{U}_t) \\ (\mathcal{U}_s, \mathcal{U}_t), \Gamma_s, \Gamma_m \vdash \mathtt{REF\text{-}BODY} \Rightarrow (\mathcal{U}_s, \mathcal{U}_t, RF)\end{array}}{\Gamma_s, \Gamma_m \vdash \mathtt{simple\text{-}refinement}\ \mathtt{REF\text{-}TYPE}\ \mathtt{REF\text{-}BODY} \Rightarrow (\mathcal{U}_s, \mathcal{U}_t, RF)}$$

$$\frac{\begin{array}{c}\Gamma_s \vdash \mathtt{ARCH\text{-}SPEC} \rhd A\Sigma \\ A\Sigma, \Gamma_s \vdash \mathtt{UR}_1 \rhd (UN_1, R\Sigma_1) \\ \dots \\ A\Sigma, \Gamma_s \vdash \mathtt{UR}_n \rhd (UN_n, R\Sigma_n) \\ A\Sigma = (C_s, U\Sigma) \\ Dom(C_s) = \{UN_1, \dots, UN_n\} \\ R_s = \{UN_i \mapsto R\Sigma_i \mid i = 1 \dots n\}\end{array}}{\Gamma_s \vdash \mathtt{arch\text{-}refinement}\ \mathtt{ARCH\text{-}SPEC}\ \mathtt{UR}_1;\ \dots;\ \mathtt{UR}_n \rhd R_s}$$

$$\frac{\begin{array}{c}\Gamma_s \vdash \texttt{ARCH-SPEC} \rhd A\Sigma \\ A\Sigma = (C_s, U\Sigma) \\ Dom(C_s) = \{UN_1, \ldots, UN_n\} \\ \Gamma_s, \Gamma_m \vdash \texttt{ARCH-SPEC} \Rightarrow \mathcal{AM} \\ A\Sigma, \Gamma_s, \Gamma_m \vdash \texttt{UR}_1 \Rightarrow (UN_1, (\mathcal{U}_1, \mathcal{U}'_1, RF_1)) \\ \ldots \\ A\Sigma, \Gamma_s, \Gamma_m \vdash \texttt{UR}_n \Rightarrow (UN_n, (\mathcal{U}_n, \mathcal{U}'_n, RF_n)) \\ Dom(AR) = \{E \mid E(UN_i) \in \mathcal{U}'_i, i = 1 \ldots n \\ \text{and there is } (E', U) \in \mathcal{AM} \text{ with } E'(UN_i) \in RF_i(E(UN_i)), i = 1 \ldots n\} \\ AR(E) = (E', U) \text{ if} \\ E(UN_i) \in \mathcal{U}'_i \text{ and } E'(UN_i) \in RF_i(E(UN_i)), i = 1 \ldots n \text{ and } (E', U) \in \mathcal{AM}\end{array}}{\Gamma_s, \Gamma_m \vdash \texttt{arch-refinement ARCH-SPEC UR}_1\texttt{; } \ldots\texttt{; UR}_n \Rightarrow AR}$$

3.4 Refinement types

```
REF-TYPE ::= simple-ref-type SPEC SPEC
           | arch-ref-type SPEC ARCH-SPEC
           | unit-ref-type UNIT-SPEC UNIT-SPEC
           | arch-unit-ref-type UNIT-SPEC ARCH-SPEC
```

$$\boxed{\Gamma_s \vdash \texttt{REF-TYPE} \rhd R\Sigma \qquad \Gamma_s, \Gamma_m \vdash \texttt{REF-TYPE} \Rightarrow (\mathcal{U}_s, \mathcal{U}_t)}$$

Γ_s and Γ_m are compatible global environments. $R\Sigma$ is a refinement signature, and $(\mathcal{U}_s, \mathcal{U}_t)$ a pair of unit classes.

$$\frac{\begin{array}{c}\emptyset, \Gamma_s \vdash \texttt{SPEC}_1 \rhd \Sigma_1 \\ \emptyset, \Gamma_s \vdash \texttt{SPEC}_2 \rhd \Sigma_2\end{array}}{\Gamma_s \vdash \texttt{simple-ref-type SPEC}_1\ \texttt{SPEC}_2 \rhd (\Sigma_1, \Sigma_2)}$$

$$\frac{\begin{array}{c}\emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \texttt{SPEC}_1 \Rightarrow \mathcal{M}_1 \\ \emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \texttt{SPEC}_2 \Rightarrow \mathcal{M}_2\end{array}}{\Gamma_s, \Gamma_m \vdash \texttt{simple-ref-type SPEC}_1\ \texttt{SPEC}_2 \Rightarrow (\mathcal{M}_1, \mathcal{M}_2)}$$

$$\frac{\begin{array}{c}\emptyset, \Gamma_s \vdash \texttt{SPEC} \rhd \Sigma_1 \\ \Gamma_s \vdash \texttt{ARCH-SPEC} \rhd (C_s, \Sigma_2) \\ \Sigma_2 \text{ is parameterless}\end{array}}{\Gamma_s \vdash \texttt{arch-ref-type SPEC ARCH-SPEC} \rhd (\Sigma_1, \Sigma_2)}$$

$$\frac{\begin{array}{c}\emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M} \\ \Gamma_s, \Gamma_m \vdash \texttt{ARCH-SPEC} \Rightarrow \mathcal{AM}\end{array}}{\Gamma_s, \Gamma_m \vdash \texttt{arch-ref-type SPEC ARCH-SPEC} \Rightarrow (\mathcal{M}, \{M \mid (E, M) \in \mathcal{AM}\})}$$

$$\frac{\begin{array}{c}\emptyset, \Gamma_s \vdash \texttt{UNIT-SPEC}_1 \rhd U\Sigma_1 \\ \emptyset, \Gamma_s \vdash \texttt{UNIT-SPEC}_2 \rhd U\Sigma_2 \\ U\Sigma_1 \text{ and } U\Sigma_2 \text{ have the same parameter signatures}\end{array}}{\Gamma_s \vdash \texttt{unit-ref-type UNIT-SPEC}_1\ \texttt{UNIT-SPEC}_2 \rhd (U\Sigma_1, U\Sigma_2)}$$

$$\frac{\begin{array}{c}\emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \texttt{UNIT-SPEC}_1 \Rightarrow \mathcal{U}_1 \\ \emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \texttt{UNIT-SPEC}_2 \Rightarrow \mathcal{U}_2\end{array}}{\Gamma_s, \Gamma_m \vdash \texttt{unit-ref-type UNIT-SPEC}_1\ \texttt{UNIT-SPEC}_2 \Rightarrow (\mathcal{U}_1, \mathcal{U}_2)}$$

$$\frac{\begin{array}{c}\emptyset, \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd U\Sigma_1 \\ \emptyset, \Gamma_s \vdash \texttt{ARCH-SPEC} \rhd (C_s, U\Sigma_2) \\ U\Sigma_1 \text{ and } U\Sigma_2 \text{ have the same parameter signatures}\end{array}}{\Gamma_s \vdash \texttt{arch-unit-ref-type UNIT-SPEC ARCH-SPEC} \rhd (U\Sigma_1, U\Sigma_2)}$$

$$\frac{\begin{array}{c}\emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \texttt{UNIT-SPEC} \Rightarrow \mathcal{U} \\ \emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \texttt{ARCH-SPEC} \Rightarrow \mathcal{AM}\end{array}}{\Gamma_s, \Gamma_m \vdash \texttt{arch-unit-ref-type UNIT-SPEC ARCH-SPEC} \Rightarrow (\mathcal{U}, \{M \mid (U, M) \in \mathcal{AM}\})}$$

