Specification refinements: calculi, tools, and applications

Citation for published version:
Codescu, M, Mossakowski, T, Sannella, D & Tarlecki, A 2017, 'Specification refinements: calculi, tools, and applications' Science of Computer Programming, vol. 144, pp. 1-49. DOI: 10.1016/j.scico.2017.04.005

Digital Object Identifier (DOI):
10.1016/j.scico.2017.04.005

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Science of Computer Programming

Specification refinements: calculi, tools, and applications

Mihai Codescu[a], Till Mossakowski[b], Donald Sannella[c], Andrzej Tarlecki[d]

[a] *Free University of Bozen-Bolzano, Italy*
[b] *Faculty of Computer Science, Otto-von-Guericke University of Magdeburg, Germany*
[c] *Laboratory for Foundations of Computer Science, University of Edinburgh, United Kingdom*
[d] *Institute of Informatics, University of Warsaw, Poland*

Abstract

We propose and study a framework for systematic development of software systems (or models) from their formal specifications. We introduce a language for formal development by refinement and decomposition, as an extension to CASL. We complement it with a notion of refinement tree and present proof calculi for checking correctness of refinements as well as their consistency. Both calculi have been implemented in the Heterogeneous Tool Set (HETS), and have been integrated with other tools like model finders and conservativity checkers.

Keywords: algebraic specifications, refinement, architectural specifications, consistency

---

Contents

Email addresses: `Mihai.Codescu@unibz.it` (Mihai Codescu), `till@iks.cs.ovgu.de` (Till Mossakowski), `dts@inf.ed.ac.uk` (Donald Sannella), `tarlecki@mimuw.edu.pl` (Andrzej Tarlecki)

1. Introduction

The standard development paradigm of algebraic specification [2, 3] postulates that development begins with a formal **requirements specification**, capturing a software project's informal requirements, that fixes only expected properties of the system to be built but ideally says nothing about implementation issues. This is followed by a number of **refinement** steps [4] that fix more and more details of the design, until a specification is obtained that is detailed enough that its conversion into a program is relatively straightforward.

Consider the task of providing an implementation (or finding a model) for a specification *SP*. The classical theory of refinement [2, 3] provides the means for decomposing this task into a sequence of refinement steps:

$$SP \rightsquigarrow SP_1 \rightsquigarrow \ldots \rightsquigarrow SP_n \rightsquigarrow P$$

Here, $SP_1, \ldots, SP_n$ are intermediate specifications and P is a final specification that directly yields an implementation or a model description.

Actually, this picture is too simple in practice: for complex software systems, it is necessary to reduce complexity by introducing branching points in the chain of refinement steps, such that a specification is decomposed into smaller components. The components will then themselves be further refined, possibly independently, e.g. by different developers, and possibly by further decomposing their specifications. CASL architectural specifications [5, 6] have been designed for this purpose, based on the insight that structuring of implementations is different from structuring of specifications [7, 8]. However, CASL architectural specifications only specify branching points in the chain of refinements, and not refinements themselves. In this work, we extend CASL with a **refinement language** that adds the means to formalize whole developments in the form of **refinement trees**, like the one in the diagram below. Using our language, developments such as the one represented in this tree can be formally specified and verified for correctness.

$$SP \rightsquigarrow \begin{cases} SP_1 \rightsquigarrow P_1 \\ \vdots \\ SP_n \rightsquigarrow \begin{cases} SP_{n1} \rightsquigarrow \begin{cases} SP_{n11} \rightsquigarrow P_{n11} \end{cases} \\ \cdots \\ SP_{nm} \rightsquigarrow P_{nm} \end{cases} \end{cases}$$

This approach is not only applicable for formal software development, where the leaves of the tree are programs, but also for finding models of larger logical theories. We propose using refinement trees as a way for managing this task. The leaves of the trees are then small theories whose consistency can be proved using an automated model finder. Once models for these are given, a model for the entire theory may be constructed using the refinement tree. One contribution of our work is that the refinement trees make informal pictures such as the one above formal on the basis of the refinement language.

We design a language for expressing refinements and branching of formal developments, which is detailed in Sect. 2. It extends CASL architectural specifications for expressing branching as in [9, 6]. We illustrate the language using an example application from [9], based on a challenge problem from [10]. The semantics of this language is based on the notion of institution, which is recalled in Sect. 3. Section 4 introduces the semantics of refinements and their parts. It contains the central contribution of the paper, together with section 5 that introduces various proof calculi. In Sect. 5.1, we contribute a new constructive proof calculus for architectural specifications, covering the whole language for the first time. Section 5.2 provides treatment of unit imports, a known source of increase in complexity of verification of architectural specifications. Section 5.3 gives a proof calculus for refinements. This calculus rephrases the proof calculus for architectural specifications and extends it to a more general setting. Completeness of this calculus is proved in Sect. 5.4. Section 5.5 provides a calculus for consistency of refinements. Section 6 introduces refinement trees. Moreover, we describe the implementation of the calculus in the Heterogeneous Tool Set Hets [11, 12] which also integrates model finders and other tools. Section 7 deals with the leaves of refinement trees and how to refine them into real programs. Section 8 concludes the paper.

This work builds on and considerably extends [13] and [14]; in particular, we cover both shared subcomponents and refinements of unit specifications with imports as mentioned in the conclusion of [13] and we study completeness of the proof calculus for refinements, mentioned in [14] as future work. Its contribution to the Casl development framework can be summarized as follows:

- a new proof calculus for architectural specifications (CASL already had one). The novelty of the new proof calculus is that it is constructive (i.e. it computes a specification instead of merely checking correctness) and that it covers the whole language in contrast to the original proof calculus (reducing the difficult case of units with imports to use of a simpler construction);
- a refinement language and its semantics;
- proof calculi for correctness and consistency of refinements;
- soundness and completeness results for the proof calculi;
- a notion of refinement trees;
- implementation of refinement trees and proof calculi in Hets.

2. Refinements in Casl

We will introduce a language for refinements that extends the Casl specification language, which we recall first.

2.1. A Brief Summary of Institution-Independent Specifications in Casl

The Common Algebraic Specification Language Casl [6] has been designed by the *Common Framework Initiative for Algebraic Specification and Development* [1] with the goal to unify the many previous algebraic specification languages and to provide a standard language for the specification and development of modular software systems. Casl has been designed to consist of orthogonal layers:

1. **basic specifications** provide means for writing relatively small, "flat" specifications;
2. **structured specifications** allow large specifications to be organized in a modular way;
3. **architectural specifications** [5] describe, in contrast to the previous layer, the structure of the *implementation* of a software system;
4. **libraries of specifications** allow storage and retrieval of named specifications.

The orthogonality of the layers means that the syntax and semantics of each layer are independent of those of the others. In particular, this allows one to replace the logic used in the layer of Casl basic specifications with a completely different logic without having to modify the other layers. This is achieved in a mathematically sound way by using institutions [15] (see Sect. 3 below), which formalize a model-oriented view of the notion of logical system. Institutions abstract from the details of sentences (logical formulas) and models, and the notion of satisfaction of a sentence in a model. All this is indexed by signatures, i.e. by the user-defined vocabularies of non-logical (domain-specific) symbols.

Institutions also feature signature morphisms, which capture the concepts of change of notation and enlargement of context. They play an important role for constructing structured and architectural specifications as well as refinements.

```
SPEC ::= (Σ, E) | SPEC and SPEC | SPEC with σ | SPEC hide σ | SPEC-NAME
```

Figure 1: CASL structured specifications (generic specifications omitted).

```
LIB-DEFN ::= lib-defn LIB-NAME LIB-ITEM*
LIB-ITEM ::= SPEC-DEFN | VIEW-DEFN | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
SPEC-DEFN ::= spec SPEC-NAME = SPEC
VIEW-DEFN ::= view VIEW-NAME : SPEC-NAME to SPEC-NAME = σ
ARCH-SPEC-DEFN ::= arch spec ARCH-SPEC-NAME = ASP
UNIT-SPEC-DEFN ::= unit spec UNIT-SPEC-NAME = USP
```

Figure 2: CASL Libraries.

Signature morphisms map signatures in a way that allows sentences over the source signature to be translated to sentences over the target signature. Thus, if φ is a Σ_1-sentence and $\sigma : \Sigma_1 \to \Sigma_2$ is a signature morphism, then $\sigma(\varphi)$ is a Σ_2-sentence. Moreover, if M is a Σ_2-model, then it can be reduced to a Σ_1-model $M|_\sigma$, called the *σ-reduct* of M. Satisfaction of sentences in models is subject to the satisfaction condition, which captures the idea that truth is invariant under change of notation and enlargement of context.

The simplest form of specifications are basic specifications. A basic specification (Σ, E) essentially consists of a signature Σ plus some set E of sentences over this signature, in a given institution. Working with basic specifications is only suitable for specifications of fairly small size. For practical situations, when large systems are to be specified, they would become very difficult to understand and use effectively. We may also want to re-use parts of specifications. Therefore, algebraic specification languages like CASL provide support for structuring specifications. Syntax and semantics of these structured specifications can be formulated over an arbitrary but fixed institution. (The same holds for architectural specifications, refinements and specification libraries.)

In Fig. 1, we present the syntax we use for a number of specification-building operations over an arbitrary institution. They are the subset of CASL structuring mechanisms that are used in our examples. Two specification can be combined using **and**, resulting in a conjunctive combination of the constraints expressed by the individual specifications. A specification SPEC can be renamed along a signature morphism σ using SPEC **with** σ, resulting in a specification over the target signature of the signature morphism. Finally, a specification SPEC can be restricted to an export interface using SPEC **hide** σ, where σ is the inclusion of the export interface signature into that of SPEC.

A semantics of structured specifications will be given in Sect. 4. For now, let us just note that a structured specification SP denotes a signature (written $\mathbf{Sig}(SP)$) and a class of models over that signature. The exact nature of syntax and semantics of structured specifications may vary and is not essential for the study of architectural specifications and refinements.

The simplest form of refinement is model class inclusion between structured specifications, possibly with a change of signature: if SP_1, SP_2 are structured specifications and $\sigma : \mathbf{Sig}(SP_1) \to \mathbf{Sig}(SP_2)$ is a signature morphism, we say that SP_1 *refines along σ to* SP_2, denoted $SP_1 \overset{\sigma}{\rightsquigarrow} SP_2$, if for each model of SP_2, its reduct along σ is a model of SP_1. If σ is the identity, it can be omitted. CASL already allows such refinements to be postulated using *views*: if SP_1 and SP_2 are structured specifications, then **view** $V : SP_1$ **to** $SP_2 = \sigma$ requires each model of SP_2 reduced along σ to be a model of SP_1. If this holds, then $\sigma : SP_1 \to SP_2$ is called a *specification morphism.* Two specifications are equivalent if they have the same signatures and the same model classes.

A CASL document consists of a library definition. A library definition is a list of definitions of named specifications and views. The names of the specifications and views must be distinct and visibility is linear. Fig. 2 gives the grammar of library definitions. Architectural specifications (ASP) and unit specifications (USP) will be covered in the next sections. The Distributed Ontology, Model and Specification Language (DOL), a standard of the Object Management Group (OMG) [16][1], uses a similar syntax to the one in Figs. 1 and 2.

[1]See http://www.omg.org/spec/DOL/ and http://www.dol-omg.org.

2.2. *Architectural Specifications: Motivation*

Architectural specifications in CASL [5] have been introduced as means of providing structure for the implementation of a software system. Each architectural specification names a number of components and gives a linking procedure which describes how to combine implementations of these components to obtain an implementation of the overall system. (By contrast, the models of structured specifications are monolithic and have no more structure than models of basic specifications.) The internals of each component are not available other than via the specification of the component. This means that components can be implemented independently of each other, and we can replace the implementation of a component with a new one without having to modify other implementations — all we need to do is to re-link the implementations of the components as described by the linking procedure of the architectural specification.

To put this more formally, in Fig. 3:

- SP is the initial specification (say, of some system),
- $UN_1, \dots UN_n$ are units (i.e. named models) with their specifications $USP_1, \dots, USP_n$,
- κ is the linking procedure involving the units, i.e. a function taking models A_i of USP_i, $i = 1, \dots, n$ to a model $\kappa(A_1, \dots, A_n)$ over the signature of SP, and
- the refinement relation is denoted $\rightsquigarrow$, expressing that the model given by $\kappa(A_1, \dots, A_n)$ is indeed a model of SP.

$$\boxed{SP \quad \rightsquigarrow \quad \kappa \begin{cases} UN_1 : USP_1 \\ \quad\vdots \\ UN_n : USP_n \end{cases}}$$

Figure 3: Refinement introducing branching via a linking procedure κ.

Intuitively, the refinement is correct if for any models A_i of USP_i, $i = 1, \dots, n$, $\kappa(A_1, \dots, A_n)$ is a model of SP. When this is indeed the case, then we can replace a model A_j of USP_j with another model A'_j of USP_j and re-link the models $A_1, \dots, A'_j, \dots A_n$ as prescribed by κ to get a realization of SP, without having to modify the models $A_1, \dots, A_{j-1}, A_{j+1}, \dots, A_n$ in any way.

To a software engineer, CASL architectural specifications appear to be similar to UML component diagrams: they share the goal of decomposing the task of implementing a software system into smaller subtasks, that are developed independently and can be flexibly redeployed. The specification of a unit in an architectural specification defines the behavior of the unit, similarly as required and provided interfaces do for components in UML component diagrams.

2.3. *CASL Architectural Specifications*

A definition of a named CASL architectural specification corresponding to the right-hand side of the refinement in in Fig. 3 is written:

arch spec ASP_NAME =
  units
    $UN_1 : USP_1$;
    ...
    $UN_n : USP_n$;
  result UE

where UE is a *unit expression* describing the linking procedure κ and possibly involving the units $UN_1, \dots, UN_n$. Units are combined in unit expressions with operations like renaming, hiding, and amalgamation, see Fig. 4. We require that common symbols of unit expressions must be interpreted in the same way. Architectural specifications are defined for an arbitrary institution, and are thus independent of the underlying formalism used for basic specifications, as well as of that used for structured specifications. Thus, unit specifications USP of (non-generic) units are just structured specifications (of models, where models provide the semantics of units).

The picture is slightly more general: units may be *generic*, in which case they correspond to partial functions taking (tuples of) models to models. The result model is required to preserve the parameter models (*persistency*), with the intuition that the implementation of the parameters must be kept and not be re-implemented, and the function

```
ASP ::= ARCH-SPEC-NAME | units UDD_1; ... ; UDD_n⟨;⟩ result UE
UDD ::= UDEFN | UDECL
UDECL ::= A : USP ⟨given UT_1, ... , UT_n⟩
USP ::= SPEC | SPEC_1 × ··· × SPEC_n → SPEC | arch spec ASP
UDEFN ::= A = UE
UE ::= UT | λ A_1 : SPEC_1, ... , A_n : SPEC_n • UT
UT ::= A | A [FIT_1] ... [FIT_n] | UT and UT | UT with σ : Σ → Σ′ |
        UT hide σ : Σ → Σ′ | local UDEFN_1 ... UDEFN_n within UT
FIT ::= UT | UT fit σ : Σ → Σ′
```

Figure 4: Syntax of CASL architectural specifications.

is only defined on *compatible* models, meaning that the implementation of the parameters must coincide on their common symbols. Unit expressions may also involve applications of generic units.

Units can be specified with *unit specifications*. A unit specification USP is either an ordinary structured specification $USP = SP$ of a non-generic unit, or the specification $USP = SP_1 \times \cdots \times SP_n \rightarrow SP$ of a generic unit, with a list of parameter specifications and a result specification. A generic unit satisfies such a unit specification if it takes as arguments models of the parameter specifications $SP_1 \times \cdots \times SP_n$ and returns a model of the result specification SP. A unit specification can be named by writing

unit spec USP_NAME = *USP*

Figure 4 presents the complete syntax of architectural specifications in CASL. Here, $A, A_1, \ldots, A_n$ stand for component names, Σ and Σ' denote signatures and σ denotes a signature morphism. The syntax can be briefly explained as follows: an architectural specification `ASP` consists of a list of unit declarations `UDECL`, with an optional list of imported units (marked as optional with $\langle \ldots \rangle$) for each unit declaration, and unit definitions `UDEFN`—where declarations assign unit specifications `USP` to unit names and definitions assign unit expressions `UE` to unit names—and a result unit expression formed with the units declared and defined. Unit declarations and definitions are separated by semicolons, with the last separator before the keyword **result** being optional. Unit expressions are used to give definitions for generic units, while unit terms define non-generic units. When the result unit of an architectural specification is generic, the system is "open", requiring some parameters to provide an implementation. In the case of fitting arguments, an omitted signature morphism σ is assumed to be the identity.

In the following sections, we restrict CASL architectural specifications by omitting unit imports. They will be discussed separately in Sect. 5.2.

Example 2.1. The task of providing an implementation of the integer, string and boolean datatypes can be decomposed into implementing each of the datatypes separately and then putting together the obtained realizations:

arch spec TYPES =
  units N : INT;
    S : STRING;
    B : BOOL;
    R = N **and** S **and** B
  result R

The architectural specification TYPES declares three units, N, S and B, one for each of the integer, string and boolean datatype, and then defines a unit as the union of N, S and B. The union is a unit term. If the implementation of another datatype, say floats, should be provided from outside the architectural specification TYPES, we could make use of a unit expression: R = **lambda** X: FLOAT . N **and** S **and** B **and** X. □

Example 2.2. We will illustrate the CASL architectural and refinement languages using an industrial case study: the specification of a steam boiler control system for controlling the water level in a steam boiler. The problem has been formulated in [10] as a benchmark for specification languages; [9] gives a complete solution using CASL, including architectural design and refinement of components. However, the refinement steps were presented there in an informal way. The refinement language introduced in Sect. 2.4 below makes it possible to formally write down the refinement steps using CASL refinements.

The specifications involved can be briefly explained as follows.[2] Components of the system communicate with each other using messages, some of which include values, such as the identifier of a pump, the water level, or the output of steam. The specification VALUE provides a very abstract notion of these values, and only assumes that values extend natural numbers and are equipped with some loosely specified operations and predicates. This specification acts as a parameter of the entire design. PRELIMINARY gathers the messages in the system, both sent and received, and also defines a series of constants characterizing the steam boiler. SBCS_STATE introduces observers for the system states, while SBCS_ANALYSIS extends this to an analysis of the messages received, failure detection and computation of messages to be sent. Finally, STEAM_BOILER_CONTROL_SYSTEM specifies the initial state and the reachability relation between states. We record the requirement that the system is open in the models for VALUE using the unit specification:

unit spec SBCS_OPEN = VALUE → STEAM_BOILER_CONTROL_SYSTEM

The initial design for the architecture of the system is recorded in the following architectural specification:

```
arch spec ARCH_SBCS =
    units
          P : VALUE → PRELIMINARY;
          S : PRELIMINARY → SBCS_STATE;
          A : SBCS_STATE → SBCS_ANALYSIS;
          C : SBCS_ANALYSIS → STEAM_BOILER_CONTROL_SYSTEM
    result λ V : VALUE • C [A [S [P [V]]]]
```

Here, the units P, S, A and C are all generic units. Moreover, the components are combined in the way prescribed in the result unit of ARCH_SBCS: for any model of VALUE, the linking procedure given by the result unit term is required to provide a model of the entire system. □

A component in a UML component diagram is a modular unit that is replaceable within its environment. It may have required and provided interfaces and its internals are otherwise inaccessible. Components can be flexibly reused by 'wiring' them together. The intention is to be able to re-deploy a component independently. A UML component diagram can be written as a CASL architectural specification using the following correspondences:

Component diagrams	Architectural specifications
component	unit
required interface	argument specification
provided interface	result specification
delegation connector to required interface	lambda unit expression
delegation connector to provided interface	result unit expression
assembly connector	unit term in an application

We assume a purely hierarchical design, with no mutual dependencies between units, and thus only a pair of required and provided interfaces exists for each connection.

Figure 5 presents an example of a UML component diagram. The CASL architectural specification obtained by applying the rules above to it is:

```
arch spec STORE =
    units PRODUCT : ORDERABLEITEM;
          CUSTOMER : ACCOUNT → PERSON;
          ORDER : ORDERABLEITEM × PERSON → ORDERENTRY
    result λ X : ACCOUNT • ORDER [PRODUCT] [CUSTOMER [X]]
end
```

[2]The complete specification of the SBCS example can be found under `https://ontohub.org/hets-lib/UserManual/Sbcs.casl`.

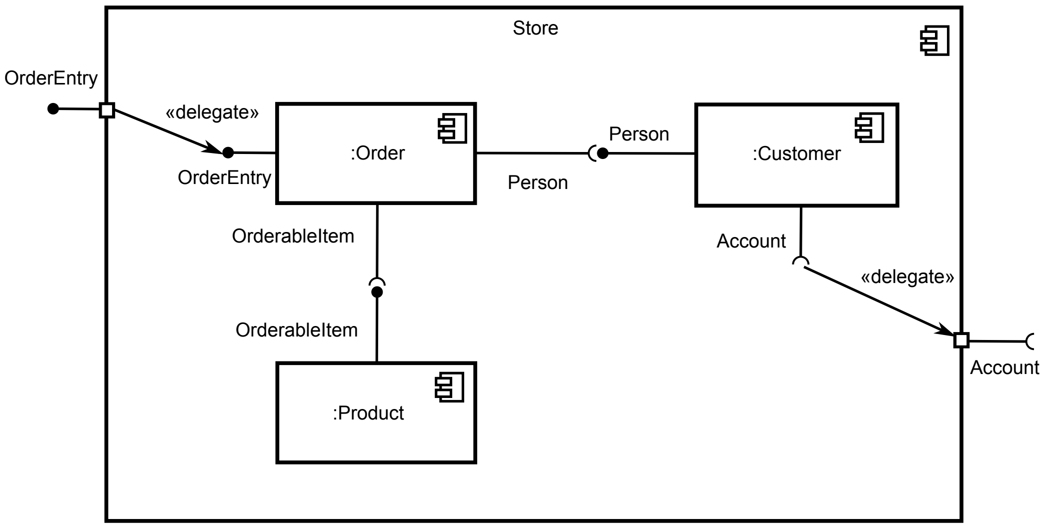


Figure 5: A UML component diagram

Furthermore, we can connect a unit A whose specification is Account with a unit S whose specification is Store using unit application $S[A]$.

With Casl architectural specifications as introduced so far, we can specify individual branching points of a formal development. Moreover, views provide a rudimentary way of expressing refinement. However, this does not suffice for all purposes. First, views formalize only refinement between non-generic unit specifications. Then, there is no way to record that a structured specification is refined to an architectural specification. Finally, while the specification of a unit in an architectural specification *ASP* can itself be an architectural specification, describing how the respective component of *ASP* is further decomposed into sub-components, the drawback is that this decision must already be recorded at the time of writing *ASP*. This means that the whole development process should be captured within a single architectural specification, without the possibility of refining the specifications of the components in a later step. Therefore, we now complement architectural specifications with a *refinement* language, which allows the user to formalize complete developments as *refinement trees*, which we formally introduce in Sect. 6.

2.4. Refinements

Our refinement language covers three kinds of refinements: refinement of unit specifications to unit specifications, called *simple refinements*; refinement of unit specifications to architectural specifications, called *branching refinements*; and refinement of named components of an architectural specification, called *component refinements*.

The intuition behind the three kinds of refinement can be easily understood using a graphical representation of refinement trees. The refinement language for Casl provides means for specifying refinement trees which are constructed using three types of building blocks, corresponding to these three kinds of refinement. Refinement trees (introduced formally in Sect. 6.2) provide a visual representation of the development process as follows:[3]

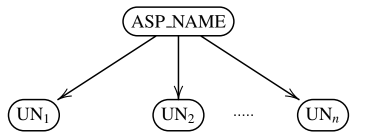
S Simple refinement steps give rise to the first type of links in refinement trees: the refinement of a specification SP to another specification SP' along σ is represented as a tree with two nodes, one labeled with the name of SP and the other with the name of SP' with an edge between the two nodes, which we write using a double arrow to denote a refinement:

$$\boxed{\text{SP}} \overset{\sigma}{\Longrightarrow} \boxed{\text{SP'}}$$

To save space, we sometimes write refinements horizontally instead of vertically. The arrow indicates which direction is "down".

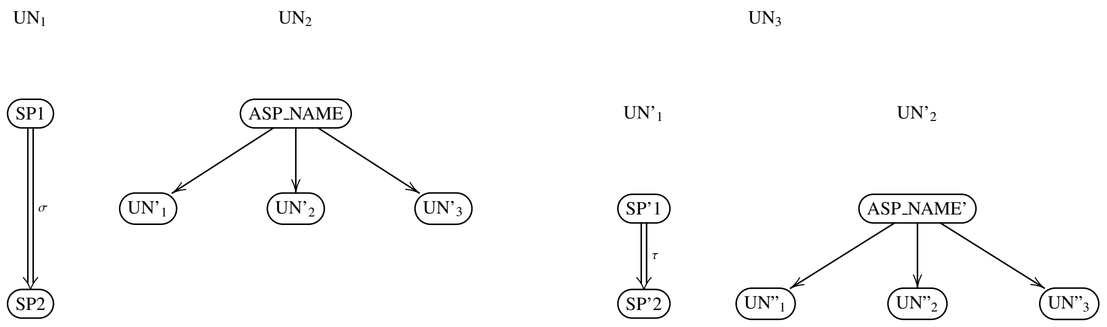
[3]We will use the notations **S**, **B** and **C** for the three types of refinement throughout the rest of the paper in informal or intuitive explanations of formal results.

B Branching refinements introduce architectural decompositions. An architectural specification ASP_NAME with n units $\text{UN}_1, \ldots \text{UN}_n$ with specifications $SPR_1, \ldots, SPR_n$, can be represented as a refinement tree as follows. In the simplest case, all SPR_i are unit specifications and then the refinement tree of ASP_NAME is



where the simple arrows denote decomposition. In the general case, SPR_i are branching refinements with refinement trees $T_1, \ldots, T_n$. The refinement tree of ASP_NAME introduces a new root node, labeled with ASP_NAME, and connects this node with the roots of each of $T_1, \ldots, T_n$ by a decomposition link, and moreover the label of the root of T_i becomes UN_i for each $i = 1, \ldots, n$. Note that the result unit term of an architectural specification is not represented in its refinement tree.