3.5 Refinement bodies

```
REF-BODY ::= simple-mor SYMB-MAP-ITEMS
           | gen-mor SYMB-MAP-ITEMS SPEC
           | named-ref REF-NAME
           | named-view VIEW-NAME FIT-ARG*
           | compose REF-NAME REF-NAME
```

$$\boxed{R\Sigma, \Gamma_s \vdash \texttt{REF-BODY} \rhd R\Sigma \qquad (\mathcal{U}_s, \mathcal{U}_t), \Gamma_s, \Gamma_m \vdash \texttt{REF-BODY} \Rightarrow SRef}$$

Γ_s and Γ_m are compatible global environments. $R\Sigma$ is a refinement signature. $\mathcal{U}_s$ and $\mathcal{U}_t$ are unit specifications and $SRef$ is a refinement function.

$$\frac{\begin{array}{c}U\Sigma_s = \Sigma_1, \ldots, \Sigma_n \to \Sigma_s \\ U\Sigma_t = \Sigma_1, \ldots, \Sigma_n \to \Sigma_t \\ \vdash \texttt{SYMB-MAP-ITEMS} \rhd r \\ \sigma = r|^{\Sigma_s}_{\Sigma_t}\end{array}}{(U\Sigma_s, U\Sigma_t), \Gamma_s \vdash \texttt{simple-mor SYMB-MAP-ITEMS} \rhd (U\Sigma_s, U\Sigma_t)}$$

$$\frac{\begin{array}{c}\vdash \texttt{SYMB-MAP-ITEMS} \rhd r \\ \sigma = r|^{\Sigma_s}_{\Sigma} \\ Dom(RF) = \mathcal{U}_t \\ Dom(RF(F)) = Dom(F) \\ RF(F)(M_1, \ldots, M_n) = F(M_1, \ldots, M_n)|_\sigma \\ \text{for } (M_1, \ldots, M_n) \in Dom(RF(F)) \\ RF(F) \in \mathcal{U}_s \text{ for } F \in \mathcal{U}_t\end{array}}{(\mathcal{U}_s, \mathcal{U}_t), \Gamma_s, \Gamma_m \vdash \texttt{simple-mor SYMB-MAP-ITEMS} \Rightarrow (\mathcal{U}_s, \mathcal{U}_t, RF)}$$

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s) \\ (\texttt{RN} \mapsto (U\Sigma_s, U\Sigma_t)) \in \mathcal{R}_s\end{array}}{(U\Sigma_s, U\Sigma_t), \Gamma_s \vdash \texttt{named-ref RN} \rhd (U\Sigma_s, U\Sigma_t)}$$

$$\frac{\begin{array}{c}\Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m, \mathcal{R}_m) \\ (\texttt{RN} \mapsto (\mathcal{U}_s, \mathcal{U}_t, RF) \in \mathcal{R}_m\end{array}}{(\mathcal{U}_s, \mathcal{U}_t), \Gamma_s, \Gamma_m \vdash \texttt{named-ref RN} \Rightarrow (\mathcal{U}_s, \mathcal{U}_t, RF)}$$

Should semantics for `FIT-VIEW`s be split in order to avoid these repetitions?

Concerning views as refinement bodies, we adapt the rules for the semantics of `FIT-VIEW`s from Sect. (((sem-sem-sec-FittingViews))). First we study the situation of a non-generic view.

$$\frac{\begin{array}{c}U\Sigma_s = \Sigma_1, \ldots, \Sigma_n \to \Sigma_s \\ U\Sigma_t = \Sigma_1, \ldots, \Sigma_n \to \Sigma_t \\ \Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s) \\ (\texttt{VN} \mapsto (\Sigma_s, \sigma, (\emptyset, (), \Sigma_t))) \in \mathcal{V}_s\end{array}}{(U\Sigma_s, U\Sigma_t), \Gamma_s \vdash \texttt{named-view VN} \rhd (U\Sigma_s, U\Sigma_t)}$$

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s) \\ (\texttt{VN} \mapsto (\Sigma_s, \sigma, (\emptyset, (), \Sigma_t))) \in \mathcal{V}_s \\ \Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m, \mathcal{R}_m) \\ (\texttt{VN} \mapsto (\mathcal{M}_s, (\mathcal{M}_\perp, (), \mathcal{M}_t)) \in \mathcal{V}_m \\ Dom(RF) = \mathcal{U}_t \\ Dom(RF(F)) = Dom(F) \\ RF(F)(M_1, \ldots, M_n) = |_{F(M_1, \ldots, M_n)}\sigma \\ \text{for all } F \in \mathcal{U}_t,\ RF(F) \in \mathcal{U}_s\end{array}}{(\mathcal{U}_s, \mathcal{U}_t), \Gamma_s, \Gamma_m \vdash \texttt{named-view VN} \Rightarrow (\mathcal{U}_s, \mathcal{U}_t, RF)}$$

Now we come to the case of generic views.

$$\frac{\begin{array}{c} U\Sigma_s = \Sigma_1,\ldots,\Sigma_n \to \Sigma_s \\ U\Sigma_t = \Sigma_1,\ldots,\Sigma_n \to \Sigma_t \\ \Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s) \\ (\mathtt{VN} \mapsto (\Sigma_s, \sigma, GS_s)) \in \mathcal{V}_s \\ GS_s = (\Sigma'_I, (\Sigma_1,\ldots,\Sigma_n), \Sigma_B) \\ n \geq 1 \\ \Sigma'_I, \Sigma_1, \Gamma_s \vdash \mathtt{FA}_1 \rhd \sigma_1, \Sigma^A_1 \\ \ldots \\ \Sigma'_I, \Sigma_n, \Gamma_s \vdash \mathtt{FA}_n \rhd \sigma_n, \Sigma^A_n \\ (\Sigma_A, \sigma'_f) = GS_s((\Sigma^A_1, \sigma_1),\ldots,(\Sigma^A_n, \sigma_n)) \text{ is defined} \\ \Sigma_A = \Sigma_t \end{array}}{(U\Sigma_s, U\Sigma_t), \Gamma_s \vdash \mathtt{named\text{-}view}\ \mathtt{VN}\ \mathtt{FA}_1 \ldots \mathtt{FA}_n \rhd (U\Sigma_s, U\Sigma_t)}$$

$$\frac{\begin{array}{c} \Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s) \\ (\mathtt{VN} \mapsto (\Sigma_s, \sigma, GS_s)) \in \mathcal{V}_s \\ GS_s = (\Sigma'_I, (\Sigma_1,\ldots,\Sigma_n), \Sigma_B) \\ n \geq 1 \\ \Sigma'_I, \Sigma_1, \Gamma_s \vdash \mathtt{FA}_1 \rhd \sigma_1, \Sigma^A_1 \\ \ldots \\ \Sigma'_I, \Sigma_n, \Gamma_s \vdash \mathtt{FA}_n \rhd \sigma_n, \Sigma^A_n \\ \Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m, \mathcal{R}_m) \\ (\mathtt{VN} \mapsto (\mathcal{M}_s, GS_m)) \in \mathcal{V}_m \\ GS_m = (\mathcal{M}'_I, (\mathcal{M}_1,\ldots,\mathcal{M}_n), \mathcal{M}_B) \\ \Sigma'_I, \Sigma_1, \mathcal{M}'_I, \mathcal{M}_1, \Gamma_s, \Gamma_m \vdash \mathtt{FA}_1 \Rightarrow \mathcal{M}^A_1 \\ \ldots \\ \Sigma'_I, \Sigma_n, \mathcal{M}'_I, \mathcal{M}_n, \Gamma_s, \Gamma_m \vdash \mathtt{FA}_n \Rightarrow \mathcal{M}^A_n \\ \mathcal{M}_A = GS_m((\mathcal{M}^A_1, \sigma_1),\ldots,(\mathcal{M}^A_n, \sigma_n)) \\ Dom(RF) = \mathcal{U}_t \\ \text{for all } RF \in \mathcal{U}_t,\ \text{for all } F \in Dom(RF)\, (M_1,\ldots,M_n) \in Dom(F)\,. \\ RF(F)(M_1,\ldots,M_n) \in \mathcal{M}_A \\ RF(F)(M_1,\ldots,M_n) = F(M_1,\ldots,M_n)|_{\sigma'_f \circ \sigma} \\ \text{for all } F \in \mathcal{U}_t,\ RF(F) \in \mathcal{U}_s \end{array}}{(\mathcal{U}_s, \mathcal{U}_t), \Gamma_s, \Gamma_m \vdash \mathtt{named\text{-}view}\ \mathtt{VN}\ \mathtt{FA}_1 \ldots \mathtt{FA}_n \Rightarrow (\mathcal{U}_s, \mathcal{U}_t, RF)}$$

$$\frac{\begin{array}{c} \Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s) \\ (\mathtt{RN}_1 \mapsto (U\Sigma_1, U\Sigma_2)) \in \mathcal{R}_s \\ (\mathtt{RN}_2 \mapsto (U\Sigma_3, U\Sigma_4)) \in \mathcal{R}_s \\ U\Sigma_4 = U\Sigma_1 \end{array}}{(U\Sigma_2, U\Sigma_3), \Gamma_s \vdash \mathtt{compose}\ \mathtt{RN}_1\ \mathtt{RN}_2 \rhd (U\Sigma_2, U\Sigma_3)}$$

$$\frac{\begin{array}{c}\Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m, \mathcal{R}_m)\\ (\mathtt{RN}_1 \mapsto (\mathcal{U}_1, \mathcal{U}_2, RF_1) \in \mathcal{R}_m\\ (\mathtt{RN}_2 \mapsto (\mathcal{U}_3, \mathcal{U}_4, RF_2)) \in \mathcal{R}_m\\ \mathcal{U}_4 \subseteq \mathcal{U}_1\\ RF(F) = RF_1(RF_2(F))\end{array}}{(\mathcal{U}_2, \mathcal{U}_3), \Gamma_s, \Gamma_m \vdash \mathtt{compose}\ \mathtt{RN}_1\ \mathtt{RN}_2 \Rightarrow (\mathcal{U}_2, \mathcal{U}_3, RF)}$$

3.6 Unit refinements

```
UNIT-REFINEMENT ::= simple-unit-ref UNIT-NAME SPEC REF-BODY
                  | arch-unit-ref UNIT-NAME ARCH-SPEC REF-BODY
                  | unit-unit-ref UNIT-NAME UNIT-SPEC REF-BODY
```

$\boxed{A\Sigma, \Gamma_s \vdash \mathtt{UNIT\text{-}REFINEMENT} \rhd (UN, R\Sigma) \qquad A\Sigma, \Gamma_s, \Gamma_m \vdash \mathtt{UNIT\text{-}REFINEMENT} \Rightarrow (U}$

Γ_s and Γ_m are compatible global environments. $A\Sigma$ is an architectural signature. UN is a unit name. $R\Sigma$ is a refinement signature, and $SRef$ is a refinement function.

In the model semantics rules below, we are completely liberal about the source unit specifications of the refinement body. This is outweighed by the fact that there is a check in the model semantics of refinements above that the refined units together form a model of the source architectural specification.

Better carry the source specs around?

$$\frac{\begin{array}{c}\mathtt{UN} \in Dom(C_s)\\ C_s(\mathtt{UN}) = \Sigma_s\\ \emptyset, \Gamma_s \vdash \mathtt{SPEC} \rhd \Sigma_t\\ (\Sigma_s, \Sigma_t), \Gamma_s \vdash \mathtt{RB} \rhd (\Sigma_s, \Sigma_t)\end{array}}{(C_s, U\Sigma), \Gamma_s \vdash \mathtt{simple\text{-}unit\text{-}ref}\ \mathtt{UN}\ \mathtt{SPEC}\ \mathtt{RB} \rhd (\mathtt{UN}, (\Sigma_s, \Sigma_t))}$$

$$\frac{\begin{array}{c}\mathtt{UN} \in Dom(C_s)\\ C_s(\mathtt{UN}) = \Sigma_s\\ \emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC} \Rightarrow \mathcal{M}_t\\ \mathcal{M}_s = \mathbf{Mod}(\Sigma)\\ (\mathcal{M}_s, \mathcal{M}_t), \Gamma_s, \Gamma_m \vdash \mathtt{RB} \Rightarrow (\mathcal{M}_s, \mathcal{M}_t, RF)\end{array}}{(C_s, U\Sigma), \Gamma_s, \Gamma_m \vdash \mathtt{simple\text{-}unit\text{-}ref}\ \mathtt{UN}\ \mathtt{SPEC}\ \mathtt{RB} \Rightarrow (\mathtt{UN}(\mathcal{M}_s, \mathcal{M}_t, RF,)}$$