C Component refinements are forests of named refinement trees. They are used to further refine the leaves of a refinement tree. For a component refinement that contains refinements SPR_i for the units UN_i, with $i = 1, \ldots, k$, its refinement tree is the forest consisting of the refinement trees $T_1, \ldots, T_k$ of $SPR_1, \ldots, SPR_k$, labeled respectively with $\text{UN}_1, \ldots, \text{UN}_k$. It can be the case that for some $i = 1, \ldots, k$, T_i is itself a forest of named refinement trees. In the example below, we have a forest of three refinement trees for UN_1, UN_2 and UN_3, where in turn the refinement tree for UN_3 is again a forest of two refinement trees for UN'$_1$ and UN'$_2$, respectively.:



A component refinement is only meaningful if we know that the names of the trees are component names occurring in an architectural specification that we want to further refine as described by the component refinement.

Moreover, such blocks can be combined using composition of refinements, which corresponds to putting together subtrees that match at the connection points. To make this intuitive notion formal, we will have to clarify how connection points are identified (see Sect. 6.2) and what it means for the connection points to match. The latter will be done by the proof calculus for refinements that we introduce in Sect. 5.3.

We now introduce the syntax of the refinement language and illustrate it with some examples from [13]. Moreover, in the examples we will also show the refinement tree for each refinement. *Simple refinement* is written *USP* **refined via** σ **to** *USP′*, where *USP* and *USP′* are generic unit specifications of the form $SP_1 \times \ldots \times SP_n \to SP$ and $SP_1 \times \ldots \times SP_n \to SP'$ respectively and $\sigma : \mathbf{Sig}(SP) \to \mathbf{Sig}(SP')$. When **via** σ is omitted, σ is assumed to be the identity. This also covers the case of non-generic unit specifications by taking $n = 0$. We make 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 $U'|_\sigma$ for generic units $U' \in Unit(USP')$ as well; in this case, the notation denotes the unit function obtained by reducing the result w.r.t. σ after applying U'. In practice, this restriction is not troublesome, since we can always write an architectural specification that adjusts the parameter specifications as required, as we will show below. We can then express correctness of a simple refinement *USP* **refined via** σ **to** *USP′* as

$$U'|_\sigma \in Unit(USP) \text{ for each unit } U' \in Unit(USP')$$

and denote this by $USP \overset{\sigma}{\rightsquigarrow} USP'$. If *USP* and *USP′* are non-generic, we can express the refinement *USP* **refined via** σ **to** *USP′* equivalently as **view** V : *USP* **to** *USP′* = σ. Therefore, simple refinements can be regarded as a generalization of views to the generic case.

Two refinements can be combined to form a *chain* of refinements, which captures the situation $USP \overset{\sigma}{\rightsquigarrow} USP' \overset{\tau}{\rightsquigarrow} USP''$. We write this *USP* **refined via** σ **to** *USP'* **refined via** τ **to** *USP''*, or using *named* refinements as one of the three equivalent constructions C1, C2 or C3 below:

refinement R = *USP* **refined via** σ **to** *USP'*
refinement R' = *USP'* **refined via** τ **to** *USP''*
refinement C1 = *USP* **refined via** σ **to** R'
refinement C2 = *USP* **refined via** σ **to** *USP'* **then** R'
refinement C3 = R **then** R'

where the last two alternatives make use of an operation of *composition* of refinements, using the keyword **then**. The semantic rules, which will be presented in Sect. 4, determine which compositions are legal.

Example 2.3. We start with a loose specification of monoids:

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

Then natural numbers are specified with the usual Peano axioms, addition is defined, and finally the successor is hidden:

spec NatWithSuc =
  free type $Nat ::= 0 \mid suc(Nat)$ **%%** Casl shorthand for Peano axioms
  op $\_\_+\_\_ : Nat \times Nat \to Nat$, *unit* 0
  $\forall x, y : Nat \bullet x + suc\,(y) = suc(x + y)$
spec Nat = NatWithSuc **hide** *suc*

We can record that natural numbers with addition form a monoid as follows:
  refinement R1 = Monoid **refined via** $Elem \mapsto Nat$ **to** Nat

$$\begin{tikzcd}\text{Monoid} \arrow[r, Rightarrow, "Elem\mapsto Nat"] & \text{Nat}\end{tikzcd}$$

Natural numbers are further implemented as lists of binary digits, constructed by two postfix operations $\_\_0$ and $\_\_1$. $\_\_ + \_\_$ is addition, and $\_\_ + +\_\_$ is addition with carry bit.
  spec NatBin =
    generated type $Bin ::= 0 \mid 1 \mid \_\_0(Bin) \mid \_\_1(Bin)$
    ops $\_\_+\_\_, \_\_++\_\_ : Bin \times Bin \to Bin$
    $\forall x, y : Bin$
    $\bullet\ 0\,0 = 0$
    $\bullet\ 0\,1 = 1$
    $\bullet\ \neg\, 0 = 1$
    $\bullet\ x\,0 = y\,0 \Rightarrow x = y$
    $\bullet\ \neg\, x\,0 = y\,1$
    $\bullet\ x\,1 = y\,1 \Rightarrow x = y$
    $\bullet\ 0 + 0 = 0$
    $\bullet\ 0 ++ 0 = 1$
    $\bullet\ x\,0 + y\,0 = (x + y)\,0$
    $\bullet\ x\,0 ++ y\,0 = (x + y)\,1$
    $\bullet\ x\,0 + y\,1 = (x + y)\,1$
    $\bullet\ x\,0 ++ y\,1 = (x ++ y)\,0$
    $\bullet\ x\,1 + y\,0 = (x + y)\,1$
    $\bullet\ x\,1 ++ y\,0 = (x ++ y)\,0$
    $\bullet\ x\,1 + y\,1 = (x ++ y)\,0$
    $\bullet\ x\,1 ++ y\,1 = (x ++ y)\,1$
We obtain a new refinement:
  refinement R2 = Nat **refined via** $Nat \mapsto Bin$ **to** NatBin

$$\begin{tikzcd}\text{Nat} \arrow[r, Rightarrow, "Nat\mapsto Bin"] & \text{NatBin}\end{tikzcd}$$

which can be composed with the first one:

  refinement R3 = R1 **then** R2

$$\begin{tikzcd}\text{Monoid} \arrow[r, Rightarrow, "Elem\mapsto Nat"] & \text{Nat} \arrow[r, Rightarrow, "Nat\mapsto Bin"] & \text{NatBin}\end{tikzcd}$$

□

As a next step, we can introduce branching by refining to an architectural specification:

refinement R = *USP* **refined via** σ **to arch spec** *ASP*

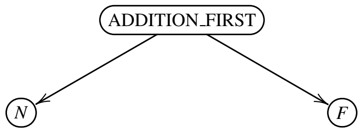
The original architectural language (as in [6]) already permits that the specification of a component unit of an architectural specification is itself an architectural specification. This means that we are allowed to record decisions in an architectural specification regarding the design of a component. Since the new refinement language provides a more expressive language for such design decisions, it is natural to allow unit declarations to make full use of it, by generalizing the specifications of units to arbitrary refinements:

refinement R = *USP* **refined via** σ **to** *USP′*
arch spec ASP_NAME = {
  units U : R
  ... }

This means that we require the developer of the component U to provide a realization of *USP′*, which, as recorded by the refinement R, is known to be an acceptable realization of *USP* along σ.

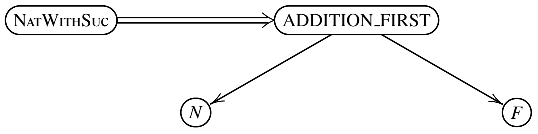
Example 2.4. Suppose that we want to implement not only Nat, but NatWithSuc, i.e. also the successor function. Now, while the presence of the successor function enables an easy specification of the natural numbers, it may be a little distracting in achieving an efficient implementation. So we can help 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;
    F : Nat → NatWithSuc
  result F[N]



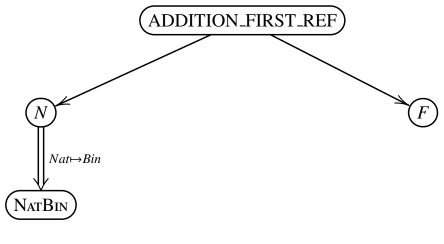
We thus have chosen to split the implementation of NatWithSuc into two independent subtasks: the implementation of Nat, and the implementation of a generic program, that given any Nat-model will realise the successor function on top of it. The generic program is then applied once to the implementation N of Nat. We can record this design decision as:

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



If we want to record at this point that we have also made the design decision to implement Nat with NatBin, we can write a refinement directly after the specification of the unit in question:

arch spec Addition_First_Ref =
  units
    N : Nat **refined via** *Nat* $\mapsto$ *Bin* **to** NatBin ;
    F : Nat → NatWithSuc
  result F[N]



□

These two constructions give us the second kind of refinements, *branching refinements*. In a bottom-up approach to system development by stepwise refinement, one combines previously developed units, possibly reused from an existing library, into a new system. The specification of each unit has been previously refined, and this information is made available when making the combination. If a system has units A_1, ..., A_n and their specifications have been refined as captured by the refinements R_1, ..., R_n, it remains to combine the units into an architectural specification:

arch spec ASP_NAME =
    units
        A_1 : R_1;
        . . .
        A_N : R_N
    result UE

that is, finding a linking procedure UE involving A_1, . . ., A_N. The refinements R_1, . . ., R_N can be arbitrarily complex.

Finally, we also allow the (named) components of architectural specifications to be further refined using *component refinements*, which are written $\{UN_i$ **to** $SPR_i\}_{i\in\mathcal{J}}$, where $\mathcal{J}$ is a set of indices and for each $i \in \mathcal{J}$, UN_i stands for a unit name and SPR_i for a refinement. This expresses that for each $i \in \mathcal{J}$, the component with the name UN_i of the architectural specification at hand is further refined to SPR_i. Component refinement is typically used in the context of a composition of the form

refinement R = **arch spec** *ASP_NAME* **then** $\{UN_i$ **to** $SPR_i\}$

which specifies that the component with the name UN_i of *ASP_NAME* is further refined to SPR_i. The semantic rules, which will be introduced in Sect. 4, prevent the specification of a unit in an architectural specification from being a component refinement; that is, only simple or branching refinements are allowed here.

From a practical perspective, component refinements can be explained as follows. In a top-down approach to system development, one would start by decomposing the task of providing an implementation for the requirement specification *SP* into smaller subtasks and record this decision with an architectural specification ASP_NAME:

arch spec ASP_NAME =
    units
        A_1 : USP_1;
        . . .
        A_N : USP_N
    result UE

where UE is a unit expression involving A_1, . . ., A_N. This is followed by decomposing the specifications of the units into architectural specifications ASP_1, . . ., ASP_N. This is recorded as follows:

refinement R = ASP_NAME **then** {A_1 **to arch spec** ASP_1, . . ., A_N **to arch spec** ASP_N }

A further decomposition of the units B_1, . . ., B_K of ASP_I, for an index I ∈ {1, . . . N}, can be recorded in the same way, using composition of refinements:

refinement R' = R **then** {A_I **to** {B_1 **to arch spec** ASP'_1, . . ., B_K **to arch spec** ASP'_K }}

Example 2.5. We can now record the decision of implementing the component N of Addition_First with NatBin in a later step, as a component refinement:

refinement RComp = {N **to** R2}
refinement R5 = R4 **then** RComp

N

$$\begin{tikzcd}
\textsc{Nat} \arrow[r, Rightarrow, "Nat\mapsto Bin"] & \textsc{NatBin}
\end{tikzcd}$$

$$\begin{tikzcd}
& \textsc{NatWithSuc} \arrow[d, Rightarrow] & \\
& \text{ADDITION\_FIRST} \arrow[dl] \arrow[dr] & \\
N \arrow[d, Rightarrow, "Nat\mapsto Bin"] & & F \\
\textsc{NatBin} & &
\end{tikzcd}$$

Using a component refinement, R5 avoids the construction of Addition_First_Ref, which has been obtained from Addition_First by syntactic substitution. This shows that component refinements increase the possibilities of re-using existing refinements. □

`LIB-ITEM` ::= ... | `REF-SPEC-DEFN`
`REF-SPEC-DEFN` ::= **refinement** `REF-SPEC-NAME` = `SPR`
`SPR` := `USP` | **arch spec** `ASP` | `SPR` **then** `SPR` |
    `USP` **refined** ⟨**via** σ⟩ **to** `SPR` | $\{A_1$ **to** $\texttt{SPR}_1, \ldots, A_n$ **to** $\texttt{SPR}_n\}$
`UDECL` ::= A : `SPR` | A : `USP` **given** $\texttt{UT}_1, \ldots, \texttt{UT}_n$
`USP` ::= `SPEC` | $\texttt{SPEC}_1 \times \cdots \times \texttt{SPEC}_n \rightarrow$ `SPEC` ~~| **arch spec** `ASP`~~

Figure 6: Syntax of the CASL refinement language.

The complete syntax for CASL refinements is presented in Fig. 6, where $A, A_1, \ldots, A_n$ stand for unit names and σ for a signature morphism. We eliminate architectural specifications as an alternative of unit specifications, since we make them available as a particular case of refinements. The syntax of CASL architectural specifications (introduced in Fig. 4) is modified for unit declarations and unit specifications as indicated in Fig. 6 and is otherwise subsumed without change by the syntax of CASL refinements.

2.5. Examples and Methodology

Example 2.6. We write the initial refinement of the steam boiler system (Example 2.2) as

refinement REF_SBCS = SBCS_OPEN **refined to**
  arch spec ARCH_SBCS

$$\begin{tikzcd}
\text{SBCS\_OPEN} \arrow[rr, Rightarrow] & & \text{ARCH\_SBCS} \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & \\
P & S & A & C
\end{tikzcd}$$

We proceed with refining individual units. The specifications of C and S in ARCH_SBCS above do not require further architectural decomposition. The specification of S, recorded in the unit specification STATE_ABSTR, can be refined by providing an implementation of states as a record of all observable values. This is done in SBCS_STATE_IMPL, assuming an implementation of PRELIMINARY; we record this development in the unit specification UNIT_SBCS_STATE.[4] The refinement of S is then written in STATE_REF[5]

unit spec STATE_ABSTR =
  PRELIMINARY → SBCS_STATE
unit spec UNIT_SBCS_STATE =
  PRELIMINARY → SBCS_STATE_IMPL
refinement STATE_REF =
    STATE_ABSTR **refined to** UNIT_SBCS_STATE

$$\begin{tikzcd}
\text{STATE\_ABSTR} \arrow[r, Rightarrow] & \text{UNIT\_SBCS\_STATE}
\end{tikzcd}$$

For the units A and P, we proceed with designing their architecture. This is recorded in the architectural specifications ARCH_ANALYSIS[6] and ARCH_PRELIMINARY [7]:

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 λ S : SBCS_STATE • MTS [ME [PR [FD [S]]]]

$$\begin{tikzcd}
 & \text{ARCH\_ANALYSIS} \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\text{FD} & \text{PR} & \text{ME} & \text{MTS}
\end{tikzcd}$$

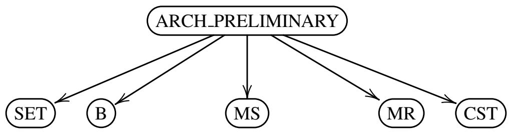
[4]See `https://spechub.org/casl/hets-lib/UserManual/Sbcs.casl` for SBCS_STATE_IMPL and other specifications in the steam boiler system example that are referred to here.

[5] This can be equivalently written **refinement** STATE_REF' = PRELIMINARY → SBCS_STATE **refined to** PRELIMINARY → SBCS_STATE_IMPL.

[6]The specification MODE_EVOLUTION is generic, and is instantiated with the actual parameter PU_PREDICTION. Although the syntax is similar, the concepts of (applications of) generic units and generic specifications are different, see [3].

[7]Since two of the units of ARCH_PRELIMINARY have imports, its refinement tree is actually slightly different than the one we show here. This will be clarified in Sect. 5.2 and the correct refinement (sub)tree is presented in Fig. 20.

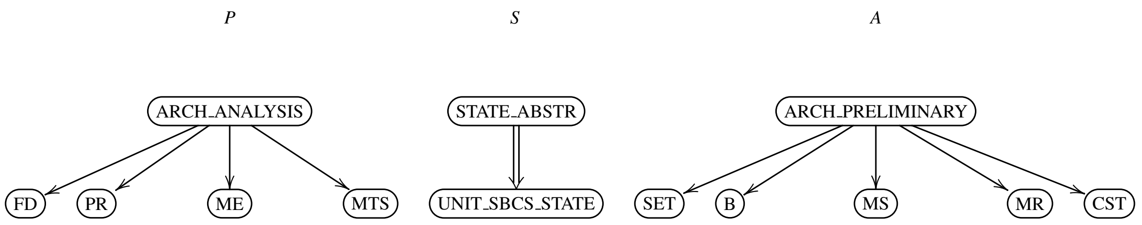
arch spec Arch_Preliminary =
units
SET : {**sort** *Elem*} × Nat → Set[**sort** *Elem*];
B : Basics;
MS : Messages_Sent **given** B;
MR : Value → Messages_Received **given** B;
CST : Value → Sbcs_Constants
result λ *V* : Value
• SET [MS **fit** *Elem* ↦ *S_Message*] [V]
and SET [MR [V] **fit** *Elem* ↦ *R_Message*] [V]
and CST [V]



We can now record the component refinement:

refinement Ref_Sbcs' = Ref_Sbcs **then**
{P **to arch spec** Arch_Preliminary, S **to** StateRef,
A **to arch spec** Arch_Analysis}

The refinement tree of the component refinement following **then** is:

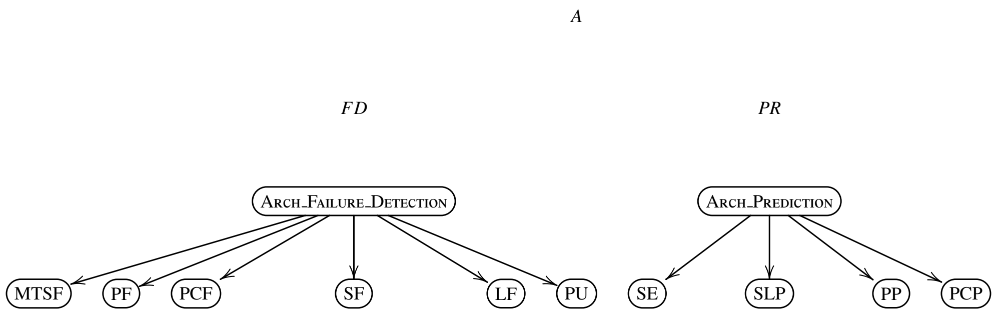


The refinement tree of REF_SBCS' consists of the first five levels of the tree in Fig. 20 in Sect. 6.2.

Moreover, the components *FD* and *PR* of Arch_Analysis are further refined (the architectural specifications Arch_Failure_Detection and Arch_Prediction are omitted here,[8]:

refinement Ref_Sbcs" =
Ref_Sbcs'
then {A **to**
{FD **to arch spec** Arch_Failure_Detection,
PR **to arch spec** Arch_Prediction }}

The refinement tree of Ref_Sbcs" is shown in Fig. 20. For the component refinement following **then** we have the following tree:



□

[8]See `https://spechub.org/casl/hets-lib/UserManual/Sbcs.casl`.

Remark 2.1. The grammar of the refinement language, presented in Fig. 6, allows combinations between refinements whose refinement trees do not match, for example:

refinement R = **arch spec** ASP_NAME **then** USP

The static and model semantics rules for refinements that we will introduce in Sect. 4 will reject such refinements as ill-formed.

3. Foundations

We now introduce the mathematical background that is needed for making the language syntax introduced in Sect. 2 precise and for developing a mathematical semantics for it. As already introduced, the central notion is that of *institution*, introduced by Goguen and Burstall [15] in order to capture the notion of logical system formally and abstractly. An institution will provide syntax and semantics for basic specifications, while the syntax and semantics of structured and architectural specifications as well as refinements can be defined over an arbitrary but fixed institution. Thus, the notion of institution is the central abstraction barrier for separating the different language layers and making them orthogonal.

We assume that the reader is familiar with the basic notions of category theory. Our notations largely follow [17], with the exception of composition of morphisms in a category, which we write in diagrammatic order using ";".

Definition 3.1. *An* institution $I = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ *consists of*[9]*:*

- *a category* **Sig** *of* signatures,
- *a functor* $\mathbf{Sen}\colon \mathbf{Sig} \to \mathbb{S}et$*, giving for each signature* Σ *a set of* sentences $\mathbf{Sen}(\Sigma)$*, and for each signature morphism* $\sigma\colon \Sigma \to \Sigma'$*, a* sentence translation map $\mathbf{Sen}(\sigma)\colon \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$*, where we may write* $\mathbf{Sen}(\sigma)(e)$ *as* $\sigma(e)$*,*
- *a functor* $\mathbf{Mod}\colon \mathbf{Sig}^{op} \to \mathbb{C}at$ *giving for each signature* Σ *a category of* models $\mathbf{Mod}(\Sigma)$ *and for each signature morphism* $\sigma\colon \Sigma \to \Sigma'$*, a* reduct functor $\mathbf{Mod}(\sigma)\colon \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$*, where we may write* $\mathbf{Mod}(\sigma)(M')$ *as* $M'|_\sigma$*;*
- *a binary relation* $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$*, for each signature* Σ*, called the* satisfaction relation

such that the following satisfaction condition *holds:*

$$M'|_\sigma \models_\Sigma e \iff M' \models_{\Sigma'} \sigma(e)$$

for each signature morphism $\sigma\colon \Sigma \to \Sigma'$ *,* Σ*-sentence* e *and* Σ'*-model* M'*.*

□

Example 3.1. [15] *First-order Logic.* In the institution $\mathbb{FOL}^=$ of (unsorted) first-order logic with equality, signatures are first-order signatures, consisting of sets of function symbols with arities and of predicate symbols with arities. Signature morphisms map symbols so that arities are preserved. Models are first-order structures with non-empty universes and sentences are closed first-order formulas. Sentence translation means replacement of symbols by their translations. Model reduct reassembles the model's components according to the signature morphism. Satisfaction is the usual satisfaction of a first-order sentence in a first-order structure. □

Example 3.2. [15] *Multi-sorted first-order logic.* In the institution $\mathbb{MSFOL}^=$ of multi-sorted first-order logic with equality, signatures consist of a set of sorts and sets of sorted function and predicate symbols, meaning that for a function symbol f with n arguments, we give the sorts $s_1, \ldots, s_n$ of the arguments and the sort s of the result (written $f : s_1 * s_2 * \ldots * s_n \to s$), and for a predicate symbols p with n argument we give the sorts $s_1, \ldots, s_n$ of the arguments

[9]As usual, $\mathbb{S}et$ is the category having all small sets as objects and functions as arrows and $\mathbb{C}at$ is the category of categories and functors (strictly speaking, a so-called quasicategory, which is a category that lives in a higher set-theoretic universe).

(written $p : s_1 * s_2 * \ldots * s_n$). Signature morphisms map sorts to sorts, function symbols to function symbols and predicate symbols to predicate symbols such that the sorts of the arguments and result (for function symbols) are preserved: if $\sigma : \Sigma \to \Sigma'$ is a signature morphism and $f : s_1 * s_2 * \ldots * s_n \to s$ is a function symbol in Σ, then its image under σ, denoted $\sigma(f)$, must have argument and result sorts $\sigma(s_1) * \sigma(s_2) * \ldots * \sigma(s_n) \to \sigma(s)$ and similarly for predicate symbols. Models interpret sorts as non-empty sets (carriers), function symbols as functions and predicate symbols as predicates on the sets given by their argument sorts. Sentences are formed from terms: if t_i is a term of sort s_i, for $i = 1, \ldots, n$, and $f : s_1 * s_2 * \ldots * s_n \to s$ is a function symbol, then $f(t_1, \ldots, t_n)$ is a term of sort s. Then sentences are closed multi-sorted first-order formulas, defined inductively as follows:

- the atomic formulas are applications of predicate symbols to a list of terms of appropriate sorts and equalities between terms of the same sort;
- if X is a sorted set of variables and Φ, Φ_1, Φ_2 are formulas, then $\neg\Phi$, $\Phi_1 \wedge \Phi_2$, $\Phi_1 \vee \Phi_2$, $\Phi_1 \implies \Phi_2$, $\Phi_1 \iff \Phi_2$, $\forall X \bullet \Phi$ and $\exists X \bullet \Phi$ are formulas.

Sentence translation, model reduct and satisfaction of formulas in models are similar to the case of first-order logic. □

Example 3.3. [6] *Multi-sorted first order logic with sort generation constraints.* This logic extends $\mathbb{MSFOL}^{=}$ with a new kind of sentences, sort generation constraints. A sort generation constraint over a signature Σ is a triple (S', F', σ') such that $\sigma' : \Sigma' \to \Sigma$ and S' and F' are sets of, respectively, sort and function symbols of Σ'. Sort generation constraints are translated along a signature morphism $\varphi : \Sigma \to \Sigma''$ by composing the morphism σ' in their third component with φ. A sort generation constraint (S', F', σ') holds in a model M if the carriers of $M|_{\sigma'}$ of the sorts in S' are generated by function symbols in F', which means that for every sort $s' \in S'$ and each $a \in (M|_{\sigma'})_{s'}$, there must exist a Σ'-term t with function symbols only from F' and variables of sorts not in S' and a variable assignment ρ into $M|_{\sigma'}$ such that the interpretation of t in $M|_{\sigma'}$ under ρ is a. Thus, the sorts specified in a sort generation constraint have no elements that cannot be reached via the constructors specified in F'. This is the so-called "no junk" principle. In CASL a sort generation constraint appears as a generated type declaration. CASL also has free type declarations, that require moreover that the ranges of the constructors of the same sort are disjoint (this is called "no confusion"). □

The CASL logic [6] adds subsorting and partiality to multi-sorted first order logic with sort generation constraints. Since we do not use these in our examples, we refrain from giving the details here. Many other logics, including propositional, higher-order, modal, temporal and Hoare logics can be presented as institutions. For more examples of institutions see [18, 3].