$$\frac{\begin{array}{c}\mathtt{UN} \in Dom(C_s)\\ C_s(\mathtt{UN}) = U\Sigma_s\\ \Gamma_s \vdash \mathtt{ARCH\text{-}SPEC} \rhd (C_s, U\Sigma_t)\\ (U\Sigma_s, U\Sigma_t), \Gamma_s \vdash \mathtt{RB} \rhd (U\Sigma_s, U\Sigma_t)\end{array}}{(C_s, U\Sigma), \Gamma_s \vdash \mathtt{arch\text{-}unit\text{-}ref}\ \mathtt{UN}\ \mathtt{ARCH\text{-}SPEC}\ \mathtt{RB} \rhd (\mathtt{UN}, (U\Sigma_s, U\Sigma_t))}$$

$$\frac{\begin{array}{c}\texttt{UN} \in Dom(C_s) \\ C_s(\texttt{UN}) = U\Sigma_s \\ \Gamma_s, \Gamma_m \vdash \texttt{ARCH-SPEC} \Rightarrow \mathcal{AM} \\ \mathcal{U}_s = \mathbf{Mod}(U\Sigma_s) \\ \mathcal{U}_t = \{F \mid (E, F) \in \mathcal{AM}\} \\ (\mathcal{U}_s, \mathcal{U}_t), \Gamma_s, \Gamma_m \vdash \texttt{RB} \Rightarrow (\mathcal{U}_s, \mathcal{U}_t, RF)\end{array}}{(C_s, U\Sigma), \Gamma_s, \Gamma_m \vdash \texttt{arch-unit-ref UN ARCH-SPEC RB} \Rightarrow (\texttt{UN}, (\mathcal{U}_s, \mathcal{U}_t, RF))}$$

$$\frac{\begin{array}{c}\texttt{UN} \in Dom(C_s) \\ C_s(\texttt{UN}) = U\Sigma_s \\ \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd U\Sigma_t \\ (U\Sigma_s, \Sigma_t), \Gamma_s \vdash \texttt{RB} \rhd (U\Sigma_s, U\Sigma_t)\end{array}}{(C_s, U\Sigma), \Gamma_s \vdash \texttt{unit-unit-ref UN UNIT-SPEC RB} \rhd (\texttt{UN}, (U\Sigma_s, U\Sigma_t))}$$

$$\frac{\begin{array}{c}\texttt{UN} \in Dom(C_s) \\ C_s(\texttt{UN}) = U\Sigma_s \\ \Gamma_s, \Gamma_m \vdash \texttt{UNIT-SPEC} \Rightarrow \mathcal{U}_t \\ \mathcal{U}_s = \mathbf{Mod}(U\Sigma_s) \\ (\mathcal{U}_s, \mathcal{U}_t), \Gamma_s, \Gamma_m \vdash \texttt{RB} \Rightarrow (\mathcal{U}_s, \mathcal{U}_t, RF)\end{array}}{(C_s, U\Sigma), \Gamma_s, \Gamma_m \vdash \texttt{unit-unit-ref UN UNIT-SPEC RB} \Rightarrow (\texttt{UN}, (U\Sigma_s, U\Sigma_t))}$$

3.7 Complete refinement trees

4

Calculus

This chapter provides a proof calculus for the simple refinement language presented in the previous chapters. The proof calculus allows to capture the success of the model-theoretic semantics for refinement with proof rules. With this, well-formedness of refinements becomes expressible entirely with rules of the static semantics and the calculus. The proof rules also can be understood to contribute to the well-formedness of libraries in the sense of Chap. (((log-log-part-Libraries))).

Strictly speaking, here we do not provide a calculus, but a verification static semantics based on that structured as well as architectural specifications as presented in Part **??**. The verification semantics generates proof obligations in form of theorem links in a development graph, and these can be checked with the calculus for development graphs. The proof obligations express that models of the target of the refinement specification are mapped to models of the source specification.

We begin with introducing the verification counterparts of the notions of refinement signature and static refinement context from Chap. 3. They are obtained by simply replacing signatures with development graph nodes and hence unit signatures by verification unit signatures as introduced in (((log-log-part-Libraries))), while keeping the requirements imposed there (here understood as requirements of the signatures associated to the nodes):

$$\begin{aligned} R\Sigma = (U\Sigma_s, U\Sigma_t) \in\ & \mathit{VerRefSig} = \mathit{VerUnitSig} \times \mathit{VerUnitSig} \\ R_s \in\ & \mathit{VerStaticRCtx} = \mathit{UnitName} \xrightarrow{\text{fin}} \mathit{VerRefSig} \\ R \in\ & \mathit{VerStaticRef} = \mathit{VerRefSig} \cup \mathit{VerStaticRCtx} \end{aligned}$$

The verification static global environments from (((log-log-part-Libraries))) are extended accordingly:

A verification static global environment Γ_s is a five-tuple $(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s)$, where

$$\mathcal{T}_s = \mathit{RefName} \xrightarrow{\text{fin}} \mathit{VerStaticRef}$$

We now come to the verification static semantics for refinements.

4.1 Refinement definitions

$$\boxed{\Gamma_s, (\mathcal{S}, Th) \vdash \texttt{REF-DEFN} \ggg \Gamma'_s, (\mathcal{S}', Th')}$$

$\Gamma_s, (\mathcal{S}, Th)$ is a verification static global environment. $(\mathcal{S}', Th')$ is a development graph extending $(\mathcal{S}, Th)$. Let Γ_s be $(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s)$, then Γ'_s is $(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}'_s)$ where $\mathcal{G}'_s$ is $\mathcal{R}_s$ extended by an association

$$\texttt{RN} \mapsto R$$

provided that RN is not in the domain of $\mathcal{R}_s$ and $R \in \mathit{VerStaticRef}$.

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s) \\ \texttt{RN} \notin \mathit{Dom}(\mathcal{G}_s) \cup \mathit{Dom}(\mathcal{V}_s) \cup \mathit{Dom}(\mathcal{A}_s) \cup \mathit{Dom}(\mathcal{T}_s) \cup \mathit{Dom}(\mathcal{R}_s) \\ \Gamma_s, (\mathcal{S}, Th) \vdash \texttt{REFINEMENT} \ggg R, (\mathcal{S}', Th') \\ \mathcal{R}'_s = \mathcal{R}_s \cup \{\texttt{RN} \mapsto R\}\end{array}}{\Gamma_s, (\mathcal{S}, Th) \vdash \texttt{ref-defn RN REFINEMENT} \ggg (\mathcal{G}'_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s), (\mathcal{S}', Th')}$$

4.2 Refinements

$$\boxed{\Gamma_s, (\mathcal{S}, Th) \vdash \texttt{REFINEMENT} \ggg R, (\mathcal{S}', Th')}$$

$\Gamma_s, (\mathcal{S}, Th)$ is a verification static global environment. $(\mathcal{S}', Th')$ is a development graph extending $(\mathcal{S}, Th)$. $R \in \mathit{VerStaticRef}$ is a verification static refinement.

$$\frac{\begin{array}{c}\Gamma_s, (\mathcal{S}_1, Th_1) \vdash \texttt{REF-TYPE} \ggg R\Sigma, (\mathcal{S}_2, Th_2) \\ R\Sigma, \Gamma_s, (\mathcal{S}_2, Th_2) \vdash \texttt{REF-BODY} \ggg R\Sigma, (\mathcal{S}_3, Th_3)\end{array}}{\Gamma_s, (\mathcal{S}_1, Th_1) \vdash \texttt{simple-refinement REF-TYPE REF-BODY} \ggg R\Sigma, (\mathcal{S}_3, Th_3)}$$

$$\frac{\begin{array}{c}\Gamma_s, (\mathcal{S}, Th) \vdash \texttt{ARCH-SPEC} \ggg A\Sigma, (\mathcal{S}', Th') \\ A\Sigma, \Gamma_s, (\mathcal{S}', Th') \vdash \texttt{UR}_1 \ggg (UN_1, R\Sigma_1), (\mathcal{S}_1, Th_1) \\ \dots \\ A\Sigma, \Gamma_s, (\mathcal{S}_{n-1}, Th_{n-1}) \vdash \texttt{UR}_n \ggg (UN_n, R\Sigma_n), (\mathcal{S}_n, Th_n) \\ A\Sigma = (C_s, U\Sigma) \\ \mathit{Dom}(C_s) = \{UN_1, \dots, UN_n\} \\ R_s = \{UN_i \mapsto R\Sigma_i \mid i = 1 \dots n\}\end{array}}{\Gamma_s, (\mathcal{S}, Th) \vdash \texttt{arch-refinement ARCH-SPEC UR}_1\texttt{;} \dots \texttt{; UR}_n \ggg R_s, (\mathcal{S}_n, Th_n)}$$

4.3 Refinement types

$$\boxed{\Gamma_s, (\mathcal{S}, Th) \vdash \texttt{REF-TYPE} \ggg R\Sigma, (\mathcal{S}', Th')}$$

$\Gamma_s, (\mathcal{S}, Th)$ is a verification static global environment. $(\mathcal{S}', Th')$ is a development graph extending $(\mathcal{S}, Th)$. $R\Sigma$ is a verification refinement signature.

$$\frac{\begin{array}{c}\emptyset, \Gamma_s, (\mathcal{S}_1, Th_1) \vdash \texttt{SPEC}_1 \ggg N_1, (\mathcal{S}_2, Th_2) \\ \emptyset, \Gamma_s, (\mathcal{S}_2, Th_2) \vdash \texttt{SPEC}_2 \ggg N_2, (\mathcal{S}_3, Th_3)\end{array}}{\Gamma_s, (\mathcal{S}_1, Th_1) \vdash \texttt{simple-ref-type SPEC}_1 \texttt{ SPEC}_2 \ggg (\Sigma_1, \Sigma_2), (\mathcal{S}_3, Th_3)}$$

$$\frac{\begin{array}{c}\emptyset, \Gamma_s, (\mathcal{S}_1, Th_1) \vdash \texttt{SPEC} \ggg N_1, (\mathcal{S}_2, Th_2) \\ \Gamma_s, (\mathcal{S}_2, Th_2) \vdash \texttt{ARCH-SPEC} \ggg (C_s, N_2), (\mathcal{S}_3, Th_3) \\ N_2 \text{ is a node, i.e. parameterless}\end{array}}{\Gamma_s, (\mathcal{S}_1, Th_1) \vdash \texttt{arch-ref-type SPEC ARCH-SPEC} \ggg (N_1, N_2), (\mathcal{S}_3, Th_3)}$$

$$\frac{\begin{array}{c}\emptyset, \Gamma_s, (\mathcal{S}_1, Th_1) \vdash \texttt{UNIT-SPEC}_1 \ggg U\Sigma_1, (\mathcal{S}_2, Th_2) \\ \emptyset, \Gamma_s, (\mathcal{S}_2, Th_2) \vdash \texttt{UNIT-SPEC}_2 \ggg U\Sigma_2, (\mathcal{S}_3, Th_3) \\ U\Sigma_1 = N_1, \ldots, N_n \to N \\ U\Sigma_2 = N_1', \ldots, N_n' \to N' \\ \Sigma^{N_i} = \Sigma^{N_i'} \ (i = 1, \ldots, n) \\ Th = Th_3 \cup \{ \ N_i' \stackrel{id}{=\!=\!\Rightarrow} N_i \ \mid i = 1, \ldots, n\}\end{array}}{\Gamma_s, (\mathcal{S}_1, Th_1) \vdash \texttt{unit-ref-type UNIT-SPEC}_1 \texttt{ UNIT-SPEC}_2 \ggg (U\Sigma_1, U\Sigma_2), (\mathcal{S}_3, Th)}$$

$$\frac{\begin{array}{c}\emptyset, \Gamma_s, (\mathcal{S}_1, Th_1) \vdash \texttt{UNIT-SPEC} \ggg U\Sigma_1, (\mathcal{S}_2, Th_2) \\ \emptyset, \Gamma_s, (\mathcal{S}_2, Th)_2 \vdash \texttt{ARCH-SPEC} \ggg (C_s, U\Sigma_2), (\mathcal{S}_3, Th_3) \\ U\Sigma_1 = N_1, \ldots, N_n \to N \\ U\Sigma_2 = N_1', \ldots, N_n' \to N' \\ \Sigma^{N_i} = \Sigma^{N_i'} \ (i = 1, \ldots, n) \\ Th = Th_3 \cup \{ \ N_i' \stackrel{id}{=\!=\!\Rightarrow} N_i \ \mid i = 1, \ldots, n\}\end{array}}{\Gamma_s, (\mathcal{S}_1, Th_1) \vdash \texttt{arch-unit-ref-type UNIT-SPEC ARCH-SPEC} \ggg (U\Sigma_1, U\Sigma_2), (\mathcal{S}_3, Th)}$$

4.4 Refinement bodies

$$\boxed{R\Sigma, \Gamma_s, (\mathcal{S}, Th) \vdash \texttt{REF-BODY} \ggg R\Sigma, (\mathcal{S}', Th')}$$

$\Gamma_s, (\mathcal{S}, Th)$ is a verification static global environment. $(\mathcal{S}', Th')$ is a development graph extending $(\mathcal{S}, Th)$. $R\Sigma$ is a verification refinement signature.