The signatures of many institutions come naturally equipped with a notion of subsignature, hence signature inclusion, and a well-defined way of forming a union of signatures. These concepts can be captured in a categorical setting using *inclusion systems* [19]. However, we will work with a slightly different version of this notion:

Definition 3.2. *An* inclusive category *is a category with a broad subcategory*[10] *which is a partially ordered class with a least element (denoted $\emptyset$), non-empty products (denoted $\cap$) and finite coproducts (denoted $\cup$), such that for each pair of objects A, B, the following is a pushout in the category:*

$$\begin{tikzcd}
A \cap B \arrow[r, hook] \arrow[d, hook] & A \arrow[d, hook] \\
B \arrow[r, hook] & A \cup B
\end{tikzcd}$$

□

For any objects A and B of an inclusive category, we write $A \subseteq B$ if there is an inclusion from A to B; the unique such inclusion will then be denoted by $\iota_{A \subseteq B} : A \hookrightarrow B$, or simply $A \hookrightarrow B$.

A functor between two inclusive categories is inclusive if it takes inclusions in the source category to inclusions in the target category.

[10]That is, with the same objects as the original category.

Definition 3.3. *An institution $I = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ is* inclusive[11] *if*

- **Sig** *is an inclusive category,*
- **Sen** *is inclusive and preserves intersections.*[12]

Moreover, we asume that reducts w.r.t. signature inclusions are surjective on objects. [13] □

The least object in the category of signatures will be referred to as the empty signature; indeed, in typical signature categories it is empty. In inclusive institutions, if $\Sigma_1 \subseteq \Sigma_2$ via an inclusion $\iota \colon \Sigma_1 \hookrightarrow \Sigma_2$ and $M \in \mathbf{Mod}(\Sigma_2)$, we write $M|_{\Sigma_1}$ for $M|_\iota$. Note that $\mathbf{Sen}(\iota) \colon \mathbf{Sen}(\Sigma_1) \to \mathbf{Sen}(\Sigma_2)$ is the usual set-theoretic inclusion, hence its application may be omitted.

The institutions presented above can be equipped with the obvious inclusion system on their signatures and models, and then become inclusive institutions.

Definition 3.4. *(adapted from [21]) A category* with coherent pushouts of inclusions *is an inclusive category with two partial operators that define for a morphism $m : A \to B$ and an inclusion $A \xhookrightarrow{\Delta} X$ an object $B(\Delta)$ that includes B and a morphism $m(\Delta) : X \to B(\Delta)$ such that the diagram on the left below is a pushout.*

$$\begin{tikzcd}
A \arrow[r, hook, "\Delta"] \arrow[d, "m"'] & X \arrow[d, "m(\Delta)"] \\
B \arrow[r, hook] & B(\Delta)
\end{tikzcd}
\qquad
\begin{tikzcd}
X \arrow[rrr, "m(\Delta)"] \arrow[dr, "f"] & & & B(\Delta) \arrow[dl, "m(f)"'] \\
& X' \arrow[r, "m(\Delta')"] & B(\Delta') & \\
A \arrow[uu, hook, "\Delta"] \arrow[ur, hook, "\Delta'"] \arrow[rrr, "m"'] & & & B \arrow[uu, hook] \arrow[ul]
\end{tikzcd}$$

In that case, given an arrow $f : X \to X'$ such that $\Delta; f = \Delta'$ we write $m(f)$ for the universal arrow from $B(\Delta)$ to $B(\Delta')$ induced by the pushout as in the diagram on the right above.

The coherence requires that pushouts of inclusions commute with identities and compositions, as in [21]:

- *when $m = id_A$, $A(\Delta) = X$ and $id_A(\Delta) = id_X$*
- *when $\Delta = id_A$, $B(id_A) = B$ and $m(id_A) = m$*
- *$(m; n)(\Delta) = m(\Delta); n(\Delta')$ and $C(\Delta) = C(\Delta')$*
- *$m(\Delta; \Delta') = (m(\Delta))(\Delta')$ and $B(\Delta; \Delta') = (B(\Delta))(\Delta')$*

where the equations between morphisms imply those between their codomains and the diagrams for the two last items are as below

$$\begin{tikzcd}
X \arrow[r, "m(\Delta)"'] \arrow[rr, bend left, "(m;n)(\Delta)"] & B(\Delta) \arrow[r, "n(\Delta')"'] & C(\Delta) = C(\Delta') \\
A \arrow[u, hook, "\Delta"] \arrow[r, "m"'] & B \arrow[u, hook, "\Delta'"] \arrow[r, "n"'] & C \arrow[u, hook]
\end{tikzcd}
\qquad
\begin{tikzcd}
Y \arrow[r, bend left, "m(\Delta;\Delta')"] \arrow[r, bend right, "(m(\Delta))(\Delta')"'] & (B(\Delta))(\Delta') = B(\Delta;\Delta') \\
X \arrow[u, hook, "\Delta'"] \arrow[r, "m^X"'] & B(\Delta) \arrow[u, hook] \\
A \arrow[u, hook, "\Delta"] \arrow[r, "m"'] & B \arrow[u, hook]
\end{tikzcd}$$

□

[11]Even though we use the same term as in [20], since the overall idea is the same, on one hand, some of our assumptions here are weaker than in [20], and on the other hand, we require a bit more structure on the category of signatures.

[12]That is, for any non-empty family of signatures $\mathbb{S} \subseteq |\mathbf{Sig}|$, $\mathbf{Sen}(\bigcap \mathbb{S}) = \bigcap_{\Sigma \in \mathbb{S}} \mathbf{Sen}(\Sigma)$.

[13]The semantics of DOL requires also that each model category of I is inclusive, and reduct functors are inclusive. This condition is not needed in our context and we have omitted it.

This allows us to speak about the selected pushout of a span where an arrow is an inclusion. In the case of the CASL logic, [6] presents a set-theoretical construction of the selected pushout.[14]

Let $\mathbb{C}$ be a weakly inclusive category. We say that $\mathbb{C}$ *has differences*, if there is a binary operation $\setminus$ on objects, such that for each pair of objects A, B, $A \setminus B$ is the largest object that is included in A and its intersection with B is the empty object.

Definition 3.5. *In an inclusive category $\mathbb{C}$, let $\sigma_1 : \Sigma_1 \to \Sigma'_1$ and $\sigma_2 : \Sigma_2 \to \Sigma'_2$ be two arrows. We define their union $\sigma_1 \cup \sigma_2$ as the universal arrow from $\Sigma_1 \cup \Sigma_2$ to $\Sigma'_1 \cup \Sigma'_2$ induced by the pushout as below, if the outer diagram is a cocone:*

$$\begin{tikzcd}
 & \Sigma_1 \arrow[rr, "\sigma_1"] \arrow[dr, hook] & & \Sigma'_1 \arrow[dr, hook] & \\
\Sigma_1 \cap \Sigma_2 \arrow[ur, hook] \arrow[dr, hook] & & \Sigma_1 \cup \Sigma_2 \arrow[rr, dashed, "\sigma_1 \cup \sigma_2"] & & \Sigma'_1 \cup \Sigma'_2 \\
 & \Sigma_2 \arrow[rr, "\sigma_2"] \arrow[ur, hook] & & \Sigma'_2 \arrow[ur, hook] &
\end{tikzcd}$$

The union of the two arrows is defined only if the outer diagram is a cocone. Intuitively, this condition ensures that σ_1 and σ_2 agree on $\Sigma_1 \cap \Sigma_2$. $\square$

This is generalized to finite unions of signature morphisms in the expected way.

We are going to make use of the following notions in an arbitrary inclusive institution I. A *diagram D* is a functor from a small category to the category of signatures of I. In the following, let D be a diagram. The objects and the morphisms of its source category are referred to as its nodes and its edges, respectively. A family of models $\mathcal{M} = \{M_p\}_{p \in Nodes(D)}$ indexed by the nodes of D is *compatible with D* if for each node p of D, $M_p \in \mathbf{Mod}(D(p))$ and for each edge $e \colon p \to q$, $M_p = M_q|_{D(e)}$. A *sink* η on a subset K of nodes consists of a signature Σ together with a family of morphisms $\{\eta_p \colon D(p) \to \Sigma\}_{p \in K}$. We say that D *ensures amalgamability* along $\eta = (\Sigma, \{\eta_p \colon D(p) \to \Sigma\}_{p \in K})$ if for every model family $\mathcal{M}$ compatible with D there is a unique model $M \in \mathbf{Mod}(\Sigma)$ such that for all $p \in K$, $M|_{\eta_p} = M_p$. If such a model M exists (without necessarily being unique) then D ensures *weak amalgamability* along η. Moreover, if K consists of all nodes of D and for each edge $i \colon p \to q$ in D we have that $\eta_p = D(i); \eta_q$, η is called a (*weakly*) *amalgamable cocone* for D. Given a family of signatures $\{\Sigma_i\}_{i=1,\dots,n}$ and a family of Σ_i-models $\mathcal{M} = \{\mathcal{M}_i \in \mathbf{Mod}(\Sigma_i)\}_{i=1,\dots,n}$, we define $\mathcal{M}_1 \oplus \cdots \oplus \mathcal{M}_n = \{M \in \mathbf{Mod}(\Sigma_1 \cup \cdots \cup \Sigma_n) \mid M|_{\Sigma_i} \in \mathcal{M}_i, i \in 1, \dots, n\}$.

For an inclusive institution I, a *presentation* is a pair (Σ, E) where Σ is a signature and E is a set of Σ-sentences. For a set E of Σ-sentences, $\mathbf{Mod}_\Sigma(E)$ denotes the class of all Σ-models satisfying E. If E_1 and E_2 are sets of Σ-sentences, we say that E_2 is a *logical consequence* of E_1, denoted $E_1 \models E_2$, if for each model M of E_1, M is also a model of E_2. A *presentation morphism* $\sigma : (\Sigma_1, E_1) \to (\Sigma_2, E_2)$ is a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ such that $E_2 \models \sigma(E_1)$. We obtain thus a new institution, which we denote I^{pres}, having as its category of signatures the category of I-presentations and their morphisms, as sentences over a presentation (Σ, E) simply all Σ-sentences, and $\mathbf{Mod}(\Sigma, E) = \mathbf{Mod}_\Sigma(E)$; the satisfaction relation is the same as in I. Presentations provide the simplest form of specifications, and they provide the formalization of *basic specifications*.

CASL uses *symbol maps* as a convenient way of writing down signature morphisms. Instead of having to write down for each element of the signature its corresponding image along the signature morphism, the user can provide a partial mapping between the symbols of the source and target signatures. The mapping may either determine unambiguously a signature morphism, which is then used in the specification, or, ambiguously, several signature morphisms, in which case an error is generated (and the user is expected to provide more detail about the mapping). Here we write signature morphisms directly instead of introducing symbol maps.

Remark 3.1. We also assume that there exists a sound proof calculus for proving refinements of structured specifications or, equivalently, correctness of views; possible choices include the calculi defined in [3, 23, 24] and the formalism of development graphs [25]. See also the discussion in Sect. 6. $\square$

[14]This works for the institutions with qualified symbols defined in [22] too.

4. Semantics of Casl Refinements

Before we come to the semantics of refinements, we first need to discuss the semantics of both structured and architectural specifications. Note that Casl has model-theoretic semantics: a Casl specification denotes a signature (determined by the *static semantics*) and a class of models over that signature (determined by the *model semantics*). This applies to all specifications, at each layer of Casl.

4.1. Semantics of Casl Structured Specifications

The signature of a structured specification SP will be given by the rules of the static semantics for proving judgements of the form $\vdash SP \rhd \Sigma$, where Σ is a signature, and then we sometimes write $\mathbf{Sig}(SP)$ for Σ. Similarly, the class of models of SP will be given by the rules of the model semantics, with judgements of the form $\vdash SP \Rightarrow \mathcal{M}$, where $\mathcal{M}$ is a class of Σ-models, and then we sometimes write $\mathbf{Mod}(SP)$ for $\mathcal{M}$. A structured specification SP is *consistent* if its model class $\mathbf{Mod}(SP)$ is non-empty.

- **Basic specifications:** Given a signature Σ and a set E of Σ-sentences, (Σ, E) is a structured specification such that:

$$\frac{}{\vdash (\Sigma, E) \rhd \Sigma} \qquad \frac{}{\vdash (\Sigma, E) \Rightarrow \mathbf{Mod}_\Sigma(E)}$$

- **Union:**

$$\frac{\begin{array}{c}\vdash SP_1 \rhd \Sigma_1\\ \vdash SP_2 \rhd \Sigma_2\end{array}}{\vdash SP_1 \ \mathbf{and}\ SP_2 \rhd \Sigma_1 \cup \Sigma_2} \qquad \frac{\begin{array}{c}\vdash SP_1 \Rightarrow \mathcal{M}_1\\ \vdash SP_2 \Rightarrow \mathcal{M}_2\end{array}}{\vdash SP_1 \ \mathbf{and}\ SP_2 \Rightarrow \{M \in \mathbf{Mod}(\Sigma_1 \cup \Sigma_2) \mid M|_{\iota_{\Sigma_i \subseteq \Sigma_1 \cup \Sigma_2}} \in \mathcal{M}_i, i = 1, 2\}}$$

- **Translation:**

$$\frac{\begin{array}{c}\vdash SP \rhd \Sigma\\ \sigma : \Sigma \to \Sigma'\end{array}}{\vdash SP \ \mathbf{with}\ \sigma \rhd \Sigma'} \qquad \frac{\vdash SP \Rightarrow \mathcal{M}}{\vdash SP \ \mathbf{with}\ \sigma \Rightarrow \{M' \in \mathbf{Mod}(\Sigma') \mid M'|_\sigma \in \mathcal{M}\}}$$

- **Hiding:**

$$\frac{\begin{array}{c}\vdash SP' \rhd \Sigma'\\ \sigma : \Sigma \to \Sigma'\end{array}}{\vdash SP' \ \mathbf{hide}\ \sigma \rhd \Sigma} \qquad \frac{\vdash SP' \Rightarrow \mathcal{M}}{\vdash SP' \ \mathbf{hide}\ \sigma \Rightarrow \{M'|_\sigma \mid M' \in \mathcal{M}\}}$$

4.2. Semantics of Unit Specifications

As discussed in Sect. 2.2, architectural specifications involve the specification of units. Units may be models or generic units mapping models to models. Specifications of units consist of parameter and result specifications, indicating the properties assumed about the arguments and the properties guaranteed of the result. Formally, if $SP_1, \ldots, SP_n$, SP are structured specifications, then $USP = SP_1, \ldots, SP_n \to SP$ is a unit specification. *Unit signatures* consist of a sequence of signatures for the arguments and a signature that extends their union for the result. For non-generic units ($n = 0$) this reduces to a single signature. The signature of a unit specification is determined as follows:

$$\frac{\begin{array}{c}\vdash SP_1 \rhd \Sigma_1\\ \ldots\\ \vdash SP_n \rhd \Sigma_n\\ \vdash SP \rhd \Sigma\\ \Sigma \text{ extends } \Sigma_1 \cup \cdots \cup \Sigma_n\end{array}}{\vdash USP \rhd \Sigma_1, \cdots, \Sigma_n \to \Sigma}$$

where $\Sigma_1, \cdots, \Sigma_n \to \Sigma$, the signature of USP, is typically denoted by $U\Sigma$. Given a unit signature $U\Sigma = \Sigma_1, \cdots, \Sigma_n \to \Sigma$, a $U\Sigma$-model, called a unit, is a partial function F mapping compatible models over $\Sigma_1, \ldots, \Sigma_n$ to Σ-models, in such a way that the arguments are protected, i.e. for each $\langle M_1, \ldots, M_n\rangle \in dom(F)$ we have that $F(\langle M_1, \ldots, M_n\rangle)|_{\Sigma_i} = M_i$ for $i = 1, \ldots, n$. Models over $\Sigma_1, \ldots, \Sigma_n$ are compatible if they can be amalgamated to a model over $\Sigma_1 \cup \ldots \cup \Sigma_n$.

The class of all $U\Sigma$-models is denoted $Unit(U\Sigma)$. The semantics of a unit specification $USP = SP_1, \ldots, SP_n \to SP$ is given by the following rule:

$$\frac{\begin{array}{c} \vdash USP \rhd U\Sigma \\ \vdash SP_1 \Rightarrow \mathcal{M}_1 \\ \ldots \\ \vdash SP_n \Rightarrow \mathcal{M}_n \\ \mathcal{M}_0 = \mathcal{M}_1 \oplus \cdots \oplus \mathcal{M}_n \\ \vdash SP \Rightarrow \mathcal{M} \end{array}}{\vdash USP \Rightarrow \{F \in Unit(U\Sigma) \mid \text{for all } M \in \mathcal{M}_0, \overline{M} = \langle M|_{\Sigma_1}, \ldots, M|_{\Sigma_n} \rangle \in dom(F) \text{ and } F(\overline{M}) \in \mathcal{M}\}}$$

When $n = 0$ (no parameters), unit signatures are plain signatures and (non-generic) units are just models of the corresponding signature. We denote the model class of USP as defined in the model semantics by $Unit(USP)$. A unit specification is consistent if $Unit(USP)$ is non-empty.

4.3. Semantics of Architectural Specifications

$$\frac{\begin{array}{c} \Gamma_s \vdash \mathtt{UDD}_1 \ldots \mathtt{UDD}_n \rhd C_s \\ \Gamma_s, C_s \vdash \mathtt{RESULT\_UNIT} \rhd U\Sigma \end{array}}{\Gamma_s \vdash \textbf{units}\ \mathtt{UDD}_1 \ldots \mathtt{UDD}_n\ \textbf{result}\ \mathtt{UE} \rhd (C_s, U\Sigma)} \qquad \frac{\begin{array}{c} \Gamma_s \vdash \mathtt{UDD}_1 \ldots \mathtt{UDD}_n \rhd C_s \\ \Gamma_s, \Gamma_m \vdash \mathtt{UDD}_1 \ldots \mathtt{UDD}_n \Rightarrow C \\ \Gamma_s, \Gamma_m, C_s, C \vdash \mathtt{RESULT\_UNIT} \Rightarrow UEv \end{array}}{\Gamma_s, \Gamma_m \vdash \textbf{units}\ \mathtt{UDD}_1 \ldots \mathtt{UDD}_n\ \textbf{result}\ \mathtt{UE} \rhd \{(E, UEv(E)) \mid E \in C\}}$$

Figure 7: Basic static and model semantics for basic architectural specifications.

We briefly recall in a slightly simplified form the semantics of architectural specifications (see [6, 5] for details). An architectural signature $A\Sigma = (C_s, U\Sigma)$ consists of a unit signature $U\Sigma$ for the result unit together with a *static context* C_s which is a map assigning unit signatures to the names of component units.

Starting with the initial empty static context $C_s^\emptyset$, the static semantics for declarations and definitions adds to it the signature of each new unit and the static semantics for unit terms and expressions does the type-checking in the current static context (see the rules in Fig. 7). The rules of the static semantics have judgements of the form $\Gamma_s \vdash ASP \rhd A\Sigma$ where $A\Sigma$ is an architectural signature and Γ_s is the global static environment, keeping track of the signatures of the named structured specifications. The model semantics is assumed to be applied only after a successful application of the basic static semantics. It produces a class of architectural models $\mathcal{AM}$ over the resulting architectural signature, where an architectural model over an architectural signature $A\Sigma$ consists of a result unit $UEv(E)$ over the result unit signature and a collection E of units over the signatures given in the static context, which is called a *unit environment*. The rules of the model semantics have judgements of the form $\Gamma_s, \Gamma_m \vdash ASP \Rightarrow \mathcal{AM}$ with $\mathcal{AM}$ being an architectural model and Γ_m being the global model environment, keeping track of the model class of the named structured specifications.

Figure 8 gives the basic static semantics and model semantics for unit declarations. Unit declarations and definitions denote a set C of unit environments. We write $C^\emptyset$ for the empty set of unit environments. Unit terms and unit expressions denote unit evaluators UEv which, given a unit environment (that records the units for the unit names that can appear in the term) deliver a unit (over the signature given by the static semantics). See Figs. 9 and 10 for the semantics of typical unit term constructs. If we only want to state that the static semantics is successful for an architectural specification ASP, we write $\vdash ASP \rhd \square$. An architectural specification ASP is consistent if the class $\mathcal{AM}$ of its architectural models is non-empty.

Figure 9 presents the basic static semantics and model semantics for unit amalgamation. The static semantics states that the signature of the amalgamation of two units is the union of the signatures of the units. The model semantics first analyses the two unit expressions T_1 and T_2 in a unit context C, obtaining the unit evaluators UEv_1 and UEv_2 respectively. Then the rule checks whether for each pair of models M_1 and M_2 resulting from the evaluation of T_1 and T_2, respectively, in the same unit environment E, their amalgamation $M_1 \oplus M_2$ is defined, and if that is the

$$\frac{\begin{array}{c}\Gamma_s \vdash \texttt{UNIT\_SPEC} \rhd \Sigma \\ UN \notin dom(C_s)\end{array}}{\Gamma_s, C_s \vdash UN \;:\; \texttt{UNIT\_SPEC} \rhd \{UN \mapsto \Sigma\}} \qquad \frac{\begin{array}{c}\Gamma_s \vdash \texttt{UNIT\_SPEC} \rhd \Sigma_1, \ldots, \Sigma_n \to \Sigma \\ UN \notin dom(C_s)\end{array}}{\Gamma_s, C_s \vdash UN \;:\; \texttt{UNIT\_SPEC} \rhd \{UN \mapsto \Sigma_1, \ldots, \Sigma_n \to \Sigma\}}$$

$$\frac{\vdash \texttt{UNIT\_SPEC} \Rightarrow \mathcal{U}}{\Gamma_s, \Gamma_m, C_s, C \vdash UN \;:\; \texttt{UNIT\_SPEC} \rhd C^{\emptyset}[UN/\mathcal{U}]}$$

Figure 8: Simplified basic static and model semantics for unit declarations.

$$\frac{\begin{array}{c}C_s \vdash T_1 \rhd \Sigma_1 \\ C_s \vdash T_2 \rhd \Sigma_2 \\ \Sigma = \Sigma_1 \cup \Sigma_2\end{array}}{C_s \vdash T_1 \textbf{ and } T_2 \rhd \Sigma} \qquad \frac{\begin{array}{c}C \vdash T_1 \Rightarrow UEv_1 \\ C \vdash T_2 \Rightarrow UEv_2 \\ \text{for each } E \in C, \text{ there is a unique } M \in \mathbf{Mod}(\Sigma) \text{ such that} \\ M|_{\Sigma_1} = UEv_1(E) \text{ and } M|_{\Sigma_2} = UEv_2(E) \qquad (\star) \\ UEv = \{E \mapsto M \mid E \in C, M|_{\Sigma_i} = UEv_i(E), \text{for } i = 1, 2\}\end{array}}{C \vdash T_1 \textbf{ and } T_2 \Rightarrow UEv}$$

Figure 9: Basic static and model semantics for unit amalgamation

case, the result of evaluating T_1 **and** T_2 in this unit environment is the amalgamation of these corresponding models. Similarly, Fig. 10 presents the basic static semantics and model semantics for unit application, simplified to the case of units with just one argument. The static semantics produces the signature of the term T and returns as signature of $F[T]$ the selected pushout $\Sigma(\Delta)$ of the span (σ, Δ), where Δ is the unit signature of F stored in the context C_s. In the general case of units with multiple arguments, the signature is obtained with the following definition.

$$\frac{\begin{array}{c}C_s(F) = \Delta : \Sigma \hookrightarrow \Sigma' \\ C_s \vdash T \rhd \Sigma^A \\ \sigma : \Sigma \to \Sigma^A \\ (\iota_{\Sigma^A \subseteq \Sigma'(\Delta)}, \sigma(\Delta), \Sigma'(\Delta)) \text{ is the pushout of } (\Delta, \sigma)\end{array}}{C_s \vdash F[T \text{ fit } \sigma] \rhd \Sigma'(\Delta)} \qquad \frac{\begin{array}{c}C \vdash T \Rightarrow UEv \\ \text{for each } E \in C, UEv(E)|_{\sigma} \in dom(E(F)) \qquad (\diamond) \\ \text{for each } E \in C, \text{ there is a unique } M \in \mathbf{Mod}(\Sigma'(\Delta)) \\ \text{such that} \\ M|_{\iota_{\Sigma^A \subseteq \Sigma'(\Delta)}} = UEv(E) \text{ and } M|_{\sigma(\Delta)} = E(F)(UEv(E)|_{\sigma}) \quad (\star) \\ UEv_R = \{E \mapsto M \mid E \in C, M|_{\iota_{\Sigma^A \subseteq \Sigma'(\Delta)}} = UEv(E), \\ M|_{\sigma(\Delta)} = E(F)(UEv(E)|_{\sigma})\}\end{array}}{C \vdash F[T \text{ fit } \sigma] \Rightarrow UEv_R}$$

Figure 10: Basic static and model semantics for unit application

Definition 4.1. *Let F be a generic unit with the unit signature $\Sigma_1, \ldots, \Sigma_n \to \Sigma$, and let $F[T_1 \textit{ fit } \sigma_1] \ldots [T_n \textit{ fit } \sigma_n]$ be a unit term involving F. Let Σ_i^A be the signature of the unit term T_i, for $i = 1, \ldots n$. Let Σ^F be the union of all Σ_i and Σ^A the union of all Σ_i^A. Let $\Delta : \Sigma^F \to \Sigma$ be the inclusion of Σ^F into Σ. Let $\sigma : \Sigma^F \to \Sigma^A$ be the union of the fitting morphisms $\sigma_i : \Sigma_i \to \Sigma_i^A$. Let $(\Sigma(\Delta), \sigma(\Delta), \iota_{\Sigma^A \subseteq \Sigma(\Delta)})$ be the selected pushout for (σ, Δ):*

$$\begin{tikzcd}
\Sigma^A \arrow[r, hook] & \Sigma(\Delta) \\
\Sigma^F \arrow[u, "\sigma"] \arrow[r, hook, "\Delta"'] & \Sigma \arrow[u, "\sigma(\Delta)"']
\end{tikzcd}$$

Then the signature of the term $F[T_1 \mathit{fit}\ \sigma_1]\ldots[T_n \mathit{fit}\ \sigma_n]$ is $\Sigma(\Delta)$. □

The model semantics first analyzes the argument T in a unit context C recording the constraints on and dependencies between previously introduced units and gives a unit evaluator UEv. Then, provided that ($\diamond$) the actual parameter fits the domain and ($\star$) the models $UEv(E)$ and $E(F)(UEv(E)|_\sigma)$ can be amalgamated to a Σ_R-model M, the result unit evaluator UEv_R gives the amalgamation M for each unit environment $E \in C$. The condition ($\diamond$) will be discharged with the proof calculus for architectural specifications in Sect. 5.1.