$$\frac{\begin{array}{c} U\Sigma_s = N_1, \ldots, N_n \to N_s \\ U\Sigma_t = N_1, \ldots, N_n \to N_t \\ \vdash \texttt{SYMB-MAP-ITEMS} \rhd\!\!\!\rhd\!\!\!\rhd r \\ \sigma = r|_{N_t}^{N_s} \\ Th' = Th \cup \{\ N_s \overset{\sigma}{=\!=\!\Rightarrow} N_t\ \} \end{array}}{(U\Sigma_s, U\Sigma_t), \Gamma_s, (\mathcal{S}, Th) \vdash \texttt{simple-mor SYMB-MAP-ITEMS} \rhd\!\!\!\rhd\!\!\!\rhd (U\Sigma_s, U\Sigma_t), (\mathcal{S}, Th')}$$

$$\frac{\begin{array}{c} \Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s) \\ (\texttt{RN} \mapsto (U\Sigma_s, U\Sigma_t)) \in \mathcal{R}_s \end{array}}{(U\Sigma_s, U\Sigma_t), \Gamma_s, (\mathcal{S}, Th) \vdash \texttt{named-ref RN} \rhd\!\!\!\rhd\!\!\!\rhd (U\Sigma_s, U\Sigma_t), (\mathcal{S}, Th)}$$

Should semantics for FIT-VIEWs be split in order to avoid these repetitions?

Concerning views as refinement bodies, we adapt the rules for the semantics of FIT-VIEWs from Sect. (((sem-sem-sec-FittingViews))). First we study the situation of a non-generic view.

$$\frac{\begin{array}{c} U\Sigma_s = N_1, \ldots, N_n \to N_s \\ U\Sigma_t = N'_1, \ldots, N'_n \to N_t \\ \Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s) \\ (\texttt{VN} \mapsto (N'_s, \sigma, (\emptyset, (), N'_t))) \in \mathcal{V}_s \\ \Sigma_{N_s} = \Sigma_{N'_s} \quad \Sigma_{N_t} = \Sigma_{N'_t} \\ Th' = Th \cup \{\ N_s \overset{id}{=\!=\!\Rightarrow} N'_t\ ;\ \ N'_t \overset{id}{=\!=\!\Rightarrow} N_t\ \} \end{array}}{(U\Sigma_s, U\Sigma_t), \Gamma_s, (\mathcal{S}, Th) \vdash \texttt{named-view VN} \rhd\!\!\!\rhd\!\!\!\rhd (U\Sigma_s, U\Sigma_t), (\mathcal{S}, Th')}$$

Now we come to the case of generic views. [Omitted here — let's first clarify overall issues.]

$$\frac{\begin{array}{c} \Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s, \mathcal{R}_s) \\ (\texttt{RN}_1 \mapsto (U\Sigma_1, U\Sigma_2)) \in \mathcal{R}_s \\ (\texttt{RN}_2 \mapsto (U\Sigma_3, U\Sigma_4)) \in \mathcal{R}_s \\ U\Sigma_1 = N_1^1, \ldots, N_n^1 \to N^1 \\ U\Sigma_4 = N_1^4, \ldots, N_n^4 \to N^4 \\ \Sigma^{N^1} = \Sigma^{N^4} \\ \Sigma^{N_i^1} = \Sigma^{N_i^4}\ (i = 1, \ldots, n) \\ Th' = Th \cup \{\ N^1 \overset{id}{=\!=\!\Rightarrow} N^4\ \} \cup \{\ N_i^4 \overset{id}{=\!=\!\Rightarrow} N_i^1\ \ |\ i = 1, \ldots, n\} \end{array}}{(U\Sigma_2, U\Sigma_3), \Gamma_s, (\mathcal{S}, Th) \vdash \texttt{compose RN}_1\ \texttt{RN}_2 \rhd\!\!\!\rhd\!\!\!\rhd (U\Sigma_2, U\Sigma_3), (\mathcal{S}, Th')}$$

4.5 Unit refinements

$$\boxed{A\Sigma, \Gamma_s, (\mathcal{S}, Th) \vdash \texttt{UNIT-REFINEMENT} \rhd\!\!\!\rhd\!\!\!\rhd (UN, R\Sigma)}$$

$A\Sigma$ is an architectural signature. UN is a unit name. $R\Sigma$ is a verification refinement signature.

$$\frac{\begin{array}{c}\mathtt{UN} \in Dom(C_s) \\ C_s(\mathtt{UN}) = N_s \\ \emptyset, \Gamma_s, (\mathcal{S}_1, Th_1) \vdash \mathtt{SPEC} \ggg N_t, (\mathcal{S}_2, Th_2) \\ (N_s, N_t), \Gamma_s, (\mathcal{S}_2, Th_2) \vdash \mathtt{RB} \ggg (N_s, N_t), (\mathcal{S}_3, Th_3)\end{array}}{(C_s, U\Sigma), \Gamma_s, (\mathcal{S}_1, Th_1) \vdash \mathtt{simple\text{-}unit\text{-}ref}\ \mathtt{UN}\ \mathtt{SPEC}\ \mathtt{RB} \ggg (\mathtt{UN}, (N_s, N_t)), (\mathcal{S}_3, Th_3)}$$

$$\frac{\begin{array}{c}\mathtt{UN} \in Dom(C_s) \\ C_s(\mathtt{UN}) = U\Sigma_s \\ \Gamma_s, (\mathcal{S}_1, Th_1) \vdash \mathtt{ARCH\text{-}SPEC} \ggg (C_s, U\Sigma_t), (\mathcal{S}_2, Th_2) \\ (U\Sigma_s, U\Sigma_t), \Gamma_s, (\mathcal{S}_2, Th_2) \vdash \mathtt{RB} \ggg (U\Sigma_s, U\Sigma_t), (\mathcal{S}_3, Th_3)\end{array}}{(C_s, U\Sigma), \Gamma_s, (\mathcal{S}_1, Th_1) \vdash \mathtt{arch\text{-}unit\text{-}ref}\ \mathtt{UN}\ \mathtt{ARCH\text{-}SPEC}\ \mathtt{RB} \ggg (\mathtt{UN}, (U\Sigma_s, U\Sigma_t)), (\mathcal{S}_3, Th_3)}$$

$$\frac{\begin{array}{c}\mathtt{UN} \in Dom(C_s) \\ C_s(\mathtt{UN}) = U\Sigma_s \\ \Gamma_s, (\mathcal{S}_1, Th_1) \vdash \mathtt{UNIT\text{-}SPEC} \ggg U\Sigma_t, (\mathcal{S}_2, Th_2) \\ (U\Sigma_s, N_t), \Gamma_s, (\mathcal{S}_2, Th_2) \vdash \mathtt{RB} \ggg (U\Sigma_s, U\Sigma_t), (\mathcal{S}_3, Th_3)\end{array}}{(C_s, U\Sigma), \Gamma_s, (\mathcal{S}_1, Th_1) \vdash \mathtt{unit\text{-}unit\text{-}ref}\ \mathtt{UN}\ \mathtt{UNIT\text{-}SPEC}\ \mathtt{RB} \ggg (\mathtt{UN}, (U\Sigma_s, U\Sigma_t)), (\mathcal{S}_3, Th_3)}$$

5

Examples

The steam boiler example

We now formalize the refinement steps in the steam-boiler control systemexample of the CASL User Manual [BM03], Chap. 13. We repeat the specifications showing the architecture of the system given in [BM03, 13.10]. All the involved structured specifications are omitted here and should be looked up in [BM03, 13].

[BM03, 13.10] starts with the following rather obvious architecture for the steam-boiler control system:

arch spec ARCH_SBCSname]Arch_Sbcs@ARCH_SBCS =
units P : VALUE → PRELIMINARY;
S : PRELIMINARY → SBCS_STATE;
A : SBCS_STATE → SBCS_ANALYSIS;
C : SBCS_ANALYSIS → STEAM_BOILER_CONTROL_SYSTEM
result $\lambda\, V$: VALUE • $C\,[A\,[S\,[P\,[V]]]]$
end

In a next step, the specification VALUE → PRELIMINARY of the component P is refined into the following architectural specification.

arch spec ARCH_PRELIMINARYname]Arch_Preliminary@ARCH_PRELIMINARY
=
units *SET* : **{ sort** *Elem***}** × NAT → SET [**sort** *Elem*];
B : BASICS;
MS : MESSAGES_SENT **given** B;
MR : VALUE → MESSAGES_RECEIVED **given** B;
CST : VALUE → SBCS_CONSTANTS
result $\lambda\, V$: VALUE • *SET* [*MS* **fit** *Elem* ↦ *S_Message*] [V]
and *SET* [*MR* [V] **fit** *Elem* ↦ *R_Message*] [V]
and *CST* [V]
end

unit spec UNIT_SBCS_STATEname]Unit_Sbcs_State@UNIT_SBCS_STATE =
PRELIMINARY → SBCS_STATE_IMPL

Summing up, this leads to the following architectural refinement:

refinement R1 : ARCH_SBCS **to**
units *P* **to** ARCH_PRELIMINARY;
S **to** UNIT_SBCS_STATE;
A **to** ARCH_ANALYSIS;
C **to** UNIT_SBCS_SYSTEM
end

Note that S is refined to a monomorphic unit specification — the development is finished at this point. A similar remark holds for the component C; however, the needed unit specification UNIT_SBCS_SYSTEM is not provided in [BM03].

The specification SBCS_STATE → SBCS_ANALYSIS of the component A of ARCH_SBCS can be refined into the following architectural specification:

arch spec ARCH_ANALYSISname]Arch_Analysis@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*]]]]
end

The specification of the components ME and MTS are simple enough to be directly implemented. The specifications of the components FD and PR can be refined as follows.

arch spec ARCH_FAILURE_DETECTIONname]Arch_Failure_Detection@ARCH_FAILURE_DETECTION =
units *MTSF* : SBCS_STATE
→ MESSAGE_TRANSMISSION_SYSTEM_FAILURE;
PF : SBCS_STATE → PUMP_FAILURE;
PCF : SBCS_STATE → PUMP_CONTROLLER_FAILURE;
SF : SBCS_STATE → STEAM_FAILURE;
LF : SBCS_STATE → LEVEL_FAILURE;
PU : MESSAGE_TRANSMISSION_SYSTEM_FAILURE
× PUMP_FAILURE × PUMP_CONTROLLER_FAILURE
× STEAM_FAILURE × LEVEL_FAILURE
→ FAILURE_DETECTION
result λS : SBCS_STATE •
PU [*MTSF*[*S*]] [*PF*[*S*]] [*PCF*[*S*]] [*SF*[*S*]] [*LF*[*S*]]
hide *Pump_OK*, *Pump_Controller_OK*, *Steam_OK*, *Level_OK*
end

Finally the specification FAILURE_DETECTION → PU_PREDICTION of the component PR of the architectural specification ARCH_ANALYSIS is refined as follows:

arch spec ARCH_PREDICTIONname]Arch_Prediction@ARCH_PREDICTION =
units *SE* : FAILURE_DETECTION →
STATUS_EVOLUTION [FAILURE_DETECTION];
SLP : FAILURE_DETECTION → STEAM_AND_LEVEL_PREDICTION;
PP : STATUS_EVOLUTION [FAILURE_DETECTION]
× STEAM_AND_LEVEL_PREDICTION
→ PUMP_STATE_PREDICTION;
PCP : STATUS_EVOLUTION [FAILURE_DETECTION]
× STEAM_AND_LEVEL_PREDICTION
→ PUMP_CONTROLLER_STATE_PREDICTION
result λ *FD* : FAILURE_DETECTION •
local *SEFD* = *SE* [*FD*]; *SLPFD* = *SLP* [*FD*] **within**
PP [*SEFD*] [*SLPFD*] **and** *PCP* [*SEFD*] [*SLPFD*]
end

This is summed up in the following refinement:

refinement R2 : ARCH_ANALYSIS **to**
units *FD* **to** ARCH_FAILURE_DETECTION;
PR **to** ARCH_PREDICTION;
ME **to** UNIT_MODE_EVOLUTION;
MTS **to** UNIT_SBCS_ANALYSIS
end

In order to reach a complete refinement tree, it now remains to provide monomorphic unit specifications UNIT_SBCS_SYSTEM, UNIT_MODE_EVOLUTION and UNIT_SBCS_ANALYSIS, and obvious architectural refinements of ARCH_FAILURE_DETECTION and ARCH_PREDICTION (and the monomorphic unit specifications needed for implementing these).

Some simple refinements

spec NAT = **free type** *Nat* ::= 0 | *suc*(*Nat*) **end**
spec NATBIN =
 free **type** *NatBin* ::= 0 | __0(*NatBin*) | __1(*NatBin*)
 op *suc*(*n* : *NatBin*) : *NatBin* = ...
end
refinement R3 : NAT **to** NATBIN =
 $Nat \mapsto NatBin$
end

spec NAT = **free type** *Nat* ::= 0 | *suc*(*Nat*) **end**
spec NATBYTE =
 free **types** *Byte* ::= 0 | 1 | ... | 255
  NatByte ::= 0 | __ ::: __(*Byte*; *NatByte*)
 op *suc*(*n* : *NatByte*) : *NatByte* = ...

endview V : NAT **to** NATBYTE =
 $Nat \mapsto NatByte$
refinement R4 : NAT **to** NATBIN = V
end