Typically one would expect conditions of type ($\star$) to be discarded statically. For this purpose, an *extended static semantics* was introduced in [26], where the dependencies between units are tracked with the help of a diagram of signatures. The idea is that we can now verify that the interpretation of two symbols is the same by looking for a "common origin" in the diagram, i.e. a symbol which is mapped via some paths to both of them. The assumption that generic units are interpreted as *functions* on compatible models requires that a generic unit yields the same result when applied to the same arguments. However, the extended static semantics is sound and complete only w.r.t. a *generative* semantics. This semantics has the characteristic that applying a generic unit multiple times to the same arguments yields results that do not "share". For a non-generative (= applicative) semantics, the calculus is only sound. If all generic units are applied at most once, the generative and the applicative semantics coincide. We denote by $\vdash ASP \Rightarrow_g \mathcal{AM}$ the generative model semantics of architectural specifications. See Sect. III.5.6.6 and IV.5.2.1 of [6] for more motivation and discussion on generative vs. non-generative semantics of architectural specifications. In the following we may refer to the class of models to which a unit term of an architectural specification may evaluate as the class of models of that unit term.

4.4. Static Semantics of Refinements

For the static semantics of refinements, we introduce *refinement signatures*, denoted $R\Sigma$.

Definition 4.2. *A* refinement signature *has the following form:*

$$\begin{array}{rcl} R\Sigma & ::= & (U\Sigma, B\Sigma) \mid \{UN_i \mapsto R\Sigma_i\}_{i\in\mathcal{J}} \\ B\Sigma & ::= & U\Sigma \mid \{UN_i \mapsto B\Sigma_i\}_{i\in\mathcal{J}} \end{array}$$

□

This means that a refinement signature $R\Sigma$ can be either:

S/B a *branching refinement signature* $(U\Sigma, B\Sigma)$ (which also cover signatures for simple refinements) consisting of a unit signature $U\Sigma$ and a *branching signature* $B\Sigma$, which can itself be either

S a unit signature $U\Sigma'$, and in this case $R\Sigma$ is a *simple refinement signature*, or

B a map $\{UN_i \mapsto B\Sigma_i\}_{i\in\mathcal{J}}$ called a *branching static context* and denoted *BstC*, assigning branching signatures to unit names.

The intuition is that a unit refinement signature stores the signatures of a unit before and after refinement, and a branching signature generalizes this to the case when a unit specification is refined to an architectural specification, so that branching is introduced by storing the signatures of all components in a map where they can be retrieved by the corresponding component's name.

C a *component refinement signature* $\{UN_i \mapsto R\Sigma_i\}_{i\in\mathcal{J}}$, storing the refinement signature of each component to be refined in the case of component refinements. When all $R\Sigma_i$, $i \in \mathcal{J}$, are branching refinement signatures $(U\Sigma_i, B\Sigma_i)$, we refer to the component refinement signature $\{UN_i \mapsto R\Sigma_i\}_{i\in\mathcal{J}}$ as a *refined-unit static context*, denoted *RstC*, which can then be naturally coerced to a static context $\pi_1(RstC) = \{UN_i \mapsto U\Sigma_i\}_{i\in\mathcal{J}}$ as well to a branching static context $\pi_2(RstC) = \{UN_i \mapsto B\Sigma_i\}_{i\in\mathcal{J}}$.

Example 4.1. The signature of R1 in Example 2.3 is the simple refinement signature (**Sig**(Monoid), **Sig**(Nat)). The signature of R4 in Example 2.4 is the branching refinement signature (**Sig**(NatWithSuc), $\{N \mapsto$ **Sig**(Nat), $F \mapsto$ **Sig**(Nat) $\rightarrow$ **Sig**(NatWithSuc)$\}$). The signature of RComp in Example 2.5 is $\{N \mapsto$ (**Sig**(Nat), **Sig**(NatBin))$\}$. Assuming we would also want to refine the component F of Addition_First_Ref using some refinement R', we could record this in a component refinement:

refinement RefComp' = { N **to** R2, F **to** R'}

and the signature of RefComp' is then $\{N \mapsto (\mathbf{Sig}(\text{Nat}), \mathbf{Sig}(\text{NatBin})), F \mapsto \mathbf{Sig}(\text{R'})\}$. □

The rules for static semantics of refinements that we give later make use of an auxiliary composition operation between refinement signatures. Intuitively, it can be explained as checking that at each connection point between the two refinement trees, the corresponding unit signatures match. Moreover, the signatures for these connection points are forgotten, and only the "outer" signatures are kept.

Definition 4.3. *Given refinement signatures $R\Sigma_1$ and $R\Sigma_2$, their* composition $R\Sigma_1 \,; R\Sigma_2$ *is defined inductively on the form of the first argument:*

S;S/S;B $R\Sigma_1 = (U\Sigma, U\Sigma')$*: 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 $R\Sigma_1 \,; R\Sigma_2 = (U\Sigma, B\Sigma'')$.*

B;C $R\Sigma_1 = (U\Sigma, BstC')$*: then $R\Sigma_1 \,; R\Sigma_2$ is defined only if $R\Sigma_2$ is a component refinement signature that* matches *$BstC'$, i.e., $dom(R\Sigma_2) \subseteq dom(BstC')$ 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*
  $BstC'[R\Sigma_2](UN) = BstC'(UN)[R\Sigma_2(UN)]$.

C;C *$R\Sigma_1$ 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 $R\Sigma_1 \,; R\Sigma_2$ is the component refinement signature with $dom(R\Sigma_1 \,; R\Sigma_2) = dom(R\Sigma_1) \cup dom(R\Sigma_2)$ that on $UN \in dom(R\Sigma_1) \setminus dom(R\Sigma_2)$ coincides with $R\Sigma_1$, on $UN \in dom(R\Sigma_2) \setminus dom(R\Sigma_1)$ coincides with $R\Sigma_2$ and on $UN \in dom(R\Sigma_1) \cap dom(R\Sigma_2)$ is by definition $R\Sigma_{UN}$.*

□

Example 4.2. With the specifications in Example 2.3, we can compose the signature $(\mathbf{Sig}(\text{Monoid}), \mathbf{Sig}(\text{Nat}))$ of R1 with the signature $(\mathbf{Sig}(\text{Nat}), \mathbf{Sig}(\text{NatBin}))$ of R2 using the case **S;S/S;B** and we obtain $(\mathbf{Sig}(\text{Monoid}), \mathbf{Sig}(\text{NatBin}))$. The branching refinement signature $(\mathbf{Sig}(\text{NatWithSuc}), \{N \mapsto \mathbf{Sig}(\text{Nat}), F \mapsto \mathbf{Sig}(\text{Nat}) \to \mathbf{Sig}(\text{NatWithSuc})\})$ of Addition_First can be composed with the component refinement signature $\{N \mapsto (\mathbf{Sig}(\text{Nat}), \mathbf{Sig}(\text{NatBin}))\}$ of RComp using the case **B;C** and the result is $(\mathbf{Sig}(\text{NatWithSuc}), \{N \mapsto \mathbf{Sig}(\text{NatBin}), F \mapsto \mathbf{Sig}(\text{Nat}) \to \mathbf{Sig}(\text{NatWithSuc})\})$. For the case **C;C**, we consider the following refinement:

refinement CompRef1 = {A **to** R4}

refinement CompRef2 = {A **to** {N **to** R2}}

refinement RefComposition = CompRef1 **then** CompRef2

where CompRef1 states that a component named A of an architectural specification should be refined as described by R4 and CompRef2 states that the unit N of the component A should be refined as described by R2. The signature of CompRef1 is the component refinement signature $\{A \mapsto (\mathbf{Sig}(\text{NatWithSuc}), \{N \mapsto \mathbf{Sig}(\text{Nat}), F \mapsto \mathbf{Sig}(\text{Nat}) \to \mathbf{Sig}(\text{NatWithSuc})\})\}$ and the signature of CompRef2 is $\{A \mapsto \{N \mapsto (\mathbf{Sig}(\text{Nat}), \mathbf{Sig}(\text{NatBin}))\}\}$. The signature of RefComposition is then $\{A \mapsto (\mathbf{Sig}(\text{NatWithSuc}), \{N \mapsto \mathbf{Sig}(\text{NatBin}), F \mapsto \mathbf{Sig}(\text{Nat}) \to \mathbf{Sig}(\text{NatWithSuc})\})\}$. □

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

$$\frac{\vdash ASP \rhd (U\Sigma, RstC)}{\vdash \mathtt{arch\ spec}\, ASP \rhd (U\Sigma, \pi_2(RstC))} \qquad \frac{\begin{array}{c}\vdash SPR_1 \rhd R\Sigma_1 \\ \vdash SPR_2 \rhd R\Sigma_2 \\ R\Sigma = R\Sigma_1 \,; R\Sigma_2\end{array}}{\vdash SPR_1 \textbf{ then } SPR_2 \rhd R\Sigma}$$

$$\frac{\begin{array}{c}UN_1, \ldots, UN_n \text{ are distinct} \\ \vdash SPR_i \rhd R\Sigma_i, i = 1, \ldots, k\end{array}}{\vdash \{UN_1 \ \mathtt{to}\ SPR_1\, , \ldots\, , UN_k \ \mathtt{to}\ SPR_k\} \rhd \{UN_1 \mapsto R\Sigma_1, \ldots, UN_k \mapsto R\Sigma_k\}}$$

Figure 11: Static semantics of CASL refinements

The rules for static semantics of refinements are presented in Fig. 11. The general format of the judgements is $\vdash SPR \rhd R\Sigma$, and the result of the analysis is a refinement signature $R\Sigma$. When we only want to state that the rules apply successfully and the result is not important, we will denote this $\vdash SPR \rhd \Box$. We have a rule for each alternative form of refinement (see non-terminal SPR in Fig. 6) and derivations are built inductively on the structure of the refinement. The first rule requires some explanation: unit specifications are a particular case of refinement, so we introduce a rule that takes the unit signature $U\Sigma$ of the unit specification *USP* and returns as result of the analysis of the unit specification regarded as a refinement specification (this is marked by *USP* qua SPR) the pair $(U\Sigma, U\Sigma)$. In the second rule, if σ is omitted, it is assumed to be the identity signature morphism. The form of judgements and further rules for architectural specifications incorporate a number of necessary changes to the semantics of architectural specifications as originally given in [6], Sect. III.5. This is because now the specifications of the units are either simple or branching refinements (note that the syntax of unit declarations, as introduced in Fig. 4, has been extended in the refinement language, see Fig. 6), and we thus derive a refined-unit static context *RstC* for the units of an architectural specification, as we illustrate here with the rule for the static semantics of unit declarations:

$$\frac{\begin{array}{c}\Gamma_s \vdash SPR \rhd R\Sigma \\ UN \notin Dom(RstC)\end{array}}{\Gamma_s, RstC \vdash UN : SPR \rhd \{UN \mapsto R\Sigma\}}$$

Note that here, as in the case of plain architectural static semantics, we build for each declared unit *UN* a refined-unit static context that maps *UN* to the refinement signature $R\Sigma$ of its specification *SPR*. The rule for unit declarations and definitions unites this refined-unit static context with the one built using the previous declarations and definitions and uses the result of the union to analyze the declarations and definitions following the declaration of *UN*. Comparing with the old rule of the plain architectural language in Fig. 8, notice that we now use a refined-unit static context instead of a static context. We can coerce *RstC* to a static context using the projection to the first component π_1 where required by the rest of plain architectural static semantics. For example, the rule for unit expressions uses judgements of the form $\Gamma_s, C_s \vdash UE \rhd U\Sigma$ and we can apply them in our new setting using $\Gamma_s, \pi_1(RstC) \vdash UE \rhd U\Sigma$. For unit definitions, we do not want to allow the defined unit to be further refined, and we use the $\perp$ sign to mark that:

$$\frac{\begin{array}{c}\Gamma_s, \pi_1(RstC) \vdash UE \rhd U\Sigma \\ UN \notin Dom(RstC)\end{array}}{\Gamma_s, RstC \vdash UN = UE \rhd \{UN \mapsto (U\Sigma, \perp)\}}$$

This ensures that the composition (defined below) of a branching refinement signature with a component refinement signature having the name of a defined unit in its domain is illegal. The needed extension of the notion of refinement signature to signatures with $\perp$ in the second component (for unit definitions) is straightforward. Since it is needed only for the special purpose of preventing further composition, we have chosen to not include it in the definition of refinement signatures (Def. 4.2), thereby reducing the number of cases to be analyzed. The other rules in the static semantics of architectural specifications can be modified in a similar way; since this is rather obvious, using the projection π_1, we will not present this in detail here. The signature of the result unit expression of the architectural specification is then paired with the projection on the second component of the refined-unit static context to obtain a branching signature which is then given as result of the analysis. Finally, the complexity of the rule for refinement composition is hidden in the definition of composition of refinement signatures.

4.5. Model Semantics of Refinements

For the model semantics of refinements, we first introduce the notion of constructor implementation [27, 3]. Constructors are simply partial functions taking models to models, of the form $\kappa : \mathbf{Mod}(\Sigma) \to \mathbf{Mod}(\Sigma')$. An example of such a constructor is already provided by the model reduct along a signature morphism $\sigma : \Sigma \to \Sigma'$, which takes any Σ'-model M' to its Σ-reduct $M'|_\sigma$.

Definition 4.4. *[27] Let SP, SP' be specifications such that $\vdash SP \rhd \Sigma$ and $\vdash SP' \rhd \Sigma'$ and let $\kappa : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ be a constructor. We say that SP' is a* constructor implementation *of SP* along *κ, denoted $SP \overset{\kappa}{\rightsquigarrow} SP'$, if for all $M' \in \mathbf{Mod}(SP')$, $M' \in dom(\kappa)$ and $\kappa(M') \in \mathbf{Mod}(SP)$.* □

Simple refinements along a signature morphism $\sigma : \Sigma \to \Sigma'$ correspond to constructors where the function κ is given by the action of $\mathbf{Mod}(\sigma)$ on objects.

To capture branchings, we need n-ary constructors of the form $\kappa : \mathbf{Mod}(\Sigma_1)\times\ldots\times\mathbf{Mod}(\Sigma_n) \to \mathbf{Mod}(\Sigma)$, which are partial functions mapping *compatible* models over $\Sigma_1, \ldots, \Sigma_n$ to Σ-models, where the compatibility requirement means that the arguments can be amalgamated to a model of the union of signatures $\Sigma_1, \ldots, \Sigma_n$. An implementation is now correct if given Σ_i-specifications SP_i, for $i = 1, \ldots, n$, any tuple of compatible models $M_1, \ldots, M_n$ of $SP_1, \ldots, SP_n$, respectively, are in the domain of κ and $\kappa(M_1, \ldots, M_n)$ is a model of the specification SP which we want to refine to $SP_1, \ldots, SP_n$. These constructors are specified using branching specification refinements.

Constructors then provide an intuitive notion of *refinement model*. This is easiest to see in the case of unary constructors, for refinements of type **S**: if $SP \overset{\kappa}{\rightsquigarrow} SP'$, we denote by $\mathcal{G}$ the graph of κ restricted to $\mathbf{Mod}(SP')$, that is, pairs of the form $(M', \kappa(M'))$, where $M' \in \mathbf{Mod}(SP')$. We then take the inverse relation $\mathcal{G}^{-1}$ to obtain on the first component models over $\mathbf{Sig}(SP)$ and on the second component models over $\mathbf{Sig}(SP')$, thus matching the order of models with the order of their unit signatures in the corresponding refinement signature. This generalizes to n-ary constructors to cover branching refinement and to families of constructors, which will be models of component refinement specifications. Overall, this leads us to the following formal definition.

Definition 4.5. *Given a refinement signature $R\Sigma$, an* $R\Sigma$-refinement relation*, usually denoted $\mathcal{R}$, is a class of $R\Sigma$-*assignments*, usually denoted R, which can themselves be:*

S/B branching assignments*: for $R\Sigma = (U\Sigma, B\Sigma')$, these are pairs (U, BM'), where (1) U is a unit over the unit signature $U\Sigma$ and (2) BM' is a* branching model *over the branching signature $B\Sigma'$, i.e (2.1) a unit over $B\Sigma'$ when $B\Sigma'$ is a unit signature (in which case the branching assignment is a* unit assignment*), or (2.2) 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. Moreover, we require that whenever two branching assignments (U, BM) and (U', BM) are in $\mathcal{R}$ then $U = U'$ (so that the corresponding constructor is a function, which in general is partial).*

C component assignments*: for $R\Sigma = \{UN_i \mapsto R\Sigma_i\}_{i\in\mathcal{J}}$, these 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 $\pi_2(RE) = \{UN_i \mapsto BM_i\}_{i\in\mathcal{J}}$.*

□

Example 4.3. We illustrate the concept of refinement relation using the specifications in Example 2.3. The signature of R1 is $(\mathbf{Sig}(\text{MONOID}), \mathbf{Sig}(\text{NAT})$ and a refinement relation over this signature is $\{(N|_\sigma, N) \mid N \in \mathbf{Mod}(\text{NAT})\}$ where $\sigma : \mathbf{Sig}(\text{MONOID}) \to \mathbf{Sig}(\text{NAT})$ is the unique signature morphism taking the sort *Elem* to *Nat*. Note that the class $\{(M, N) \mid M \in \mathbf{Mod}(\text{MONOID}), N \in \mathbf{Mod}(\text{NAT})\}$ is not a refinement relation, as it contains for each model $N \in \mathbf{Mod}(\text{NAT})$ pairs (M, N) and (M', N) with M and M' being two different models of MONOID.

Let T be the term model of NATWITHSUC and T' its reduct to the signature of NAT. The function γ mapping T' to T is a unit in $Unit(\text{NAT} \to \text{NATWITHSUC})$. A branching assignment over the branching refinement signature $(\mathbf{Sig}(\text{NATWITHSUC}), \{N \mapsto \mathbf{Sig}(\text{NAT}), F \mapsto \mathbf{Sig}(\text{NAT}) \to \mathbf{Sig}(\text{NATWITHSUC})\})$ is then $(T, \{N \mapsto T', F \mapsto \gamma\})$.

A component assignment over the signature $\{N \mapsto (\mathbf{Sig}(\text{NAT}), \mathbf{Sig}(\text{NATBIN}))\}$ of RCOMP is $\{N \mapsto \{(B|_{\sigma'}, B) \mid B \in \mathbf{Mod}(\text{NATBIN})\}\}$ where $\sigma' : \mathbf{Sig}(\text{NAT}) \to \mathbf{Sig}(\text{NATBIN})$ is the signature morphism mapping the sort *Nat* to *Bin*, which is the identity on the operations.

□

Similarly to the static semantics, we define an auxiliary partial operation to compose refinement relations. The intuition is that at each refinement step we further restrict the domain of a constructor by composition, thus narrowing the class of acceptable realizations, which is the image of the constructor.

S;S/S;B The intuition here is that the image I of the second constructor should be in the domain of the first constructor. The composition has then the domain of the second constructor as its domain and the restriction of the image of the first constructor to the image of I along the constructor as its image.

B;C The intuition here is that if we take a branching environment BE in the domain of $\mathcal{R}_1$ and replace, for each unit UN in the domain of the second constructor $\mathcal{R}_2$, the unit $BE(UN)$ with a unit delivered by the second constructor, we get a new branching environment $BE[R_2]$ which should still be in the domain of $\mathcal{R}_1$. For each such unit name UN the domain of the composition is now determined by the domain of $\mathcal{R}_2(UN)$.

C;C Here, we require that for each unit name UN in the domains of $\mathcal{R}_1$ and $\mathcal{R}_2$, the image of $\mathcal{R}_2(UN)$ should be in the domain of $\mathcal{R}_1(UN)$. This might require several recursive applications of the definition, until $\mathcal{R}_1(UN)$ is no longer a component refinement, which means we have navigated through the tree of the first refinement until we reached a leaf. We can then apply one of the two rules above to obtain the modified domain and range of the composition for UN.

Definition 4.6. *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:*

S;S/S;B *$R\Sigma_1 = (U\Sigma, U\Sigma')$, $R\Sigma_2 = (U\Sigma', B\Sigma'')$: 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*

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

B;C *$R\Sigma_1 = (U\Sigma, B\Sigma)$ and $R\Sigma_2$ is a component refinement signature that matches $B\Sigma$: then $\mathcal{R}_1 ; \mathcal{R}_2$ is defined only if for each (U, BE) in $\mathcal{R}_1$ and each $R_2 \in \mathcal{R}_2$ there exists $U' \in Unit(U\Sigma)$ such that $(U', BE[R_2]) \in \mathcal{R}_1$, where $BE[R_2]$ modifies BE on each $UN \in dom(R_2)$ as follows:*

- *if $BstC'(UN)$ is a unit signature then $BE[R_2](UN) = U''$, where $R_2(UN) = (U'', BE'')$;*
- *if $BstC'(UN)$ is a branching static context then we put*
  $BE[R_2](UN) = BE(UN)[R_2(UN)]$.

Then

$$\mathcal{R}_1;\mathcal{R}_2 = \{(U', BE\langle R_2\rangle) \mid (U, BE) \in \mathcal{R}_1, R_2 \in \mathcal{R}_2\}$$

where U' is determined by the requirement that $(U', BE[R_2])$ should be in $\mathcal{R}_1$ and $BE\langle R_2\rangle$ modifies BE on each $UN \in dom(R_2)$ as follows:

- *if $BstC'(UN)$ is a unit signature then $BE\langle R_2\rangle(UN) = BE''$ where $R_2(UN) = (U'', BE'')$;*
- *if $BstC'(UN)$ is a branching static context then we put $BE\langle R_2\rangle(UN) = BE(UN)\langle R_2(UN)\rangle$.*

C;C *$R\Sigma_1$ and $R\Sigma_2$ are component refinement signatures such that for all $UN \in dom(R\Sigma_1) \cap dom(R\Sigma_2)$, $R\Sigma_{UN} = R\Sigma_1(UN)\,;R\Sigma_2(UN)$; then $\mathcal{R}_1;\mathcal{R}_2$ is defined iff $\mathcal{R}_2$ matches $\mathcal{R}_1$, which means that for each $R_1 \in \mathcal{R}_1$ and for each $R_2 \in \mathcal{R}_2$ we have that for each $UN \in dom(\mathcal{R}_1) \cap dom(\mathcal{R}_2)$:*

- *if $R_2(UN) = (U', BM)$, there exists $(U, U') \in \mathcal{R}_1(UN)$, where $\mathcal{R}_1(UN) = \{R(UN) \mid R \in \mathcal{R}_1\}$,*
- *if $R_2(UN)$ is a component assignment,*
  - *if $R_1(UN) = (U, BE)$ then there exists $(U', BE[R_2(UN)]) \in \mathcal{R}_1(UN)$, where $BE[R_2(UN)]$ is defined as in the second case of the main definition,*
  - *if $R_1(UN)$ is a component assignment, then $\mathcal{R}_2(UN)$ matches $\mathcal{R}_1(UN)$.*

When defined, $\mathcal{R}_1\,;\mathcal{R}_2$ is the class of all assignments $R_1\langle R_2\rangle$ where $R_1 \in \mathcal{R}_1$, $R_2 \in \mathcal{R}_2$ and we define $R_1\langle R_2\rangle$ as the assignment with $dom(R_1\langle R_2\rangle) = dom(R_1) \cup dom(R_2)$ that on $UN \in dom(R_1) \setminus dom(R_2)$ coincides with R_1, on $UN \in dom(R_2) \setminus dom(R_1)$ coincides with R_2, and on $UN \in dom(R_1) \cap dom(R_2)$

- *if $R\Sigma_1(UN)$ is a unit refinement signature, and $R_2(UN) = (U', BM'')$, then by the matching condition there must exist $(U, U') \in \mathcal{R}_1(UN)$. The uniqueness of U is ensured by the definition of assignments. Then $R_1\langle R_2\rangle(UN) = (U, BM'')$;*
- *if $R\Sigma_1(UN)$ is a branching refinement signature, and $R_1(UN) = (U, BE)$, then we define $R_1\langle R_2\rangle(UN) = (U', BE\langle R_2(UN)\rangle)$ where U' is uniquely determined by the matching condition, which in this case means that $BE[R_2(UN)]$ is still in the domain of the constructor given by $\mathcal{R}_1(UN)$, and $BE\langle R_2(UN)\rangle$ is as defined in the second case of the main definition;*
- *if $R\Sigma_1(UN)$ is a component refinement signature, then $R_1\langle R_2\rangle(UN) = R_1(UN)\langle R_2(UN)\rangle$.*

□

The model semantics of refinements is presented in Fig. 12. The judgements are now of the form $\vdash SPR \Rightarrow \mathcal{R}$, where SPR is a refinement and $\mathcal{R}$ is a refinement relation and they are derived by induction on the structure of the refinements, using the rules determined by the form of the refinement concerned. Again, we need to modify the model semantics for architectural specifications as given in [6], Sect. III.5. We build a class of refined-unit environments RE for the units of an architectural specification that we can coerce to a unit environment using the projection to the first component π_1. Then, the result unit expression is analysed using the rules of the plain architectural model semantics of the form $\Gamma_s, \Gamma_m, C_s, C \vdash UE \Rightarrow U$ in our new setting using $\Gamma_s, \Gamma_m, \pi_1(RstC), \pi_1(RE) \vdash UE \Rightarrow U$, where $RstC$ is obtained with the rules of the static semantics. Finally, the result of the analysis of the architectural specification is the class of all $(U, \pi_2(RE))$.

Using the model semantics of refinements, we can now define:

Definition 4.7. *A refinement SPR is* consistent *if its denotation $\mathcal{R}$ is a non-empty class of assignments.* □

Remark 4.1 (Refinement of arbitrary unit types)**.** Given unit specifications $SP \rightarrow SP'$ and $SP_1 \rightarrow SP'_1$ with a specification morphism $\sigma : SP_1 \rightarrow SP$, the following is a correct specification refinement:[15]

unit spec $USP = SP \rightarrow SP'$
unit spec $USP' = SP_1 \rightarrow SP'_1$
refinement R = USP **refined via** τ **to arch spec** {
    units F : USP'
    result lambda $X : SP \bullet$ F [X **fit** σ]}

[15]We assume that all symbols shared between SP'_1 and SP originate in SP_1, as imposed by CASL rules for application of generic units. In particular, if the pushout is amalgamable (as is always the case in so-called semi-exact institutions), then static correctness of the application of F is ensured.

$$\frac{\vdash USP \Rightarrow \mathcal{U}}{\vdash USP\ \text{qua}\ \mathtt{SPR} \Rightarrow \{(U,U) \mid U \in \mathcal{U}\}}$$

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

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

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

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

Figure 12: Model semantics of CASL refinements

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

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

where the square is the selected pushout of the diagram formed by the signatures of the specifications and $SP'_1 \oplus SP$ is the specification $\{SP'_1$ **with** $\sigma'\}$ **and** $\{SP$ **with** $\theta'\}$. □

5. Calculi for Refinements

An important motivation for formalizing the development process using CASL architectural specifications and refinements is that one can then formally *prove* correctness of the entire development. In [6], Sect. IV:5, a proof calculus for verification of architectural specifications was introduced as an algorithm for checking whether the resulting units of an architectural specification satisfy a given unit specification. In order to simplify presentation, in [6] the architectural language was restricted as in Fig. 13; however, it is rather straightforward to generalize the proof calculus to the whole architectural language, with the notable exception of unit imports.

In the following we will start by presenting a new proof calculus for architectural specifications, in Sect. 5.1. The motivation for introducing another proof calculus is that it covers the entire architectural language, including unit imports in a natural and easy way, as we will show in Sect. 5.2, and it can be more easily generalized in the context of the refinement language, where the specifications of units of an architectural specifications can be refinements. We present this generalization in Sect. 5.3. In Sect. 5.4 we discuss completeness of the proof calculus for refinements and in Sect. 5.5 we introduce a calculus for checking consistency of refinements.

ASP ::= ~~S |~~ **units** UDD$_1$ ... UDD$_n$ **result** UE
UDD ::= ~~UDEFN |~~ UDECL
UDECL ::= *UN* : USP ~~⟨**given** UT$_1$, ..., UT$_n$⟩~~
USP ::= SPEC | SPEC$_1$ ~~$\times \cdots \times$ SPEC$_n$~~ $\rightarrow$ SPEC ~~| **arch spec** ASP~~
UDEFN ::= A = UT ~~UE~~
UE ::= UT | λ A_1 : SPEC$_1$, ~~..., A_n : SPEC$_n$~~ • UT
UT ::= A | A [FIT$_1$] ~~... [FIT$_n$]~~ | UT **and** UT | UT **with** $\sigma : \Sigma \rightarrow \Sigma'$ |
  UT **hide** $\sigma : \Sigma \rightarrow \Sigma'$ | **local** UDEFN$_1$ ~~... UDEFN$_n$~~ **within** UT
FIT ::= UT | UT **fit** $\sigma : \Sigma \rightarrow \Sigma'$

Figure 13: Restricted CASL architectural language as in [6].

5.1. Proof Calculus for Architectural Specifications

We first recall the existing calculus of [6] and begin by introducing a number of auxiliary concepts. A *context* Γ is a diagram in the signature category of the given institution I, whose nodes are additionally labeled by sets of specifications, which in the calculus will be either empty or singleton sets. We will write $A :_\Sigma \mathcal{SP}$ to denote that a node A of a context is labeled with the signature Σ and the set $\mathcal{SP}$ of Σ-specifications. We use $A : \Sigma$ to denote that the signature of the node A is Σ. Finally, we use $\sigma\colon A \rightarrow B$ to denote an edge between the nodes $A : \Sigma_1$ and $B : \Sigma_2$ labeled with $\sigma : \Sigma_1 \rightarrow \Sigma_2$. Thus, we can regard contexts as sets of such declarations of labeled nodes and edges. Given a context Γ and a family of models $\mathcal{M} = \{M_p\}_{p \in Nodes(\Gamma)}$ indexed by the nodes of Γ, we say that $\mathcal{M}$ is *compatible with* Γ if for each $A :_\Sigma \mathcal{SP}$ in Γ, M_A is a Σ-model such that $M_A \in \mathbf{Mod}(SP)$, for each $SP \in \mathcal{SP}$, and $M_A = M_B|_\sigma$ for each edge $\sigma\colon A \rightarrow B$. A *generic context* Γ_{gen} is a finite set of declarations of the form $A :_{\Sigma \rightarrow \Sigma'} SP \rightarrow SP'$, where $\Sigma \rightarrow \Sigma'$ is the unit signature of $SP \rightarrow SP'$.

The proof calculus of [6] is then given in Figs. 14 and 15, together with the general format of the judgements (in the box preceding the rules for each judgement). The main judgement is of the form $\vdash ASP :: USP$, where *ASP* is an architectural specification over an institution I and *USP* is a given unit specification. $\vdash ASP :: USP$ asserts that each result unit that can be built following the architectural specification *ASP* satisfies the unit specification *USP*.

The proof calculus can be regarded as having two components. The first one is a *constructive* component, building a context Γ to keep track of the dependencies between units as well a generic context Γ_{gen} to store the generic units. This is done with the rules for unit declarations *UDECL* and unit terms *UT* in the proof calculus below. The second component is *deductive* and it uses the contexts Γ and Γ_{gen} built with the constructive component to check whether models of a unit expression satisfy a given unit specification *USP*. This component contains the rules for architectural specifications *ASP* and unit expression *UE* in the proof calculus below. For simplicity, we have chosen to express the arising proof obligations only semantically; [6] provides means to express them syntactically, using specifications assigned to the nodes in Γ, and with a further calculus to discharge them.

The theorem below states that the proof calculus is successful for a given unit specification if and only if the architectural specification is correct w.r.t. the generative model semantics and the units produced with it satisfy the unit specification. It follows easily from a result in [6].

Theorem 5.1. Let *ASP* be an architectural specification such that $\vdash ASP \rhd \Box$ and no generic unit declaration of *ASP* is inconsistent. For each unit specification *USP* we have that $\vdash ASP :: USP$ if and only if $\vdash ASP \Rightarrow_g \mathcal{AM}$ for some $\mathcal{AM}$ such that for all $(U, BM) \in \mathcal{AM}$, $U \in Unit[USP]$. □

We show that the second assumption of Thm. 5.1 is indeed necessary.

Example 5.1. Let us consider the following specifications:

spec Consts =
  sort s
  ops a,b : s
spec EqConsts = Consts
  then
  • $a = b$

$$\boxed{\begin{array}{c}\vdash ASP :: USP\\ \text{(deductive)}\end{array}}$$

$$\frac{\begin{array}{c}\vdash UDECL_1 :: \Gamma^1_{gen}, \Gamma^1\\ \vdots\\ \vdash UDECL_n :: \Gamma^n_{gen}, \Gamma^n\\ \bigcup_{i=1,\ldots,n}\Gamma^i_{gen}, \bigcup_{i=1,\ldots,n}\Gamma^i \vdash UE :: USP\end{array}}{\vdash \textbf{units}\ UDECL_1 \ldots UDECL_n\ \textbf{result}\ UE :: USP}$$

$$\boxed{\begin{array}{c}\vdash UDECL :: \Gamma_{gen}, \Gamma\\ \text{(constructive)}\end{array}}$$

$$\frac{}{\vdash A : SP :: \emptyset, \{A :_{\mathbf{Sig}[SP]} \{SP\}\}} \qquad \frac{}{\vdash A : SP \to SP' :: \{A :_{\mathbf{Sig}[SP]\to\mathbf{Sig}[SP']} SP \to SP'\}, \emptyset}$$

$$\boxed{\begin{array}{c}\Gamma_{gen}, \Gamma \vdash UE :: USP\\ \text{(deductive)}\end{array}}$$

$$\frac{\begin{array}{c}\Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A\\ \text{for any } \mathcal{M} \text{ compatible with } \Gamma',\ M_A \in \mathbf{Mod}(SP)\end{array}}{\Gamma_{gen}, \Gamma \vdash UT \text{ qua } UE :: SP} \qquad \frac{\begin{array}{c}\mathbf{Sig}[SP_1] = \mathbf{Sig}[SP] = \Sigma\\ SP \text{ and } SP_1 \text{ are equivalent}\\ \Gamma_{gen}, \Gamma \cup \{A :_\Sigma \{SP\}\} \vdash UT :: \Gamma', B\\ B : \mathbf{Sig}[SP_2] \text{ in } \Gamma'\\ \text{for any } \mathcal{M} \text{ compatible with } \Gamma',\ M_B \in \mathbf{Mod}(SP_2)\end{array}}{\Gamma_{gen}, \Gamma \vdash \lambda A : SP \bullet UT :: SP_1 \to SP_2}$$

Figure 14: Proof calculus for CASL architectural specifications (continued in Fig. 15).

spec DiffConsts = Consts
    then
    • ¬ ($a = b$)
arch spec ASP_NAME =
    units
        U : EqConsts;
        F : DiffConsts → EqConsts;
    result F[U]

The unit specification DiffConsts → EqConsts is inconsistent, because the extension is obviously non-conservative: we cannot construct a model in which the interpretation of the constants a and b are equal from a model where they are different in such a way that the argument model is preserved. Therefore, ASP_NAME denotes the empty class of models. Hence, the right hand-side of the equivalence in Thm. 5.1 holds for any unit specification *USP* over the signature of Consts. However, according to the rule for unit applications, F[U] is correct only if for any family of models $\mathcal{M}$ compatible with the diagram of ASP_NAME, M_U is a model of DiffConsts. For any such family $\mathcal{M}$ we have however by construction of the diagram of ASP_NAME that M_U is a model of EqConsts, which cannot be at the same time a model of DiffConsts. Therefore the left hand-side of the equivalence is false for any unit specification

$$\boxed{\begin{array}{c}\Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A \\ \text{(constructive)}\end{array}}$$

$$\frac{}{\Gamma_{gen}, \Gamma \vdash A :: \Gamma, A}$$

$$\frac{\begin{array}{c} A :_{\Sigma_f \to \Sigma_r} SP \to SP_r \text{ in } \Gamma_{gen} \\ \Gamma_{gen}, \Gamma \vdash UT :: \Gamma_a, A_a \\ \text{for any } \mathcal{M} \text{ compatible with } \Gamma_a,\ M_{A_a} \in \mathbf{Mod}(SP_f) \\ (\Sigma(\Delta), \sigma(\Delta), \iota) \text{ is the selected pushout of } (\sigma, \iota_{\Sigma_f \subseteq \Sigma_r}) \\ A_f, A_r, B \notin \mathrm{dom}(\Gamma_a) \end{array}}{\begin{array}{l} \Gamma_{gen}, \Gamma \vdash A\ [\ UT\ \mathbf{fit}\ \sigma : \Sigma_f \to \Sigma_a\ ] :: \Gamma_a \cup \{A_f :_{\Sigma_f} \{SP\}, \sigma : A_f \to A_a, \\ \quad A_r :_{\Sigma_r} \{SP_r\},\ \iota_{\Sigma_f \subseteq \Sigma_r} : A_f \to A_r,\ B :_{\Sigma(\Delta)} \emptyset,\ \iota : A_a \to B,\ \sigma(\Delta) : A_r \to B\}, B \end{array}}$$

$$\frac{\begin{array}{c} \Gamma_{gen}, \Gamma \vdash UT_1 :: \Gamma_1, A_1 \\ \Gamma_{gen}, \Gamma \vdash UT_2 :: \Gamma_2, A_2 \\ A_1 : \Sigma_1 \text{ in } \Gamma_1 \\ A_2 : \Sigma_2 \text{ in } \Gamma_2 \\ dom(\Gamma_1) \cap dom(\Gamma_2) = dom(\Gamma) \\ B \notin dom(\Gamma_1) \cup dom(\Gamma_2) \end{array}}{\begin{array}{l} \Gamma_{gen}, \Gamma \vdash UT_1\ \mathbf{and}\ UT_2 :: \\ \quad \Gamma_1 \cup \Gamma_2 \cup \\ \quad \{B :_{\Sigma_1 \cup \Sigma_2} \emptyset,\ \iota_{\Sigma_1 \subseteq \Sigma_1 \cup \Sigma_2} : A_1 \to B,\ \iota_{\Sigma_2 \subseteq \Sigma_1 \cup \Sigma_2} : A_2 \to B\}, B \end{array}}$$

$$\frac{\begin{array}{c} \Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A \\ B \notin dom(\Gamma') \\ \sigma : \Sigma \to \Sigma' \end{array}}{\begin{array}{l} \Gamma_{gen}, \Gamma \vdash UT\ \mathbf{with}\ \sigma :: \\ \quad \Gamma' \cup \{B :_{\Sigma'} \emptyset, \sigma : A \to B\}, B \end{array}}$$

$$\frac{\begin{array}{c} \Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A \\ B \notin dom(\Gamma') \\ \sigma : \Sigma \to \Sigma' \end{array}}{\begin{array}{l} \Gamma_{gen}, \Gamma \vdash UT\ \mathbf{hide}\ \sigma :: \\ \quad \Gamma' \cup \{B :_{\Sigma} \emptyset, \sigma : B \to A\}, B \end{array}}$$

$$\frac{\begin{array}{c} \Gamma_{gen}, \Gamma \vdash UT :: \Gamma', B \\ B : \Sigma \text{ in } \Gamma' \\ A \notin dom(\Gamma') \\ \Gamma_{gen}, \Gamma' \cup \{A :_{\Sigma} \emptyset, \mathrm{id}_{\Sigma} : A \to B\} \vdash UT' :: \Gamma'', E \\ D \notin dom(\Gamma'') \end{array}}{\begin{array}{l} \Gamma_{gen}, \Gamma \vdash \mathbf{local}\ A = UT\ \mathbf{within}\ UT' :: \\ \quad \Gamma''[D/A], E[D/A] \end{array}}$$

Figure 15: Rules for unit terms.

USP. This shows that the requirement that generic units be consistent is needed in Thm. 5.1. □

In the context of the entire architectural language, we modify the proof calculus such that it becomes fully constructive. Instead of checking that the units produced following an architectural specification *ASP* satisfy a given unit specification *USP*, we define the specification $\mathcal{S}_{\mathrm{ASP}}(UE)$ of the result unit expression *UE* of *ASP* inductively on the structure of *UE*. Clearly, the specification of a unit term given by a declared unit name can be read off from the declaration. If a unit term is translated, then its specification is translated as well, that is, $\mathcal{S}_{\mathrm{ASP}}(UT\ \mathbf{with}\ \sigma) = \mathcal{S}_{\mathrm{ASP}}(UT)\ \mathbf{with}\ \sigma$. Similarly for hiding. A complication arises when determining the unit specification of a unit amalgamation or unit application. The naïve solution, to define $\mathcal{S}_{\mathrm{ASP}}(UT_1\ \mathbf{and}\ UT_2)$ to be $\mathcal{S}_{\mathrm{ASP}}(UT_1)\ \mathbf{and}\ \mathcal{S}_{\mathrm{ASP}}(UT_2)$, does not work. We illustrate this with an example.

Example 5.2. Let us consider the following architectural specification:

arch spec ASP_NAME =
  units
    UN : **sort** s;
    $UT = (UN$ **with** $s \mapsto t)$ **and** $(UN$ **with** $s \mapsto u)$
  result UT
end

Then $\mathcal{S}_{\text{ASP\_NAME}}(UN) = ($**sort** $s; \emptyset)$. The naïve definition of $\mathcal{S}_{\text{ASP\_NAME}}(UT)$ would be $($**sorts** $t, u; \emptyset)$. Now a model of ASP_NAME is a pair (U, BM) where $BM(UN)$ is a model of $($**sort** $s; \emptyset)$ and $BM(UT)$ interprets both the sort t and the sort u as $BM(UN)_s$. This requirement is not recorded in $($**sorts** $t, u; \emptyset)$; hence, this specification is too weak. In order to express that sorts t and u must be interpreted in the same way, we need to look at the following diagram:

$$\begin{tikzcd}
s \arrow[r, "s\mapsto t"] \arrow[d, "s\mapsto u"'] & t \arrow[d] & \\
u \arrow[r] & t,u \arrow[r, dotted, "t,u\mapsto s", "\eta_{UT}"'] & s
\end{tikzcd}$$

Now $\{($**sorts** $t, u; \emptyset)$ **and** $($**sort** $s, \emptyset)\}$ **hide** $t, u \mapsto s$ is the desired specification; it requires that sorts t and u are interpreted in the same way. □

In general, we will need to keep track of the dependencies between symbols, using a weakly amalgamable cocone of the diagram of the unit term. The existence of the latter is ensured by a successful run of the extended static semantics. This is captured by the following definition:

Definition 5.1. *Let ASP be an architectural specification and let Γ denote the context constructed by the rules of the proof calculus to track dependencies between units. For each unit term UT of ASP, we denote by D_{UT} the sub-diagram of Γ corresponding to the node of UT.*[16] *Then we define $\mathcal{S}_{amalg}(UT) = (\Sigma, \emptyset)$ **hide** η_A where A is the node of the unit term UT in D_{UT} and $\eta = (\Sigma, \{\eta_X\}_{X \in dom(D_{UT})})$ is a weakly amalgamable cocone for D_{UT}.* □

We make use of an auxiliary structure for storing the specifications of the units declared and defined in an architectural specification.

Definition 5.2. *A* verification context *is a finite map Γ_v assigning unit specifications to unit names. We denote the empty verification context by $\Gamma_v^{\emptyset}$.* □

The specification of a unit expression is then defined relative to a verification context.

Definition 5.3. *Let Γ_v be a verification context and UE be a unit expression. Then the* specification of *UE w.r.t. Γ_v, denoted $\mathcal{S}_{\Gamma_v}(UE)$, is defined as follows:*

- *if UE is a unit term UT, then $\mathcal{S}_{\Gamma_v}(UT)$ is defined inductively:*
  - *if UT is a unit name, then $\mathcal{S}_{\Gamma_v}(UT) = SP$ where $\Gamma_v(UT) = SP$;*
  - *if $\mathcal{S}_{\Gamma_v}(UT) = SP$, then $\mathcal{S}_{\Gamma_v}(UT$ **with** $\sigma) = SP$ **with** σ;*
  - *if $\mathcal{S}_{\Gamma_v}(UT) = SP$, then $\mathcal{S}_{\Gamma_v}(UT$ **hide** $\sigma) = SP$ **hide** σ;*
  - *if $UT = UT_1$ **and** UT_2 and $\mathcal{S}_{\Gamma_v}(UT_i) = SP_i$ for $i = 1, 2$ then $\mathcal{S}_{\Gamma_v}(UT) =$ SP_1 **and** SP_2 **and** $\mathcal{S}_{amalg}(UT)$;*
  - *if $UT = F[UT_1$ **fit** $\sigma_1] \ldots [UT_n$ **fit** $\sigma_n]$, where $\Gamma_v(F) = SP_1 \times \cdots \times SP_n \to SP$ and for $i = 1, \ldots, n$, the verification condition SP_i **with** $\sigma_i \rightsquigarrow\!\!\!\!\!> \mathcal{S}_{\Gamma_v}(UT_i)$ holds, then $\mathcal{S}_{\Gamma_v}(UT) = \{SP$ **with** $\sigma\}$ **and** $\mathcal{S}_{\Gamma_v}(UT_1)$ **with** $\iota_1; \iota$ **and** ... **and** $\mathcal{S}_{\Gamma_v}(UT_n)$ **with** $\iota_n; \iota$ **and** $\mathcal{S}_{amalg}(UT)$, where the application is done as in the diagram below, with $\Sigma_i =$ **Sig**$[SP_i]$ $\Sigma_i^a =$ **Sig**$[UT_i]$, $\Sigma_f = \cup_{i=1,\ldots,n}\Sigma_i$, $\Sigma_r =$ **Sig**$[SP]$, $\Sigma_a = \cup_{i=1,\ldots,n}\Sigma_i^a$, Δ and ι and all ι_i are inclusions, $\sigma = \cup_{i=1,\ldots,n}\sigma_i : \Sigma_f \to \Sigma_a$, and $(\Sigma_a(\Delta), \iota, \sigma(\Delta))$ is the selected pushout of (Δ, σ) (see Def. 4.1) :*

[16] This can be understood as restricting the declarations and definitions of the units of *ASP* to those involved in *UT*. Then, D_{UT} is the context built by rules of the proof calculus for the restricted architectural specification.

$$\begin{tikzcd}
\Sigma_i^a \arrow[r, hook, "\iota_i"] & \Sigma_a \arrow[r, hook, "\iota"] & \Sigma_a(\Delta) \\
\Sigma_i \arrow[r, hook] \arrow[u, "\sigma_i"] & \Sigma_f \arrow[r, hook, "\Delta"'] \arrow[u, "\sigma"] & \Sigma_r \arrow[u, "\sigma(\Delta)"']
\end{tikzcd}$$

– $\mathcal{S}_{\Gamma_v}$(**local** *UDEFN* **within** *UT*) = $\mathcal{S}_{\Gamma'_v}(UT)$, *where* Γ'_v *extends* Γ_v *according to UDEFN in the obvious way.*

- *if UE is a lambda expression* $\lambda X : SP.\ UT$, *then* $\mathcal{S}_{\Gamma_v}(UE) = SP \to \mathcal{S}_{\Gamma'_v}(UT)$, *where* Γ'_v *extends* Γ_v *with* $X : SP$.

$\square$

If Γ_v has been built from all the units declared and defined in an architectural specification *ASP*, we may denote the specification of a unit expression *UE* by $\mathcal{S}_{ASP}(UE)$ instead of $\mathcal{S}_{\Gamma_v}(UE)$. Moreover, we denote by $\mathcal{S}_{ASP}$ the specification of the result unit expression in *ASP*.

If $\vdash UT \rhd \Sigma_{UT}$ and no non-generic unit is used more than once in *UT*, directly or indirectly via unit definitions, we have that $\mathcal{S}_{amalg}(UT) = \Sigma_{UT}$ **hide** $id_{\Sigma_{UT}}$. This allows us to omit $\mathcal{S}_{amalg}(UT)$ from $\mathcal{S}_{\text{ASP\_NAME}}(UT)$ in such cases.

In general, the specification of a unit term does not give an exact axiomatization of the class of models of the unit term. The reason is that non-generativity of Casl architectural semantics cannot always be captured by a structured specification, and this becomes visible when the unit term involves more than one application of the same generic unit, as we can see from Example 5.3 below. However, we will use $\mathcal{S}_{\Gamma_v}(UT)$ as an approximation, since models of the unit term are also models of this specification.

Example 5.3. [28] Consider the following architectural specification:

arch spec ASP_NAME =
  units
    A : {**sort** *s*};
    F : {**sort** *s*} → {**sort** *s*; **op** *a* : *s*};
    G : {**sort** *s*} → {**sort** *s*; **op** *a* : *s*};
    H : {**sort** *s*; **op** *a* : *s* } → {**sort** *s*; **ops** *a*, *b* : *s* };
    B = F [A];
    C = G [A]
  result H [B] **and** {H [C] **with** $a \mapsto a'$, $b \mapsto b'$}
end

The specification $\mathcal{S}_{\text{ASP\_NAME}}$ of the result unit term of ASP_NAME is {**sort** *s*; **ops** $a, b, a', b' : s$}. However, if *M* is a model of $\mathcal{S}_{\text{ASP\_NAME}}$ such that $M_a = M_{a'}$, the non-generative semantics imposes that applying H twice (with B and C as arguments, respectively) to the same model yields the same result, and therefore M_b and $M_{b'}$ must also be equal. However, $a = a' \implies b = b'$ is not a consequence of $\mathcal{S}_{\text{ASP\_NAME}}$, which is in this case only an over-approximation of the model class of the result unit term of ASP_NAME. $\square$

We are now ready to introduce a new calculus for correctness of architectural specifications, that we refer to as *constructive*, given in Fig. 16. The judgements of the calculus are of the form $\vdash ASP ::_c USP$, where *USP* is now constructed by the rules of the calculus. Therefore, the only verification conditions of this calculus are those introduced in the definition of the specification of a unit expression. It is also no longer necessary to carry dependencies between units in a diagram labeled with specifications. Instead, the calculus builds a verification context for the units declared or defined, and this verification context is then used to construct the specification of the result unit expression of the architectural specification being verified. The rule for unit declarations takes into account the fact that the specification *USP* of a unit can itself be an architectural specification. The result of applying the calculus to *USP* is a unit specification *USP′*, which can be either *USP* if *USP* was already a unit specification (last rule of the calculus) or the specification of the result unit of *USP* if *USP* is an architectural specification. We moreover use the context of outer declarations of definitions when verifying the architectural specification *USP′*. Thus we can de-sugar unit imports into architectural specifications using units declared outside their scope, see Sect. 5.2.

In the sequel we will use the following framework.

$$\frac{\Gamma_v^{\emptyset} \vdash ASP ::_c USP}{\vdash \textbf{arch spec}\, ASP ::_c USP}$$

$$\frac{\begin{array}{c}\Gamma \vdash UDD_1 ::_c \Gamma_1 \\ \vdots \\ \Gamma_{n-1} \vdash UDD_n ::_c \Gamma_n\end{array}}{\Gamma \vdash \textbf{units}\ UDD_1 \ldots UDD_n\ \textbf{result}\ UE ::_c \mathcal{S}_{\Gamma_n}(UE)}$$

$$\frac{\Gamma \vdash UDECL ::_c \Gamma'}{\Gamma \vdash UDECL\ \textbf{qua}\ UDD ::_c \Gamma'} \qquad \frac{\Gamma \vdash UDEFN ::_c \Gamma'}{\Gamma \vdash UDEFN\ \textbf{qua}\ UDD ::_c \Gamma'}$$

$$\frac{\Gamma \vdash USP ::_c USP'}{\Gamma \vdash UN : USP ::_c \Gamma \cup \{UN \mapsto USP'\}} \qquad \frac{}{\Gamma \vdash UN = UE ::_c \Gamma \cup \{UN \mapsto \mathcal{S}_\Gamma(UE)\}}$$

$$\frac{}{\Gamma \vdash SP_1 \times \ldots \times SP_n \to SP ::_c SP_1 \times \ldots \times SP_n \to SP}$$

Figure 16: Proof calculus for CASL architectural specifications.