Composition of refinements

from BASIC/STRUCTUREDDATATYPES **get** LIST
spec BINLISTname]BinList@BINLIST =
 free **type** *Bin* ::= 0 | 1
then
 List[**sort** *Bin*]
then **ops** *add0*(*l* : *List*[*Bin*]) : *List*[*Bin*] = 0 :: *l*;
  add1(*l* : *List*[*Bin*]) : *List*[*Bin*] = 1 :: *l*
end

refinement R5 : NATBIN **to** BINLIST =
 $NatBin \mapsto List[Bin], 0 \mapsto [], \_\_0 \mapsto add0, \_\_1 \mapsto add1$
end

refinement R6 : NAT **to** BINLIST = R3 **then** R5
end

An architectural refinement

%{ The following example illustrates the difference between
the structure of specifications and the architectural specification of structure. **}%**

spec NUMname]Num@NUM =
  sort *Num*
  ops $0 \quad : Num;$
  $succ : Num \to Num$
end

spec NUM_MONOIDname]Num_Monoid@NUM_MONOID =
  MONOID **with** $Elem \mapsto Num, n \mapsto 0, \_\_ * \_\_ \mapsto \_\_ + \_\_$

spec ADD_NUMname]Add_Num@ADD_NUM =
  NUM **and** NUM_MONOID
then $\forall x, y : Num \bullet x + succ(y) = succ(x + y)$
end

spec ADD_NUM_EFFICIENTLYname]Add_Num_Efficiently@ADD_NUM_EFFICIENTLY
  =
  generated type $Bin ::= 0 \mid 1 \mid \_\_0(Bin) \mid \_\_1(Bin)$
  ops $\_\_ + \_\_ , \_\_ ++ \_\_ : Bin \times Bin \to Bin$
  %{ $\_\_ + \_\_$ is binary addition; $\_\_ ++ \_\_$ is binary addition with carry. **}%**
  $\forall x, y : Bin$
  - $0\ 0 = 0$
  - $0\ 1 = 1$
  - $x\ 0 + y\ 0 = (x + y)\ 0$
  - $x\ 0 ++ y\ 0 = (x + y)\ 1$
  - $x\ 0 + y\ 1 = (x + y)\ 1$
  - $x\ 0 ++ y\ 1 = (x ++ y)\ 0$
  - $x\ 1 + y\ 0 = (x + y)\ 1$
  - $x\ 1 ++ y\ 0 = (x ++ y)\ 0$
  - $x\ 1 + y\ 1 = (x ++ y)\ 0$
  - $x\ 1 ++ y\ 1 = (x ++ y)\ 1$
end
  %{ It is more efficient to implement successor in terms of (binary) addition, while it is easier to specify addition in terms of successor than in terms of binary addition. Thus, the structure of the implementation differs from the structure of the specification: **}%**

arch spec EFFICIENT_ADD_NUMname]Efficient_Add_Num@EFFICIENT_ADD_
  NUM =
units N : ADD_NUM_EFFICIENTLY;
  M : **{ op** $succ(n : Bin) : Bin = n + 1$ **} given** N
result
  M **hide** $1, \_\_0, \_\_1, \_\_ ++ \_\_$
end

%% We have now that EFFICIENT_ADD_NUM is a refinement of ADD_NUM.
refinement R7 : ADD_NUM **to** EFFICIENT_ADD_NUM =
  $Num \mapsto Bin$
end

Refining one specification in two directions

from BASIC/STRUCTUREDDATATYPES **get** LIST, SET
arch spec NATLISTname]NatList@NATLIST =
units N : NAT;
L : LIST[NAT] **given** N
result L
end

arch spec NATSETname]NatSet@NATSET =
units N : NAT;
S : SET[NAT] **given** N
result S
end

refinement R8 : NATLIST
units N **to** NATBIN;
L **to** ELEM $\rightarrow$ LIST[ELEM]
end

refinement R9 : NATLIST
units N **to** NATBYTE;
S **to** ELEM $\rightarrow$ SET[ELEM]
end

The example shows that refinement trees cannot always be built automatically from the specifications in a library.

Perhaps we should also allow partial refinements of architectural specifications that only refine some of the units, while the remaining units are considered to be determined by their monomorphic unit specifications?

Stacks implemented as arrays with pointer

This famous problem can be solved with simple refinement as follows. (With behavioural refinement, one would not need to specify th equality on $StackAsArray[Elem]$ explicitly.)

spec STACKname]Stack@STACK[ELEM] =
free type $Stack[Elem] ::= empty \mid push(Elem;\ Stack[Elem])$
op $pop : Stack[Elem] \rightarrow? Stack[Elem]$
$\forall$ $x : Elem;\ s : Stack[Elem]$
• $\neg\, def\ pop(empty)$
$pop(push(x, s)) = s$
end

spec ARRAYname]Array@ARRAY[ELEM] =

```
        Nat
then    generated type Array[Elem] ::= init | __!__ = __(Array[Elem]; Nat; Elem)
        op      __!__ : Array[Elem] × Nat →? Elem
        ∀       a : Array[Elem]; x : Elem; m, n : Nat
        •       ¬ def init!n
        •       (a!n := x)!n = x
        •       (a!m := x)!n = a!n if ¬m = n
        •       a1 = a2 ⇔ (∀n : Nat.a1!n = a2!n)
end

spec    StackAsArrayname]StackAsArray@StackAsArray[Elem] = %mono
        Array[Elem]
then    generated type StackAsArray[Elem] ::= __@__(Array[Elem]; Nat)
        ∀       a1, a2 : Array[Elem]; n1, n2 : Nat
        •       a1@n1 = a2@n2 ⇔
                    (n1 = n2 ∧ ∀i : Nat • i < n1 ⇒ a1!i = a2!i)
        ops     empty : StackAsArray[Elem];
                push  : StackAsArray[Elem] × Elem → StackAsArray[Elem];
                pop   : StackAsArray[Elem] →? StackAsArray[Elem]
        ∀       a : Array[Elem]; x : Elem; n : Nat
        •       empty = init@0
                push(a@n, x) = a!n := x@succ(n)
                ¬ def pop(a@0)
                pop(a@succ(n)) = a@n
end

unit spec UStackname]UStack@UStack = Elem → Stack [Elem]

unit spec UArrayname]UArray@UArray = Elem → Array [Elem]

arch spec ArchStackAsArrayname]ArchStackAsArray@ArchStackAsArray
        =
units   A  : UArray;
        AS : Array [Elem] → StackAsArray [Elem]
result  λ X : Elem • AS [A [X]]
end

refinement R10 : UStack to ArchStackAsArray =
        Stack ↦ StackAsArray
end
```

References

AKKB99. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner. *Algebraic Foundations of Systems Specification*. Springer, 1999.

BM03. Michel Bidoit and Peter D. Mosses. *CASL User Manual*. Lecture Notes in Computer Science. Springer, 2003. To appear.

BST02. Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273, 2002.

CoF. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.brics.dk/Projects/CoFI/`.

ST87. Donald Sannella and Andrzej Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.

ST88. D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica*, 25:233–281, 1988.