Framework 5.1. *ASP* is an architectural specification such that $\vdash ASP \rhd \Box$ and no generic unit specification in *ASP* is inconsistent. $\Box$

If an architectural specification *ASP* is of the form **units** $UDD_1 \ldots UDD_n$ **result** *UE*, we write $\vdash UDD^+ :: \Gamma_{gen}, \Gamma$ if $\vdash UDD_i :: \Gamma^i_{gen}, \Gamma^i$ for $i = 1, \ldots, n$, $\Gamma_{gen} = \bigcup_{i=1,\ldots,n} \Gamma^i_{gen}$ and $\Gamma = \bigcup_{i=1,\ldots,n} \Gamma^i$. Similarly, we write $\Gamma_0 \vdash UDD^+ ::_c \Gamma$ if $\Gamma_0 \vdash UDD_i ::_c \Gamma_i$ for $i = 1, \ldots, n$ and $\Gamma = \Gamma_n$.

To be able to compare the two calculi, Thm. 5.1 needs to be generalized to the whole architectural language. For now, unit imports are excluded—these will be covered in Sect. 5.2. This means we have to treat unit definitions and generalise from single parameter to multi-parameter units. This is rather straightforward. The constructive and the deductive versions of the proof calculus are then related by the following result.

Theorem 5.2. Under the requirements of Framework 5.1, $\vdash ASP ::_c USP$ implies $\vdash ASP :: USP$. $\Box$

Together with Thm. 5.1, we get the following immediate result.

Theorem 5.3. [Soundness] Under the requirements of Framework 5.1, if $\vdash ASP ::_c USP$, we have that $\vdash ASP \Rightarrow_g \mathcal{AM}$ and $U \in Unit(USP)$ for all $(U, BM) \in \mathcal{AM}$. $\Box$

For the implication in the other direction, we need to strengthen the framework, because the specification of the unit term merely approximates its model class:

Framework 5.2. *ASP* is an architectural specification such that $\vdash ASP \rhd \Box$, no generic unit specification in *ASP* is inconsistent, no generic unit is applied more than once, and the specification of each non-generic unit which is not used (directly or indirectly) in the result unit expression is consistent. $\Box$

Let us first notice that in some cases, if *ASP* is an architectural specification with result unit expression *UE*, the model class of $\mathcal{S}_{ASP}(UE)$ and the class of models to which *UE* may evaluate coincide. Let therefore *ProjRes* take any model of *ASP* to the interpretation of *UE* in this model.

Theorem 5.4. Under the requirements of Framework 5.2, if $\vdash ASP ::_c USP$ then $ProjRes(\mathbf{Mod}(ASP)) = \mathbf{Mod}(USP)$. $\Box$

Theorem 5.5. Under the requirements of Framework 5.2, if $\vdash ASP :: USP$ for some *USP*, then $\vdash ASP ::_c USP'$ where $USP' = \mathcal{S}_{ASP}$ is the specification of the result unit expression *UE* of *ASP* and moreover $USP \rightsquigarrow USP'$. $\Box$

We can now combine the results that we have obtained so far. Therefore, Thm. 5.1 needs to be generalized to the whole architectural language. For now, unit imports are excluded—these will be covered in Sect. 5.2. This means we have to treat unit definitions and generalise from single parameter to multi-parameter units. This is rather straightforward. We then get the following result.

Theorem 5.6. [Completeness] Under the requirements of Framework 5.2, if $\vdash ASP \Rightarrow_g \mathcal{AM}$ for some $\mathcal{AM}$ such that for all $(U, BM) \in \mathcal{AM}$, $U \in Unit[USP]$ for some USP then $\vdash ASP ::_c USP'$, where $USP' = \mathcal{S}_{ASP}$ is the specification of the result unit expression UE of ASP and moreover $USP \rightsquigarrow USP'$. □

5.2. Unit Imports in Casl

Recall that so far we have restricted the architectural language to a variant without unit imports. In the full Casl architectural language, a unit may have several imports, as in the following example.

Example 5.4. Let us consider the following architectural specification with unit imports:

arch spec ASP_NAME =
  units M_1: SP_1
    ...
    $M_n : SP_n$;
    $UN : SP \to SP'$ **given** $M_1 , \cdots , M_n$
    ...
  result ...

This can be equivalently expressed using an anonymous architectural specification with a single component which is a generic unit that is applied just in the result unit expression:

arch spec ASP_NAME =
  units $M_1 : SP_1$
    ...
    $M_n : SP_n$;
    UN : **arch spec** {
      units $F : SP_1 \times \cdots \times SP_n \times SP \to SP'$
      result lambda $X : SP \bullet F\ [M_1] \cdots\ [M_n]\ [X]$};
    ...
  result ...

□

In general, an imported unit term may be arbitrarily complex and therefore its specification may not be available directly. To cover this general case, we can make use of the constructive nature of our new calculus to obtain the specification of a unit term in an architectural specification, as introduced in Def. 5.3. Note that this would not have been possible in the context of the old architectural calculus for refinements, as the specification of a unit term in an architectural specification was not defined. A declaration $UN : SP$ **given** UT in a verification context Γ_v, resulting from the analysis of previous declarations, can be written as UN : **arch spec** {**units** $F : \mathcal{S}_{\Gamma_v}(UT) \to SP$; **result** $F[UT]$}. As discussed already—see Example 5.3 and the paragraph preceding it—the specification of a unit term provides in general only an over-approximation of the class of models of the unit term.[17] Therefore, the equivalence between the two ways of writing down the specification of UN (with imports and with generic units applied once, respectively) is precise only up to this approximation. With this in mind, we can use the results of [29] that modify the rules of the semantics of Casl architectural specifications in such a way that one is allowed to replace units with imports with an equivalent construction using anonymous architectural specifications, as in the example above: [29] makes the units $\mathrm{M}_1, \ldots \mathrm{M}_n$ visible in the anonymous architectural specification of UN and also corrects the rule for lambda expressions to keep track of the units used in its unit term. This makes the two syntactic constructions semantically

[17]An alternative idea would be to add a specification to the unit term as an annotation.

equivalent. As a result, we can introduce imports in the architectural language without having to add new semantic rules, as they are now just a notational variant for a construction that is already supported in the language, and thus this modification does not imply any new condition on the soundness and completeness results for the architectural proof calculus.

We now come to the task of refining units with imports. Given that imports are now only a convenient way for writing down an anonymous architectural specification, we simply need to refine the implicit generic unit introduced in the equivalent syntactic construction, as we illustrate below.

Example 5.5. The refinement signature of ASP_NAME from Example 5.4 is a branching refinement signature $(U\Sigma, BstC)$, with $BstC(UN)$ being itself a branching static unit context mapping F to its unit signature. Notice that when using the first syntactic construction, the unit name F is not in scope and therefore we cannot refine the component UN of ASP_NAME as:

refinement R = **arch spec** ASP_NAME **then** {*UN* **to** {*F* **to** R'}}

However, the construction of the anonymous architectural specification of UN ensures that it will always have only one unit and therefore we want to allow:

refinement R = **arch spec** ASP_NAME **then** {*UN* **to** R'}

when the signature of UN is a branching static unit context with a single unit name in the domain and the signature of that unit name matches the source signature of R'. This would require that in the composition of refinement signatures, we simply transform the refinement signature $R\Sigma'$ of R' into a component refinement signature mapping the name of the generated generic unit (in our case F) to $R\Sigma'$ before updating the signature of UN in $BstC$, and thus the name F is not made visible to the user. The rule in the model semantics is similar. □

We therefore extend the composition of refinement signatures (Def. 4.3) with a new case:

- $R\Sigma_1 = (U\Sigma, BstC)$ with only one unit name UN in the domain of $BstC$ and the composition $R\Sigma'$ of $BstC(UN)$ with $R\Sigma_2$ is defined. In this case, $R\Sigma_1; R\Sigma_2 = (U\Sigma, BstC[\{UN \mapsto R\Sigma'\}])$.

Similarly, the composition of refinement relations (Def. 4.6) must be extended to a new case as well:

- $R\Sigma_1 = (U\Sigma, BstC')$ and $BstC'$ has only UN in the domain, and
  $R\Sigma' = BstC'(UN); R\Sigma_2$ is defined. In this case, $\mathcal{R}_1 ; \mathcal{R}_2$ is defined as $\mathcal{R}_1 ; \{UN \mapsto \mathcal{R}_2\}$

This does not introduce any ambiguities between units written at different nesting levels in the same architectural specification, as they have different refinement signatures: if we have a unit declaration of the form UN : **arch spec** {**units** $UN : SP; \ldots$}, the signature of the inner declaration of UN is a simple refinement signature, while the signature of the outer declaration of UN is a branching refinement signature.

5.3. Proof Calculus for Refinements

The proof calculus for architectural specifications checks that result units satisfy a unit specification (given or constructed). We now introduce its counterpart for specifications of refinements, tailored according to the three kinds of refinements we consider. This allows us to express the proof calculus rule for compositions of refinements in a more concise manner.

Definition 5.4. *Let $R\Sigma$ be a refinement signature. A* refinement specification *S over $R\Sigma$ is defined as follows:*

S/B *if $R\Sigma = (U\Sigma, B\Sigma)$, then S takes the form (USP, BSP), where $\vdash USP \rhd U\Sigma$ and BSP is a* branching specification, *which is in turn either a unit specification USP' such that $\vdash USP' \rhd U\Sigma'$ when $B\Sigma = U\Sigma'$, or a map $\mathcal{SPM}$ such that $dom(\mathcal{SPM}) = dom(BstC)$ and $\mathcal{SPM}(X)$ is a branching specification over $BstC(X)$, for each $X \in dom(BstC)$, when $B\Sigma = BstC$;*

C *if $R\Sigma = \{UN_i \mapsto R\Sigma_i\}_{i\in\mathcal{J}}$, then S takes the form $\{UN_i \mapsto S_i\}_{i\in\mathcal{J}}$, where S_i is a refinement specification over $R\Sigma_i$.*

□

The intuitive idea is that if a refinement specifies a constructor $\kappa\colon \mathbf{Mod}(\Sigma) \to \mathbf{Mod}(\Sigma')$, a refinement specification (USP', USP) consists of two unit specifications USP and USP' with $\vdash USP \rhd \Sigma$ and $\vdash USP' \rhd \Sigma'$ such that $dom(\kappa) = \mathbf{Mod}(USP)$ and $cod(\kappa) \subseteq \mathbf{Mod}(USP')$. The latter two conditions are captured by Def. 5.6 below. This generalizes to n-ary constructors and to families of constructors in the obvious way. We denote the empty map by $\mathcal{SPM}_\emptyset$.

Again, the proof calculus rules rely on a composition operation for refinement specifications.

Definition 5.5. *Let $R\Sigma_1$ and $R\Sigma_2$ be two refinement signatures such that $R\Sigma_1; R\Sigma_2$ is defined and let S_i be a refinement specification over $R\Sigma_i$ for $i = 1, 2$. The* composition $S_1; S_2$ *is defined inductively as follows:*

S;S/S;B *if $S_1 = (USP_1, USP_2)$, then $S_2 = (USP_3, BSP)$ and $S_1; S_2 = (USP_1, BSP)$, provided that $USP_2 \rightsquigarrow USP_3$.*

B;C *if $S_1 = (USP_1, \mathcal{SPM}_1)$ then S_2 must be of the form $\{UN_i \mapsto S_i\}_{i\in\mathcal{J}}$. We define $S_1; S_2 = (USP_1, \mathcal{SPM}_1[S_2])$, where $\mathcal{SPM}[S]$ modifies $\mathcal{SPM}$ for $A \in dom(S)$ as follows:*

- *if $\mathcal{SPM}(A)$ is a unit specification USP, then $S(A)$ must be of the form (USP', BSP'). Then $\mathcal{SPM}[S](A) = BSP'$, provided that $USP \rightsquigarrow USP'$.*
- *if $\mathcal{SPM}(A) = \mathcal{SPM}'$, then $S(A)$ must be of the form $\{UN'_j \mapsto S'_j\}_{j\in\mathcal{J}'}$. Then $\mathcal{SPM}[S](A) = \mathcal{SPM}'[S(A)]$.*

C;C *if $S_1 = \{UN_i \mapsto S_i\}_{i\in\mathcal{J}}$, then $S_1; S_2$ is defined only if $S_2 = \{UN'_j \mapsto S_j\}_{j\in\mathcal{J}'}$. Then $S_1; S_2$ modifies the ill-defined union of S_1 and S_2 by putting $(S_1; S_2)(A) = S_1(A); S_2(A)$ for $A \in dom(S_1) \cap dom(S_2)$.*

□

The constructive proof calculus for architectural specifications is extended to the level of refinements as in Fig. 17. The judgments of the proof calculus for refinements are of the form $\vdash SPR ::_c S$, where SPR is a refinement and S is a refinement specification and the calculus is defined inductively on the structure of the refinements. A simple refinement USP **refined via** σ **to** SPR is correct if SPR is itself correct and its refinement specification is (USP', BSP) and moreover the crucial verification condition $USP \overset{\sigma}{\rightsquigarrow} USP'$, stated semantically here, holds. Other verification conditions appear as side-conditions of the definition of composition of refinement specification, in the rule for composition of refinements. We also complete the architectural proof calculus, as introduced in Fig. 16, as specifications of units are now simple or branching refinements. The idea is that for each unit declaration $UN_i : SPR_i$ of an architectural specification ASP, we define $\Gamma_v(UN_i) = USP_i$ if $\vdash SPR_i ::_c (USP_i, BSP_i)$, where Γ_v is the verification context of ASP. This would suffice for the verification of ASP with the architectural calculus. Furthermore, we need to set $\mathcal{SPM}(UN_i) = BSP_i$ in the refinement specification $\mathcal{SPM}$ of ASP, to be able to check correctness of further refinements of the units UN_i.

Definition 5.6. *Let $R\Sigma$ be a refinement signature, S be a refinement specification over $R\Sigma$ and $\mathcal{R}$ be a refinement relation over $R\Sigma$. We define the* satisfaction *of a refinement specification by a refinement relation, denoted $\mathcal{R} \models S$, inductively as follows:*

S *if $R\Sigma = (U\Sigma, U\Sigma')$, then $S = (USP, USP')$ and $\mathcal{R} \subseteq Unit(U\Sigma) \times Unit(U\Sigma')$. Then $\mathcal{R} \models S$ iff $\mathcal{R} \subseteq Unit(USP) \times Unit(USP')$ and moreover, for each $U' \in Unit(USP')$ there is a $U \in Unit(USP)$ such that $(U, U') \in \mathcal{R}$;*

B *if $R\Sigma = (U\Sigma, BstC)$, then $S = (USP, \mathcal{SPM})$ and $\mathcal{R} \subseteq \{(U, BM) | U \in Unit(U\Sigma), BM$ is a branching model over $BstC\}$. Then $\mathcal{R} \models S$ iff for all $(U, BM) \in \mathcal{R}$, $U \in Unit(USP)$ and for $A \in dom(\mathcal{SPM})$ we have that $BM(A) \models \mathcal{SPM}(A)$ ($\mathcal{SPM}$ and BM have the same domain), and moreover for each branching model BM of $\mathcal{SPM}$ there is a unit U such that $(U, BM) \in \mathcal{R}$;*

C *if $R\Sigma = \{UN_i \mapsto R\Sigma_i\}_{i\in\mathcal{J}}$, then $S = \{UN_i \to S_i\}_{i\in\mathcal{J}}$ and $\mathcal{R} = \{UN_i \to \mathcal{R}_i\}_{i\in\mathcal{J}}$. Then $\mathcal{R} \models S$ iff $\mathcal{R}_i \models S_i$ for all $i \in \mathcal{J}$.*

□

The following result states that if a statically well-formed refinement SPR can be proven correct w.r.t. a refinement specification S using the proof calculus for refinements, then SPR has a denotation according to the model semantics and moreover the refinement relation thus obtained satisfies S.

$$\frac{}{\vdash USP ::_c (USP, USP)} \qquad \frac{\begin{array}{c}\vdash SPR ::_c (USP', BSP) \\ USP \overset{\sigma}{\rightsquigarrow} USP'\end{array}}{\vdash USP \textbf{ refined via } \sigma \textbf{ to } SPR ::_c (USP, BSP)}$$

$$\frac{\begin{array}{c}\vdash SPR_1 ::_c S_1 \\ \vdash SPR_2 ::_c S_2 \\ S = S_1 ; S_2\end{array}}{\vdash SPR_1 \textbf{ then } SPR_2 ::_c S} \qquad \frac{\text{for each } i \in \mathcal{J} \ \vdash SPR_i ::_c S_i}{\vdash \{UN_i \textbf{ to } SPR_i\}_{i \in \mathcal{J}} ::_c \{UN_i \to S_i\}_{i \in \mathcal{J}}}$$

$$\frac{\Gamma_v^{\emptyset}, \mathcal{SPM}_{\emptyset} \vdash ASP ::_c (USP, \mathcal{SPM})}{\vdash \textbf{arch spec } ASP ::_c (USP, \mathcal{SPM})} \qquad \frac{\begin{array}{c}\Gamma, \mathcal{SPM} \vdash UDD_1 ::_c (\Gamma_1, \mathcal{SPM}_1) \\ \vdots \\ \Gamma_{n-1}, \mathcal{SPM}_{n-1} \vdash UDD_n ::_c (\Gamma_n, \mathcal{SPM})\end{array}}{\Gamma \vdash \textbf{units } UDD_1 \ldots UDD_n \textbf{ result } UE ::_c (\mathcal{S}_{\Gamma_n}(UE), \mathcal{SPM})}$$

$$\frac{\Gamma, \mathcal{SPM} \vdash UDECL ::_c (\Gamma', \mathcal{SPM}')}{\Gamma, \mathcal{SPM} \vdash UDECL \textbf{ qua } UDD ::_c (\Gamma', \mathcal{SPM}')} \qquad \frac{\Gamma \vdash UDEFN ::_c \Gamma'}{\Gamma, \mathcal{SPM} \vdash UDEFN \textbf{ qua } UDD ::_c (\Gamma', \mathcal{SPM})}$$

$$\frac{\Gamma \vdash SPR ::_c (USP, BSP)}{\Gamma, \mathcal{SPM} \vdash UN : SPR ::_c (\Gamma \cup \{UN \mapsto USP\}, \mathcal{SPM} \cup \{UN \mapsto BSP\})} \qquad \frac{}{\Gamma \vdash UN = UE ::_c \Gamma \cup \{UN \mapsto \mathcal{S}_{\Gamma}(UE)\}}$$

Figure 17: Proof calculus for CASL refinements.

Theorem 5.7. [**Soundness**] Under the requirements of Framework 5.1, let SPR be a refinement such that $\vdash SPR \rhd \Box$. If $\vdash SPR ::_c S$, then $\vdash SPR \Rightarrow \mathcal{R}$ for some refinement relation $\mathcal{R}$ and moreover $\mathcal{R} \models S$. □

Example 5.6. We illustrate how the proof calculus for the CASL refinement language applies to the specifications in Examples 2.3–2.5.

For the unit specifications MONOID, NATWITHSUC, NAT and NATBIN, the refinement specification is of the form (*USP*, *USP*), where *USP* is in each case the name of the unit specification.

For R1, we use the proof calculus rule for simple refinements (second rule in Fig. 17), which means that we check that MONOID $\overset{Elem \mapsto Nat}{\rightsquigarrow}$ NAT. Since addition of natural numbers is associative and 0 is the unit for addition, the refinement condition holds and the refinement R1 is correct. The refinement specification of R1 is (MONOID, NAT).

For R2, we use again the proof calculus rule for simple refinements and check that NAT $\overset{Nat \mapsto Bin}{\rightsquigarrow}$ NATBIN. The refinement specification of R2 is (NAT, NATBIN).

For R3, we need to compose the refinement specifications (MONOID, NAT) and (NAT, NATBIN). The verification condition NAT $\rightsquigarrow$ NAT holds trivially and thus the composition of R1 and R2 is correct, with the resulting refinement specification (MONOID, NATBIN).

For ADDITION_FIRST, we first replace the imports with generic units (see Sect. 5.2):

```
arch spec ADDITION_FIRST =
    units N : NAT;
        M :arch spec {
              units F :NAT → NATWITHSUC
              result F[N]}
    result M
```

and the refinement specification of ADDITION_FIRST is (NATWITHSUC, $\{N \mapsto$ NAT, $M \mapsto \{F \mapsto$ NAT $\to$ NATWITHSUC$\}\}$).

Correctness of R4 is immediate because the condition NatWithSuc $\rightsquigarrow$ NatWithSuc holds trivially and R4 has the same refinement specification as Addition_First.

Finally for R5, the refinement specification of RComp (which is the same specification as the one following **then**) is $\{N \mapsto (\text{Nat}, \text{NatBin})\}$ and we need to check that it composes with the refinement specification of R4, which we denote by V_4. Indeed, N is in the domain of the branching specification B on the second component of V_4 and $B(N) = \text{Nat}$; thus, the verification condition Nat $\rightsquigarrow$ Nat again holds immediately. The refinement specification of R5 becomes $(\text{NatWithSuc}, \{N \mapsto \text{NatBin}, M \mapsto \{F \mapsto \text{Nat} \to \text{NatWithSuc}\}\})$. □

Example 5.7. We now present how the proof calculus applies in the case of the refinements in Example 2.6.

For Ref_Sbcs, the second rule in Fig. 17 must be applied (note that σ is the identity). This amounts to checking correctness of Arch_Sbcs and proving that

$$\text{Sbcs\_Open} \rightsquigarrow \mathcal{S}_{\text{Arch\_Sbcs}}(UE)$$

where $UE = \lambda V : \text{Value} \bullet \text{C[A[S[P[V]]]]}$.

Since Arch_Sbcs contains only unit declarations, each unit is assigned its declared specification and $\mathcal{S}_{\text{Arch\_Sbcs}}(UE)$ is obtained as follows:

- UE is a lambda expression, so $\mathcal{S}_{\text{Arch\_Sbcs}}(UE) = \text{Value} \to SP$, where $SP = \mathcal{S}_{\text{Arch\_Sbcs}}(\text{C[A[S[P[V]]]]})$;
- since V : Value and P : Value $\to$ Preliminary, the verification condition for the unit application P[V] is Value $\rightsquigarrow$ Value which holds trivially. The specification of P[V] is Preliminary (because Value is included in Preliminary);
- the other units are similar, and the last specification obtained is Steam_Boiler_Control_System.

We thus obtain $\mathcal{S}_{\text{Arch\_Sbcs}}(UE) = \text{Value} \to \text{Steam\_Boiler\_Control\_System}$, and since this is precisely Sbcs_Open, the refinement Ref_Sbcs is correct. The refinement specification of Ref_Sbcs is (Sbcs_Open, $\mathcal{SPM}$) where $\mathcal{SPM}(\text{P}) = \text{Value} \to \text{Preliminary}$, $\mathcal{SPM}(\text{S}) = \text{Preliminary} \to \text{Sbcs\_State}$, $\mathcal{SPM}(\text{A}) = \text{Sbcs\_State} \to \text{Sbcs\_Analysis}$ and $\mathcal{SPM}(\text{C}) = \text{Sbcs\_Analysis} \to \text{Steam\_Boiler\_Control\_System}$.

For Ref_Sbcs', the third rule in Fig. 17 is applied. We have just checked correctness of Ref_Sbcs and obtained its refinement specification (Sbcs_Open, $\mathcal{SPM}$). Therefore, we only have to check correctness of the component refinement following **then** and that the refinement specification obtained, which will be a map $\mathcal{SPM}'$, can be composed with (Sbcs_Open, $\mathcal{SPM}$).

With the fourth rule in Fig. 17, we must check correctness of the refinement specification of each of the components P, S and A.

For the unit S, we must check correctness of State_Ref, which amounts to proving that models of Sbcs_State_Impl are indeed models of Sbcs_State. The refinement specification obtained is (State_Abstr, Unit_Sbcs_State). Note that the correctness proof is non-trivial in this case and needs to use a proof calculus for structured specifications, see Remark 3.1 at the end of Sect. 3.

For the unit P, we must check that Arch_Preliminary is correct. The verification conditions for the two anonymous architectural specifications obtained for MS and MR (as in Sect. 5.2) hold trivially, and we get $\mathcal{S}_{\text{Arch\_Preliminary}}(\text{MS}) = \text{Messages\_Sent}$ and $\mathcal{S}_{\text{Arch\_Preliminary}}(\text{MR}) = \text{Value} \to \text{Messages\_Received}$. The specification of the result unit expression (call it UE') of Arch_Preliminary is of the form Value $\to SP'$ where SP' is the specification of the unit term (call it UT) of the lambda expression. By definition, because UT is a unit amalgamation, SP' is the union of the specifications of the terms SET [MS **fit** *Elem* $\mapsto$ *S_Message*][V], SET [MR[V] **fit** *Elem* $\mapsto$ *R_Message*][V] and CST [V] with the specification $\mathcal{S}_{amalg}(\text{UT})$.

For the first term, the verification conditions are

$$\text{Messages\_Sent} \models \{\textbf{sort } Elem\} \textbf{ with } Elem \mapsto S\_Message$$

and Value $\rightsquigarrow$ Value. Both hold immediately and the specification of the term is

$$(\text{Set}[\textbf{sort } Elem] \textbf{ with } Elem \mapsto S\_Message) \textbf{ and } \text{Messages\_Sent}$$

For the second term, we similarly obtain the specification

$$(\text{SET}[\textbf{sort } \textit{Elem}]\ \textbf{with } \textit{Elem} \mapsto \textit{R\_Message})\ \textbf{and}\ \text{MESSAGES\_RECEIVED}$$

For the third term, notice that V : VALUE and the condition VALUE $\rightsquigarrow$ VALUE holds immediately. Because the specification SBCS_CONSTANTS extends VALUE, the specification obtained is just SBCS_CONSTANTS.

Finally, for the unit A, we need to check correctness of the architectural specification ARCH_ANALYSIS. This does not bring anything new to the cases discussed before and therefore we omit a detailed presentation.

We must then check that the refinement specifications (SBCS_OPEN, $\mathcal{SPM}$) and $\mathcal{SPM}'$ compose. Notice that the domain of $\mathcal{SPM}'$ is included in the domain of $\mathcal{SPM}$. For the unit S, the verification condition is immediate and the corresponding specification of S is updated to UNIT_SBCS_STATE. For the unit A, the verification condition holds by noticing that PRELIMINARY is equivalent to the specification SP' obtained in the specification of the result unit of ARCH_PRELIMINARY. $\mathcal{SPM}$ is then updated in A to the map taking each of the units of ARCH_PRELIMINARY to its specification. □

5.4. Completeness of the Proof Calculus

Recall that the intuition behind Def. 5.6 is that we check that a refinement specifies a constructor $\kappa : \mathbf{Mod}(\Sigma) \rightarrow \mathbf{Mod}(\Sigma')$ that takes any model U of some Σ-specification USP to a model $\kappa(U)$ of some Σ'-specification USP'. While we ensure that the constructor should be total on $\mathbf{Mod}(USP)$, the definition does not require that the constructor should be surjective on $\mathbf{Mod}(USP')$. In particular, there can be another Σ'-specification USP'' that provides a more precise description of $\kappa(\mathbf{Mod}(USP))$. This is obvious in the case of simple refinements: if the refinement R $= USP$ **refined via** σ **to** USP' is correct, we have that $\vdash$ R $\Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U|_\sigma, U) | U \in \mathit{Unit}(USP')\}$. Then the models produced by the constructor associated with R are not necessarily all USP-models, but generally only (USP' **hide** σ)-models.

An architectural specification defines a construction that appears in a top-down development process, when a given requirement specification is implemented by a number of specifications and these specifications act like an interface between the components of the architectural specification. In particular, this introduces an abstraction barrier between a given requirement specification and its refinement. The implementation of the units of an architectural specification can be changed (as long as they respect their specifications) while keeping the possibility of assembling the units to produce a system satisfying the given requirement specification.

The calculus that we introduced in the previous section respects this abstraction barrier. However, this comes at a certain price: Example 5.8 below shows that in some cases, we can prove a refinement correct only if we exploit the properties induced by a certain implementation we have chosen during development of the system. Using information about the choice of implementation gives a more precise description of the image of a constructor. But this of course breaks the abstraction barrier mentioned above. In the example, we have to make use of a particular choice for a refinement of such a component when proving correctness of the entire development.

Hence, we cannot expect to prove completeness of the calculus introduced above, because it respects the abstraction barrier. In the following we will introduce an enhanced proof calculus that can be proven complete and hence necessarily breaks the abstraction barrier.

Example 5.8. Let us consider the following specifications:

spec SIG = **sort** s
          ops $a, b : s$

spec EQ = SIG **then** { • $a = b$ }

refinement INCL = SIG **refined to** EQ

arch spec ASP_NAME = **units** M : SIG **result** M

refinement ASP_EQ = **arch spec** *ASP_NAME* **then** {M **to** INCL}

refinement REF_EQ = EQ **refined to** ASP_EQ

Here the tree of ASP_NAME (in the sense of Sects. 2.4 and 6.2) "grows" in both directions: first the branch corresponding to the unit M is extended via the refinement ASP_EQ, then the tree of ASP_EQ "grows" towards the root, via the refinement REF_EQ.

We have that $\vdash$ ASP_NAME $::_c$ (SIG, $\{M \mapsto$ SIG$\}$) and $\vdash$ {M *to* INCL} $::_c$ $\{M \mapsto$ (SIG, EQ)$\}$. Thus, $\vdash$ ASP_EQ $::_c$ (SIG, $\{M \mapsto$ EQ$\}$) and the verification condition of REF_EQ is EQ $\rightsquigarrow$ SIG, which obviously does not hold. This is due to the fact that when proving the correctness of the entire development, we do not make use of the fact that M has been further refined: in the specification of ASP_EQ we have only modified the specification of the component M. However, this induces a restriction on the domain of the associated constructor, and the codomain gets restricted as well. This change is not captured in the definition of composition of refinement specifications.

□

The enhanced calculus will enable us to capture such further refinements. We can simplify the task of keeping track of the changes introduced by these further refinements by expressing every refinement as equivalent architectural specification(s), in the sense that their models are in a one-to-one correspondence, as we will see below. We have seen that for any architectural specification *ASP*, $\mathcal{S}_{ASP}$ captures exactly (under some conditions) the models produced by the architectural specification. The idea is that rather than defining $\mathcal{S}_{ASP}$ as a specification, we define it as a *specification expression* involving the specifications of the units involved. The expression is then evaluated when proving a refinement and composition of refinements induces substitution of the specification of the refined unit with the expression associated to the refinement we compose with. This gives a dynamic nature to the verification process: at each moment, the specifications provide only a snapshot of the model classes involved, and they can be further restricted by composition.

Example 5.9. Let us again assume that the refinement R = *USP* **refined via** σ **to** *USP*$'$ is correct, and then we have that $\vdash$ R $\Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U|_\sigma, U) \mid U \in \mathit{Unit}(\mathit{USP}')\}$.

The same constructor as the one specified by R, namely **Mod**(σ): $\mathit{Unit}(\mathit{USP}') \to (\mathit{Unit}(\mathit{USP}')$ **hide** $\sigma)$, can be equivalently specified by the architectural specification

arch spec ASP_NAME =
  units *UN* : *USP*$'$
  result *UN* **hide** σ

Given that $\vdash$ ASP_NAME $\Rightarrow \{(U|_\sigma, \{UN \mapsto U\}) \mid U \in \mathit{Unit}(\mathit{USP}')\}$, we get a one-to-one correspondence between the model class of R on one hand, consisting of all assignments $(U|_\sigma, U)$, with $U \in \mathit{Unit}(\mathit{USP}')$, and the model class of ASP_NAME on the other hand, consisting of all $(U|_\sigma, \{UN \mapsto U\})$, with $U \in \mathit{Unit}(\mathit{USP}')$. In general, whenever such a correspondence can be set up, we will say that the models of R and ASP_NAME *correspond up to unit names*. □

In the case of component refinements, the refinement of each component needs to be replaced by an equivalent architectural specification. We define then *enhanced verification specifications* $\mathcal{VS}$:

$$\mathcal{VS} ::= \mathit{ASP} \mid \{UN_i \mapsto \mathcal{VS}_i\}_{i \in \mathcal{J}}$$

where *ASP* is an architectural specification, and we call enhanced verification specifications of the form $\{UN_i \mapsto \mathcal{VS}_i\}_{i \in \mathcal{J}}$ *enhanced verification maps*. The idea is that if $\mathcal{VS}$ is the enhanced verification specification of a refinement *SPR*, $\mathcal{VS}$ specifies the constructor(s) induced by *SPR*. Moreover, we define a partial operation of *composition* between enhanced verification specifications, denoted $\mathcal{VS}_1;\mathcal{VS}_2$, as follows:

S;S/S;B if $\mathcal{VS}_1 = \mathit{ASP}_1$ such that ASP_1 has no branching[18] and $\mathcal{VS}_2$ is also an architectural specification ASP_2, then $\mathcal{VS}_1;\mathcal{VS}_2$ replaces the specification of the leaf in ASP_1 with ASP_2;

B;C if $\mathcal{VS}_1 = \mathit{ASP}_1$ and $\mathcal{VS}_2$ is an enhanced verification map, then $\mathcal{VS}_1;\mathcal{VS}_2$ is defined iff for each $UN \in \mathit{dom}(\mathcal{VS}_2)$, *UN* is in $\mathit{dom}(\mathcal{VS}_1)$ such that:

[18]This means that the architectural specification has just one unit, and the specification of that unit can be an architectural specification only if it has one component itself, and so on, until a leaf is reached when the specification of the current unit is a unit specification.

$$\frac{\textbf{arch spec } ASP = \textbf{units } UN : USP \textbf{ result } UN}{\vdash USP ::_e ASP}$$

$$\frac{\begin{array}{c}\vdash SPR' ::_e ASP' \\ USP \overset{\sigma}{\rightsquigarrow} \mathcal{S}_{ASP'} \\ \textbf{arch spec } ASP = \textbf{units } UN : \textbf{arch spec } ASP' \textbf{ result } (UN \textbf{ hide } \sigma)\end{array}}{\vdash USP \textbf{ refined via } \sigma \textbf{ to } SPR' ::_e ASP}$$

$$\frac{\begin{array}{c}\vdash SPR_i ::_e ASP_i \\ \textbf{arch spec } ASP' = ASP[\{UN_i/\textbf{arch spec } ASP_i\}_{i=1,\ldots,n}]\end{array}}{\vdash ASP = \textbf{units } \{UN_i : SPR_i\}_{i=1,\ldots,n} \textbf{ result } UE ::_e ASP'}$$

$$\frac{\vdash SPR_i ::_e \mathcal{VS}_i}{\vdash \{UN_i \textbf{ to } SPR_i\}_{i\in\mathcal{J}} ::_e \{UN_i \textbf{ to } \mathcal{VS}_i\}_{i\in\mathcal{J}}} \qquad \frac{\begin{array}{c}\vdash SPR_1 ::_e \mathcal{VS}_1 \\ \vdash SPR_2 ::_e \mathcal{VS}_2 \\ \mathcal{VS} = \mathcal{VS}_1;\mathcal{VS}_2\end{array}}{\vdash SPR_1 \textbf{ then } SPR_2 ::_e \mathcal{VS}}$$

Figure 18: Enhanced calculus for refinements.

- if the specification of UN in ASP_1 is a unit specification USP, then $\mathcal{VS}_2(UN)$ must be an architectural specification ASP_2 and $USP \rightsquigarrow \mathcal{S}_{ASP_2}$. Then we update the specification of UN in ASP_1 to ASP_2;
- if the specification of UN in ASP_1 is an architectural specification ASP', then $\mathcal{VS}_2(UN)$ must be an enhanced verification map $\mathcal{VS}'$ such that $ASP';\mathcal{VS}'$ is defined. Then we update the specification of UN in ASP to $ASP';\mathcal{VS}'$;

C;C if $\mathcal{VS}_1$ is an enhanced verification map, then $\mathcal{VS}_2$ must be an enhanced verification map as well and we define $\mathcal{VS}_1;\mathcal{VS}_2$ by modifying the ill-defined union of $\mathcal{VS}_1$ and $\mathcal{VS}_2$, for each unit name UN in the intersection of the domains of $\mathcal{VS}_1$ and $\mathcal{VS}_2$, by taking $\mathcal{VS}_1;\mathcal{VS}_2(UN) = \mathcal{VS}_1(UN);\mathcal{VS}_2(UN)$.

Models of enhanced verification specifications are obvious generalizations of models of architectural specifications: if an enhanced verification specification $\mathcal{VS}$ is an architectural specification ASP, then the model of $\mathcal{VS}$ is just the architectural model $\mathcal{AM}$ such that $\vdash ASP \Rightarrow \mathcal{AM}$, while if $\mathcal{VS}$ is an enhanced verification map $\{UN_i \mapsto \mathcal{VS}_i\}_{i\in\mathcal{J}}$ and $\mathcal{M}_i$ is a model for $\mathcal{VS}_i$, for each $i \in \mathcal{J}$, then $\{UN_i \mapsto \mathcal{M}_i\}_{i\in\mathcal{J}}$ is a model for $\mathcal{VS}$. By a slight abuse of notation, we denote the models of enhanced verification specifications also by $\mathcal{AM}$, as in the case of architectural specifications, and we write $\vdash \mathcal{VS} \Rightarrow \mathcal{AM}$ to denote that $\mathcal{AM}$ is the model of $\mathcal{VS}$.

An enhanced proof calculus for refinements is presented in Fig. 18, with judgements of the form $\vdash SPR ::_e \mathcal{VS}$, where SPR is a refinement and $\mathcal{VS}$ is an enhanced verification specification. In the rule for architectural specifications, $ASP[\{UN_i/\textbf{arch spec } ASP_i\}_{i=1,\ldots,n}]$ denotes the architectural specification obtained from an architectural specification ASP with unit declarations $UN_i : SPR_i, i = 1, \ldots, n$, by replacing the specification of each declared unit UN_i with an architectural specification ASP_i and keeping all unit definitions as in ASP.

Theorem 5.8. [Soundness of the enhanced calculus] Let SPR be a refinement such that $\vdash SPR \rhd \square$ and the requirements of Framework 5.1 hold for all architectural specifications that appear in SPR. If $\vdash SPR ::_e \mathcal{VS}$ for some enhanced verification specification $\mathcal{VS}$, then $\vdash SPR \Rightarrow \mathcal{R}$ for some refinement relation $\mathcal{R}$, and $\vdash \mathcal{VS} \Rightarrow \mathcal{AM}$ for some $\mathcal{AM}$ such that $\mathcal{R}$ and $\mathcal{AM}$ correspond up to unit names. $\square$

Theorem 5.9. [Completeness of the enhanced calculus] Let SPR be a refinement such that $\vdash SPR \rhd \square$ and the assumptions of Framework 5.2 hold for all architectural specifications appearing in SPR. If $\vdash SPR \Rightarrow \mathcal{R}$ then $\vdash SPR ::_e \mathcal{VS}$ for some enhanced verification specification $\mathcal{VS}$ and $\vdash \mathcal{VS} \Rightarrow \mathcal{AM}$ for some $\mathcal{AM}$ such that $\mathcal{R}$ and $\mathcal{AM}$ correspond up to unit names. $\square$

$$\frac{\vdash cons(USP)}{\vdash cons(USP\ qua\ \mathtt{SPR})} \qquad \frac{\vdash cons(SPR)}{\vdash cons(USP\ \textbf{refined via}\ \sigma\ \textbf{to}\ SPR)}$$

$$\frac{\vdash cons(SPR) \text{ for each } UN : SPR \text{ in } ASP}{\vdash cons(ASP)} \qquad \frac{\vdash cons(SPR_i), i \in \mathcal{J}}{\vdash cons(\{UN_i\ \textbf{to}\ SPR_i\}_{i \in \mathcal{J}})}$$

$$\frac{\begin{array}{c} SPR_1 \text{ contains branchings} \\ \vdash cons(SPR_1) \\ \vdash cons(SPR_2) \end{array}}{\vdash cons(SPR_1\ \textbf{then}\ SPR_2)} \qquad \frac{\begin{array}{c} SPR_1 \text{ does not contain branchings} \\ \vdash cons(SPR_2) \end{array}}{\vdash cons(SPR_1\ \textbf{then}\ SPR_2)}$$

Figure 19: Consistency calculus for refinements.

5.5. Checking Consistency of Refinements

Consistency of a refinement is an important property, because an inconsistent refinement is useless from a software development perspective: it cannot be refined to a program. Hence, detection of consistency should happen at an early stage of the development, before more refinement steps are added. We introduce a calculus for checking whether a refinement *SPR* is consistent, that is, if $\vdash SPR \Rightarrow \mathcal{R}$, then $\mathcal{R}$ is non-empty. This notion is also useful beyond software development: in [30], we have successfully applied this calculus to verify the consistency of the upper ontology Dolce. Dolce is too large for contemporary model finders. Instead of hand-crafting a large and specific model, we have shown the consistency of Dolce using a refinement to an architectural specification, and showing its consistency. This has the advantage of giving a modular model for Dolce that can be changed at various local places (i.e. leaves of the refinement tree) without affecting the possibility to assemble (via the semantics of architectural specifications) a global model of Dolce.

The proof calculus for refinements relies on an obvious observation made already in [27] that constructors preserve consistency. Intuitively, a refinement is consistent if its target is consistent, and an architectural specification is consistent if all its unit specifications are consistent. This makes it clear that our calculus (for checking consistency of the leaves of the refinement tree) must eventually be based on a calculus for the consistency of unit specifications. We assume this to be given, with judgements of the form $\vdash cons(USP)$ capturing consistency of a unit specification *USP*. Checking consistency of non-generic unit specifications amounts to checking consistency of structured specifications; a calculus for this has been introduced in [31] (this is in turn based on an institution-specific calculus for consistency of basic specifications). Checking consistency of generic unit specification amounts to checking conservativity of extensions of structured specifications; for the case of first-order logic and Casl basic specifications, a sound but necessarily incomplete calculus has been developed in [32].

The consistency calculus for refinements is given by the rules in Fig. 19. For checking consistency of compositions SPR_1 **then** SPR_2, if SPR_1 contains a branching, it does not suffice to check consistency of SPR_2 (which must be a component refinement), because some component of SPR_1 outside the domain of SPR_2 might be inconsistent. On the other hand, if SPR_1 does not contain branchings and the composition $SPR_1; SPR_2$ is checked to be correct by the proof calculus for refinements, the verification condition ensures that the units produced with the second refinement are in the domain of the first refinement, and thus the first refinement cannot be inconsistent unless the second one is.

Theorem 5.10. [Soundness] If the requirements of Framework 5.1 hold, and if the calculi for checking consistency of structured specifications and conservativity of extensions are sound, then the calculus for checking consistency of refinements is sound as well. That is, if $\vdash SPR ::_c \Box$ and $\vdash cons(SPR)$, then the denotation of *SPR* is a non-empty refinement relation. □

Completeness holds again only if the specification of each unit term does not approximate, but exactly captures, the model class of the unit term.

Theorem 5.11. [Completeness] If the requirements of Framework 5.2 hold, and if the calculi for checking consistency of structured specifications and conservativity of extensions are complete, then the calculus for checking consistency of refinements is complete as well. That is, if $\vdash SPR ::_c \square$ and *SPR* is non-empty, then $\vdash cons(SPR)$. □

6. Tool Support

6.1. The Heterogeneous Tool Set

The Heterogeneous Tool Set (Hets) [12, 11] is an open source software tool providing a general framework for formal methods integration and proof management. Hets is currently available for Linux and Mac OS X from the Hets home page `http://hets.eu`.

Hets currently supports about 25 institutions, with varying degrees of support. Each institution can be used for the basic layer of specifications (see Sect. 2.1). On top of that, Hets supports libraries of structured and architectural specifications and specification refinements. Intuitively, one can think of Hets as a motherboard where different expansion cards can be plugged in, the expansion cards here being individual logics with their analysis and proof tools. The Hets motherboard already has a number of expansion cards plugged in (e.g., first-order logic with theorem provers like SPASS [33], Vampire [34], higher-order logic with theorem provers Leo-II [35] and Isabelle [36], OWL 2 with provers Fact and Pellet, and more, as well as model finders). Hence, a variety of tools is available, without the need to hard-wire each tool to the logic at hand.

Hets naturally extends Casl structured specifications to heterogeneous specifications: heterogeneity is achieved by parameterizing Hets with a graph of logics and their translations, where logics are formalized as institutions and translations between them as institution comorphisms [37]. The graph is flattened using the so-called Grothendieck construction [38, 39] that again gives rise to an institution. This institution can be plugged into the layer of basic specifications, which means that the theory developed in this paper can also be applied to heterogeneous specifications, with support by Hets. The heterogenous language thus obtained is called HetCASL, which has been further extended to the Distributed Ontology, Model and Specification Language (DOL) [16], a standard of the OMG.[19] DOL can express relations among theories[20] such as logical consequence, conservative extension, views, and refinement[21]. DOL is also capable of expressing such relations between theories written in different logics, as well as translations of theories along logic translations. A DOL file usually imports files written in specific logics. Therefore, besides DOL and HetCASL, Hets supports a number of other input languages directly, such as TPTP, Common Logic, OWL 2 and (part of) UML.

From an implementation perspective, Hets consists of logic-specific tools for the parsing and static analysis of basic logical theories written in the different logics involved, as well as a logic-independent parsing and static analysis tool for structured theories and theory relations. The latter of course needs to call the logic-specific tools whenever a basic logical theory is encountered. Hets uses the formalism of heterogeneous development graphs [25] for structured theorem proving and proof management. A development graph consists of a set of nodes, corresponding to parts of structured specifications, and a set of arrows, called definition links, indicating the dependency between the specification fragments involved. In addition to definition links, theorem links serve to postulate relations between theories. Note that due to the possibility of sharing, the graph structure provides better scalability to industrial-sized developments than the traditional proof calculi for refinement among term-based structured specifications [3, 23, 24].

Based on this support for the layer of structured specifications, we have extended Hets to fully support the architectural and refinement language developed in this paper: refinements can be parsed and statically analysed, and the calculus for correctness of refinements introduced in Sect. 5.3 has been implemented, using development graphs to represent proof obligations for individual refinement steps.

[19]See `http://www.omg.org/spec/DOL/` and `http://www.dol-omg.org`.

[20]Actually, DOL widens the perspective from specifications to models (in the sense of model-driven engineering, which is different from our use of "model" in the sense of model theory in this paper) and ontologies, all of which can thought of as logical theories for our purposes. Hence, we will now use the term "theory" to express this more general perspective.

[21]Currently, architectural specifications and branching refinements are not part of DOL, because these were still under development when DOL was standardized. Still, Hets supports them when analyzing DOL documents.

Example 6.1. A refinement between two unstructured specifications *USP* and *USP*′, written **refinement** R = *USP* **refined via** σ **to** *USP*′, is represented as a development graph with two nodes, one for *USP* and one for *USP*′, and a theorem link from the node of *USP* to the node of *USP*′ labeled with σ. The proof obligation introduced by the theorem link can be discharged using the development graph proof calculus [40], which transforms theorem links into local proof goals at the level of nodes, to be discharged using logic-specific provers.

If *USP* and *USP*′ are *structured*, the development graph will be more complex, as the structure of the two specifications will be represented. For example, assuming that both *USP* and *USP*′ extend an unstructured specification USP_0, the development graph will consist of three nodes, one for each specification involved, and three links, two definition links from USP_0 to *USP* and *USP*′, respectively, and one theorem link from *USP* to *USP*′. □

Hets' implementation of the consistency calculus relies heavily on *refinement trees*, an auxiliary concept for refinements that we introduce below.

6.2. Refinement Trees

The concept of *refinement tree* has already been introduced informally in Sect. 2.4. We now give a formal definition. Refinement trees provide a way of visualising the structure of the development, allowing navigation along the tree of the development and offering information about the individual nodes. Moreover, in Hets, they behave as access points to the logical properties of architectural specifications and refinements. Refinement trees complement development graphs[22], which represent the import structure of the specifications involved and can be used for discharging proof obligations, including those introduced by simple refinement steps. The structure of the development, as captured by refinement trees, may be orthogonal to the specification structure (see [3]).

While intuitively clear, refinement trees have a slightly involved formalization. This is because they are built in a stepwise manner and must be *combined* in the way prescribed by the refinements: composition of refinements gives rise to composition of refinement trees, and we need a mechanism to keep track of the branches and nodes to retrieve the appropriate connection points between trees. Moreover, in the case of branching refinement we obtain a tree with branching and in the case of component refinement we obtain a (sub)tree for each component, to be later connected with the tree of the component that is further refined.

Example 6.2. Figure 20 presents the refinement tree of the specifications in Example 2.6. Single arrows denote components, while double arrows denote refinements. Ref_Sbcs' refines the specifications of the components P, S and A. The subtree corresponding to S displayed in the figure is the result of composition of refinement subtrees (a notion that we introduce below), as will be explained in Example 6.5. In the case of Ref_Sbcs", we first build the trees of the architectural specifications Arch_Failure_Detection and Arch_Prediction. These trees must then be connected with the refinement tree of Ref_SBCS at the nodes corresponding to the units FD and PR, and therefore we must be able to identify these nodes. □

Remark 6.1. Note that there is an arrow from the nodes of MS and MR to a node without a label. Recall from Example 2.6 that these units import B. As we explained in Sect. 5.2, this gives rise to anonymous architectural specifications, each containing just one generic unit with a generated name. The refinement tree of this architectural specification has just one branch from the root to a node corresponding to this generic unit, and since we do not want to introduce the generated name into the scope, we do not add it as a label in the refinement tree. The rules introduced in Sect. 5.2 ensure that this node can be further refined. □

The example shows that refinement trees should consist of a collection of trees (such that we can represent component refinements), and that trees can grow not only at the leaves, but also at the root, with old roots becoming subtrees. This leads to the following definition.

Definition 6.1. *A* refinement tree $\mathcal{RT}$ *is*

- *a tree with*
  - *nodes labelled with names and unit specifications and*

[22]We keep the original terminology, but a more appropriate name for development graphs in this context would be specification graphs.

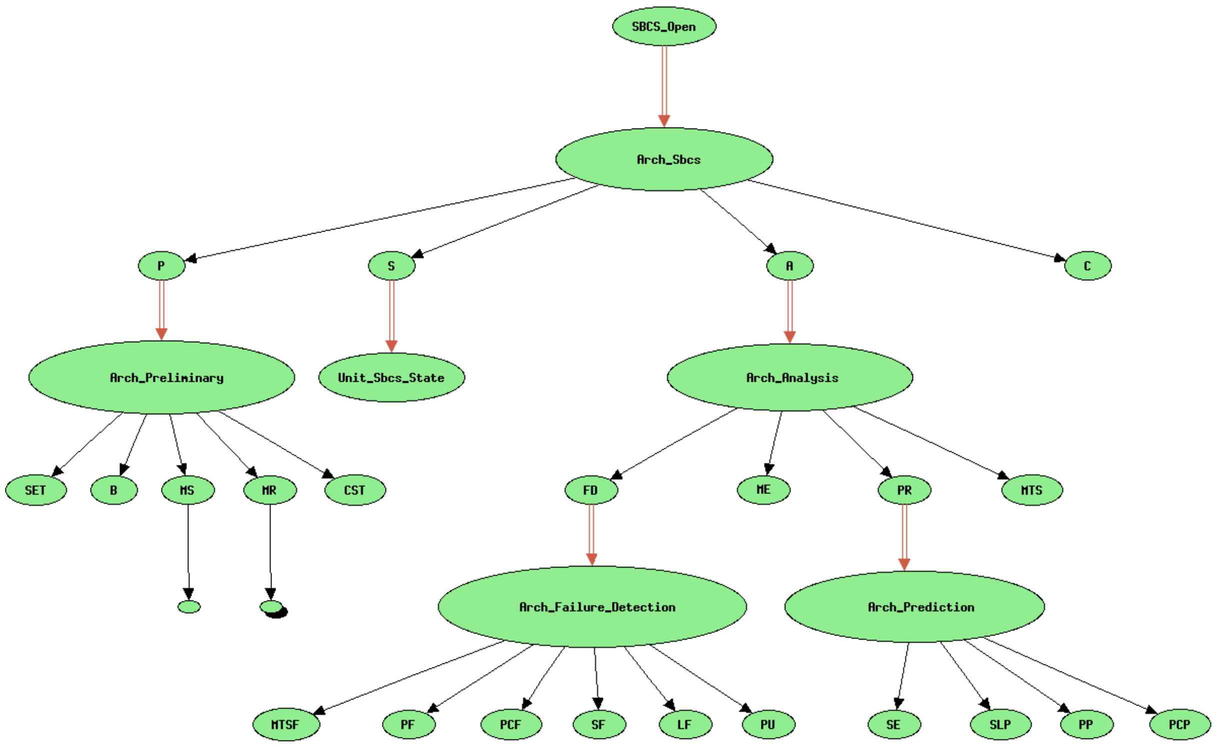


Figure 20: The refinement tree of the steam boiler control system.

– *directed edges between nodes n_1, n_2 of the tree, that can be either*

1. refinement links $n_1 \Rightarrow n_2$ *(to denote refinement steps) or*
2. component links $n_1 \rightarrow n_2$ *(to denote architectural decomposition).*

or

- *a forest of named trees as above.*

□

Example 6.3. In both situations described in Example 6.1, the refinement tree of R has two nodes, one for *USP* and one for *USP′*, and a refinement link between them. □

We need to define an auxiliary structure to keep track of the roots, leaves and nodes of the branching decompositions; this will make it possible to compose refinement trees. We stress that this information is only needed for book-keeping during the construction of refinement trees; it can be dispensed with when looking at a completed refinement tree. Let us define *refinement tree pointers* in a refinement tree $\mathcal{RT}$ as either

S *simple refinement pointers* of the form (n_1, n_2) where n_1 and n_2 are nodes in $\mathcal{RT}$, with the intuition that the first node is the root and the second node is the leaf of a path through the tree,[23]

B *branching refinement pointers* of the form (n, f), where n is a node and f is a map assigning refinement tree pointers to unit names, or

[23]Notice that the two nodes can coincide.

C *component refinement pointers* which are maps assigning refinement tree pointers to unit names.

Example 6.4. We will use the names of the specifications and of the units of architectural specifications in their refinement tree pointers. With this convention in mind, we present some refinement tree pointers for the specifications in Examples 2.3–2.5:

- The pointer of the specification Monoid is (Monoid, Monoid): the refinement tree consists of just one node, which is at the same time the root and the leaf.
- The pointer of R1 is (Monoid, Nat): the refinement tree consists of just one edge, with Monoid at the root and Nat at the leaf. Similarly the pointer of R2 is (Nat, NatBin).
- The pointer of Addition_First is (Addition_First, $\{F \mapsto (F,F), N \mapsto (N,N)\}$): the root of the tree corresponds to the result unit of Addition_First, and we must keep track of the subtrees of the two components. In the case of both F and N, the subtrees consist just of one node.
- The pointer of RComp is $\{N \mapsto$ (Nat, NatBin)$\}$: we assign to N the pointer of R2.

□

We introduce a series of notations and operations on refinement trees. We denote the empty tree by $\mathcal{RT}_\emptyset$. If $\mathcal{RT}$ is a refinement tree, $\mathcal{RT}[USP]$ is obtained by adding to it a new isolated node n labelled with USP. For refinement trees $\mathcal{RT}_1, \ldots, \mathcal{RT}_k$, $\mathcal{RT}[n_1 \to \mathcal{RT}_1, \ldots, \mathcal{RT}_k]$ denotes the tree obtained by inserting component links from the node n_1 of $\mathcal{RT}$ to the roots of each of the argument trees. Moreover, refinement trees can be composed as defined below.

Definition 6.2. *Given two refinement trees $\mathcal{RT}_1$ with pointer p_1 and $\mathcal{RT}_2$ with pointer p_2 we denote by $(\mathcal{RT}, p)$ the* composition $\mathcal{RT}_1;_{p_1,p_2}\mathcal{RT}_2$*, which extends the union of $\mathcal{RT}_1$ and $\mathcal{RT}_2$ as follows:*

S;S *if p_1 is a simple refinement pointer (n_1, n_2) and p_2 is a simple refinement pointer (m_1, m_2), then $\mathcal{RT}$ is obtained by adding a refinement link from n_2 to the successor of the node m_1 along the path (m_1, m_2) in $\mathcal{RT}_2$. The pointer p is then (n_1, m_2).*

S;B *if p_1 is a simple refinement pointer (n_1, n_2) and p_2 is a branching refinement pointer (m_1, f), then $\mathcal{RT}$ is obtained by adding a refinement link from n_2 to m_1. The pointer p is (n_1, f).*

B;C *if p_1 is a branching refinement pointer (n_1, f_1) and p_2 is a component refinement pointer f_2, then $\mathcal{RT}$ is obtained by forming for each X in dom(f_2) the composition of the subtree designated by $f_1(X)$ with the tree designated by $f_2(X)$, yielding a pointer p_X for each X in dom(f_2). The pointer p is then $(n_1, f_1[f_2])$, where $f_1[f_2]$ updates the value of X in f_1 with the pointer p_X.*

C;C *if p_1 is a component refinement pointer f_1 and p_2 is a component refinement pointer p_2, then $\mathcal{RT}$ is obtained by forming for each X in dom(f_2) the composition of the subtree designated by $f_1(X)$ with the tree designated by $f_2(X)$, yielding a pointer p_X for each X in dom(f_2). The pointer p is then $f_1[f_2]$.*

The composition is undefined otherwise. □

Refinement trees will be constructed for correct refinements, in parallel with the verification process. To ease understanding, we have separated the parts that build the refinement trees, as presented in Fig. 21. In particular, the verification calculus will require that corresponding nodes match when making the composition, and thus such requirements can be omitted when forming the composition of refinement trees, as we will only compose refinement trees that match.

Example 6.5. We illustrate the definition of composition of refinement trees with the help of Examples 2.3–2.5. We use the names of the specifications and of the units of architectural specifications to identify nodes in the refinement trees and therefore the names also appear in pointers. The refinement trees and the results of their compositions are presented in Fig. 22.

$$\frac{(n,\mathcal{RT}) = \mathcal{RT}_\emptyset[USP]}{\vdash USP ::_c \mathcal{RT}, (n,n)} \qquad \frac{\begin{array}{c}\vdash USP ::_c \mathcal{RT}_1, p_1 \\ \vdash SPR ::_c \mathcal{RT}_2, p_2 \\ (\mathcal{RT}, p) = \mathcal{RT}_1 ;_{p_1,p_2} \mathcal{RT}_2\end{array}}{\vdash USP \textbf{ refined via } \sigma \textbf{ to } SPR ::_c \mathcal{RT}, p}$$

$$\frac{\begin{array}{c}\vdash SPR_i ::_c \mathcal{RT}_i, p_i \\ UE \text{ is the result unit of } ASP \\ (n,\mathcal{RT}') = \mathcal{RT}_\emptyset[\mathcal{S}_\Gamma(UE)] \\ \mathcal{RT} = \mathcal{RT}'[n \rightarrow \mathcal{RT}_1, \ldots, \mathcal{RT}_k] \\ p = (n, \{UN_i \rightarrow p_i\}_{i=1,\ldots,k})\end{array}}{\vdash ASP ::_c \mathcal{RT}, p} \qquad \frac{\begin{array}{c}\text{for each } i \in \mathcal{J} \\ \vdash SPR_i ::_c \mathcal{RT}_i, p_i \\ \mathcal{RT} = \cup_{i\in\mathcal{J}} \mathcal{RT}_i \\ p = \{UN_i \mapsto p_i\}_{i\in\mathcal{J}}\end{array}}{\vdash \{UN_i \textbf{ to } SPR_i\}_{i\in\mathcal{J}} ::_c \mathcal{RT}, p} \qquad \frac{\begin{array}{c}\vdash SPR_1 ::_c \mathcal{RT}_1, p_1 \\ \vdash SPR_2 ::_c \mathcal{RT}_2, p_2 \\ (\mathcal{RT}, p) = \mathcal{RT}_1 ;_{p_1,p_2} \mathcal{RT}_2\end{array}}{\vdash SPR_1 \textbf{ then } SPR_2 ::_c \mathcal{RT}, p}$$

Figure 21: Construction of refinement trees.

First, refinements R1 and R2 are composed in R3 to form a chain; this is case **S;S** of the definition of composition of refinement trees (Def. 6.2). The pointer p_1 of R1 is $(Monoid, Nat)$ and the pointer p_2 of R2 is $(Nat, NatBin)$. The pointer of the result of their composition is $(Monoid, NatBin)$.

The refinement tree of R4 is obtained by composing the tree of the unit specification NatWithSuc with a single node and with pointer $q_1 = (NatWithSuc, NatWithSuc)$ with the tree of the architectural specification Addition_First (case **S;B**). The pointer of the latter tree is $q_2 = (Addition\_First, \{F \mapsto (F,F), N \mapsto (N,N)\})$. After composition, the pointer q_3 is $(NatWithSuc, \{F \mapsto (F,F), N \mapsto (N,N)\})$.

Finally, the tree of R5 is obtained by composing the tree obtained at the previous step, with pointer q_3, with the set containing the tree of $R2$ with pointer $q_4 = \{N \mapsto (Nat, NatBin)\}$. The pointer of the result is $(NatWithSuc, \{F \mapsto (F,F), N \mapsto (N, NatBin)\})$ (case **B;C**).

Another example for case **B;C** is provided by the component S of the steam boiler control system (Example 2.6). The refinement subtree of S in the tree of Arch_Sbcs consists of one node labeled S, and thus its pointer is (S,S). This tree is composed with the refinement tree of StateRef, which is (State_Abstr,Unit_Sbcs_State), and the result is (S, Unit_Sbcs_State).

We can obtain an example for case **C;C** by writing the refinement of the components in Example 2.6 equivalently, but in a different way:

refinement R = {
    P **to arch spec** Arch_Preliminary, S **to** StateRef,
    A **to arch spec** Arch_Analysis }
    then {
    A **to** {FD **to arch spec** Arch_Failure_Detection,
            PR **to arch spec** Arch_Prediction }}

and the resulting refinement tree is a forest with three trees: one for P, consisting of the subtree with the root in Arch_Preliminary in Fig. 20, one for A, consisting of the subtree with the root in Arch_Analysis in Fig. 20, and one for S, consisting of the tree of StateRef, explained in the last example of the previous case. □

Refinement trees prove useful for making consistency checks with Hets. Checking consistency has been added as a context menu option for nodes in refinement trees: in the case of architectural specifications, the branching points in refinement trees provide the appropriate representation. Selecting "Check consistency" leads to introducing consistency obligations in the development graph of the specification: nodes corresponding to non-generic units carry consistency proof obligations, while morphisms corresponding to theory extensions of generic units carry conservativity obligations. If the node is a branching point, consistency is checked recursively for all components. If an edge in the refinement tree is a refinement link, it suffices to check consistency of the target of the edge, assuming that the refinements are already verified. Hets can be further employed for discarding these obligations, by making use

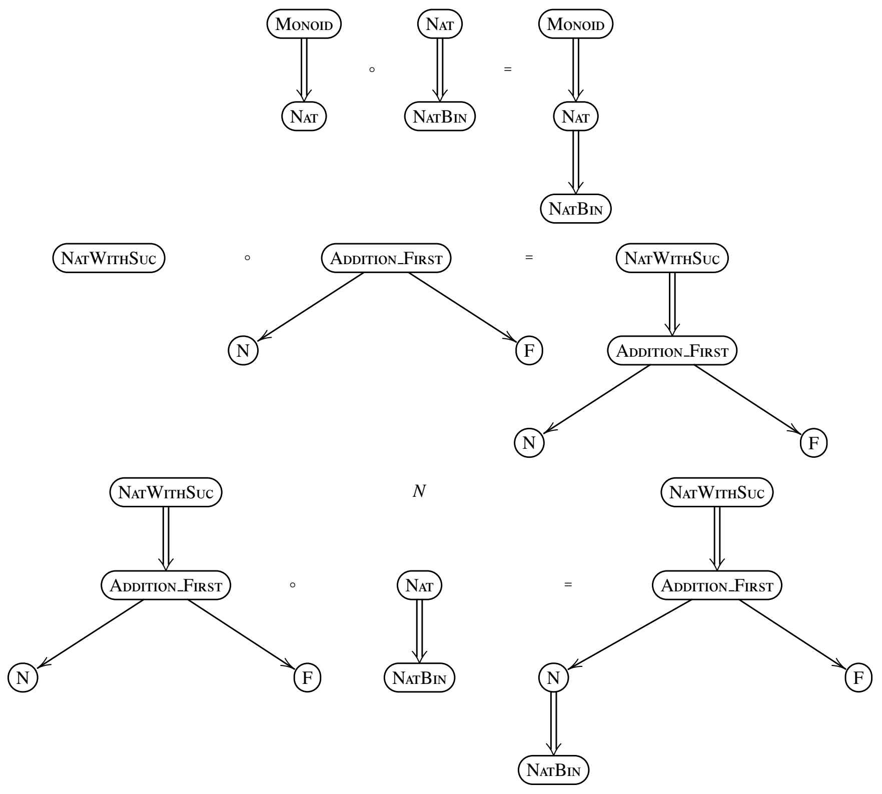


Figure 22: Composition of refinement trees.

of the model finders available in the tool, e.g. `Isabelle-refute` [41], Darwin [42], or SPASS [33], and also the conservativity checker of [32].

7. Remarks on 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 operations on units that can be used for unit terms or unit expressions in architectural specifications. Concerning construction of datatypes, one could add a simple version of freely generated datatypes (modeled by the institution independent notion of so-called free extension [43, 3]), 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 [3, 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 applies to the definition of derived signature morphisms.

An alternative is to approximate the institution-independent essence of programs by considering *monomorphic specifications*. A generic unit specification is monomorphic if the result specification is a monomorphic extension of the argument specifications. This means that it specifies a construction uniquely up to isomorphism. For non-generic

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

Figure 23: Unit term constructs in ML and Haskell

units, this means a unit that is specified uniquely up to isomorphism. Ultimately, monomorphic unit specifications need to be translated to (parameterized) 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.[24] The translation of a parameterized program then provides a construction that is unique, not only unique up to isomorphism. See [44] for details, and [45] for a translation of a subset of CASL to OCaml. Using free extensions, it is also possible to capture partial recursive functions, see [45, 44]. 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. 23.

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 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 parameterized program κ in a constructor implementation $SP \overset{\kappa}{\rightsquigarrow} SP'$ is expressed as

arch spec
 unit κ : { SP' → SP } **refined to** USP
 result λ X : SP' • κ[X]

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

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

[24]In case of non-termination, the (partial) function is not specified uniquely. Consider e.g. the specification $\forall x.f(x) = f(x)$.

is expressed as

```
refinement R5 =
    SP refined to
    arch spec
        units κ : { SP' → SP } refined to USP
              A' : SP' refined to
                      arch spec
                          units κ' : { SP" → SP' } refined to USP'
                                A" : SP"
                      result κ'[A"]
    result κ[A']
```

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

8. Conclusions, Related and Future Work

The issue of refinement has been discussed in many specification frameworks, starting with [46] and [47], 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 refinement relationships between specifications that are analogous to those presented here for Casl. A notable exception is Specware [48], 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 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. This is quite a severe restriction (see [8, 3]), which is overcome by our framework.

We have introduced a language for refinements, together with its semantics, a (sound and complete) proof calculus for proving refinements, another one for checking their consistency, and a notion of refinement tree. We stress that all this is given here in an *institution-independent* way; that is, it applies to any logic that satisfies very mild conditions.

Moreover, we also support refinement trees in practice: we have implemented them in the Heterogeneous Tool Set Hets, so that browsing through and inspection of complex formal developments becomes possible. Note that the proof calculus for architectural specifications of [6] was given for a restricted version of the language; we here extended it to the whole language in a way that is substantially simplified by the transformation of units with imports into generic units. We have also introduced and implemented a sound and complete calculus for consistency of refinements and architectural specifications, which has already been applied for proving the consistency of the upper ontology Dolce in a modular way.

Future work includes extending the language to support *behavioural refinement*, corresponding to *abstractor implementations* in [27]. 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 [49, 50]; then specification refinements as considered here become behavioural [3, 24].

Another useful addition would be amalgamability checks for logics other than Casl in the logic graph of Hets, thus enabling a complete treatment of architectural specifications in those logics.

Acknowledgements

We are grateful to the anonymous reviewers for their careful reading of the paper and suggesting a number of improvements. This work has been partially supported by the German Research Council (DFG) under grant Mo-971/2 "Logic Atlas and Integration (LATIN)" (MC, TM) and by the (Polish) National Science Centre, grant 2013/11/B/ST6/01381 (AT).

References

[1] CoFI, The Common Framework Initiative for algebraic specification and development, electronic archives, Notes and Documents accessible from `http://www.cofi.info/` (2009).

[2] E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, Algebraic Foundations of Systems Specification, Springer, 1999.

[3] D. Sannella, A. Tarlecki, Foundations of algebraic specification and formal software development, EATCS Monographs in Theoretical Computer Science, Springer, 2012.

[4] N. Wirth, Program development by stepwise refinement, Commun. ACM 14 (4) (1971) 221–227.

[5] M. Bidoit, D. Sannella, A. Tarlecki, Architectural specifications in CASL, Formal Aspects of Computing 13 (2002) 252–273.

[6] P. D. Mosses (Ed.), CASL Reference Manual, LNCS 2960 (IFIP Series), Springer, 2004.

[7] J. Fitzgerald, C. Jones, Modularizing the formal description of a database system, in: Proc. VDM'90 Conference, Vol. 428 of Lecture Notes in Computer Science, Springer, 1990, pp. 189–210.

[8] D. Sannella, S. Sokołowski, A. Tarlecki, Toward formal development of programs from algebraic specifications: Parameterisation revisited, Acta Informatica 29 (8) (1992) 689–736.

[9] M. Bidoit, P. D. Mosses, CASL User Manual, LNCS 2900 (IFIP Series), Springer, 2004.

[10] J.-R. Abrial, E. Börger, H. Langmaack (Eds.), Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control, Vol. 1165 of Lecture Notes in Computer Science, Springer, 1996.

[11] T. Mossakowski, Heterogeneous specification and the Heterogeneous Tool Set, Habilitation thesis, University of Bremen (2005).

[12] T. Mossakowski, C. Maeder, K. Lüttich, The Heterogeneous Tool Set, in: TACAS 2007, Vol. 4424 of LNCS, Springer, 2007, pp. 519–522.

[13] T. Mossakowski, D. Sannella, A. Tarlecki, A simple refinement language for CASL, in: J. L. Fiadeiro (Ed.), WADT 2004, Vol. 3423 of Lecture Notes in Computer Science, Springer, 2005, pp. 162–185.

[14] M. Codescu, T. Mossakowski, Refinement trees: calculi, tools and applications, in: A. Corradini, B. Klin (Eds.), Algebra and Coalgebra in Computer Science, CALCO'11, Vol. 6859 of Lecture Notes in Computer Science, Springer, 2011, pp. 145–160.

[15] J. A. Goguen, R. M. Burstall, Institutions: abstract model theory for specification and programming, Journal of the ACM 39 (1992) 95–146.

[16] T. Mossakowski, M. Codescu, F. Neuhaus, O. Kutz, The Distributed Ontology, modeling and specification Language - DOL, in: A. Koslow, A. Buchsbaum (Eds.), The Road to Universal Logic, Vol. 2, Birkhäuser, 2015, pp. 489–520.

[17] J. Adámek, H. Herrlich, G. Strecker, Abstract and Concrete Categories, Wiley, New York, 1990.

[18] R. Diaconescu, Institution-independent Model Theory, Birkhäuser Basel, 2008.

[19] R. Diaconescu, J. Goguen, P. Stefaneas, Logical support for modularisation, in: Proc. 2nd Workshop on Logical Environments, CUP, New York, 1993, pp. 83–130.

[20] J. A. Goguen, G. Roşu, Composing hidden information modules over inclusive institutions, in: O. Owe, S. Krogdahl, T. Lyche (Eds.), From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl, Vol. 2635 of Lecture Notes in Computer Science, Springer, 2004, pp. 96–123.

[21] F. Rabe, How to identify, translate and combine logics?, Journal of Logic and Computation.

[22] T. Mossakowski, Specification in an arbitrary institution with symbols, in: C. Choppy, D. Bert, P. Mosses (Eds.), Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France, Vol. 1827 of Lecture Notes in Computer Science, Springer, 2000, pp. 252–270.

[23] T. Borzyszkowski, Logical systems for structured specifications, Theoretical Computer Science 286 (2002) 197–245.

[24] M. Bidoit, M. Cengarle, R. Hennicker, Proof systems for structured specifications and their refinements, in: E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner (Eds.), Algebraic Foundations of Systems Specification, Springer, 1999, pp. 385 – 433.

[25] T. Mossakowski, S. Autexier, D. Hutter, Development graphs – proof management for structured specifications, Journal of Logic and Algebraic Programming 67 (1-2) (2006) 114–145.

[26] L. Schröder, T. Mossakowski, P. Hoffman, B. Klin, A. Tarlecki, Semantics of architectural specifications in CASL, in: Fundamental Approaches to Software Engineering, Vol. 2029 of Lecture Notes in Computer Science, Springer, 2001, pp. 253–268.

[27] D. Sannella, A. Tarlecki, Toward formal development of programs from algebraic specifications: Implementations revisited, Acta Inf. 25 (1988) 233–281.

[28] P. Hoffman, Architectural specifications and their verification, Ph.D. thesis, Warsaw University (2005).

[29] M. Codescu, Lambda expressions in CASL architectural specifications, in: T. Mossakowski, H.-J. Kreowski (Eds.), Recent Trends in Algebraic Development Techniques, 20th International Workshop, WADT 2010, Vol. 7137 of Lecture Notes in Computer Science, Springer, 2011, pp. 98–117.

[30] O. Kutz, T. Mossakowski, A modular consistency proof for Dolce, in: W. Burgard, D. Roth (Eds.), Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence and the Twenty-Third Innovative Applications of Artificial Intelligence Conference, AAAI Press; Menlo Park, CA, 2011, pp. 227–234.

[31] M. Roggenbach, L. Schröder, Towards trustworthy specifications I: Consistency checks, in: M. Cerioli, G. Reggio (Eds.), Recent Trends in Algebraic Specification Techniques, 15th International Workshop, WADT 2001, Vol. 2267 of Lecture Notes in Computer Science, Springer, 2001, pp. 305 – 327.

[32] M. Codescu, T. Mossakowski, C. Maeder, Checking conservativity with HETS, in: R. Heckel, S. Milius (Eds.), CALCO 2013, Vol. 8089 of Lecture Notes in Computer Science, Springer, 2013, pp. 315–321.

[33] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, D. Topic, SPASS version 2.0, in: A. Voronkov (Ed.), Automated Deduction, CADE-18, Vol. 2392 of Lecture Notes in Computer Science, Springer, 2002, pp. 275–279.

[34] A. Riazanov, A. Voronkov, The design and implementation of VAMPIRE, AI Communications 15 (2-3) (2002) 91–110.

[35] C. Benzmüller, L. C. Paulson, F. Theiss, A. Fietzke, Leo-II - a cooperative automatic theorem prover for classical higher-order logic (system description), in: A. Armando, P. Baumgartner, G. Dowek (Eds.), IJCAR, Vol. 5195 of Lecture Notes in Computer Science, Springer, 2008, pp. 162–170.

[36] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL — A proof assistant for higher-order logic, Vol. 2283 of Lecture Notes in Computer Science, Springer, 2002.

[37] J. A. Goguen, G. Roşu, Institution morphisms, Formal Aspects of Computing 13 (3-5) (2002) 274–307.

[38] R. Diaconescu, Grothendieck institutions, Applied Categorical Structures 10 (2002) 383–402.

[39] T. Mossakowski, Comorphism-based Grothendieck logics, in: K. Diks, W. Rytter (Eds.), Mathematical Foundations of Computer Science, Vol. 2420 of Lecture Notes in Computer Science, Springer, 2002, pp. 593–604.

[40] T. Mossakowski, S. Autexier, D. Hutter, Extending development graphs with hiding, in: Fundamental Approaches to Software Engineering, Vol. 2029 of Lecture Notes in Computer Science, Springer, 2001, pp. 269–283.

[41] T. Weber, Bounded model generation for Isabelle/HOL, Electronic Notes in Theoretical Computer Science 125 (3) (2005) 103–116.

[42] P. Baumgartner, A. Fuchs, C. Tinelli, Darwin: A theorem prover for the model evolution calculus, in: S. Schulz, G. Sutcliffe, T. Tammet (Eds.), IJCAR Workshop on Empirically Successful First Order Reasoning (ESFOR (aka S4)), Electronic Notes in Theoretical Computer Science, 2004, p. 191.

[43] H. Ehrig, B. Mahr, Fundamentals of Algebraic Specification 1: Equations und Initial Semantics, Vol. 6 of Monographs in Theoretical Computer Science. An EATCS Series, Springer, 1985.

[44] T. Mossakowski, Two "functional programming" sublanguages of CASL, in [1] (Mar. 1998).
URL `http://www.cofi.info/old/Notes/L-9/`

[45] T. Brunet, Génération automatique de code à partir de spécifications formelles, Master's thesis, Université de Poitiers (2003).

[46] C. A. R. Hoare, Proof of correctness of data representations, Acta Informatica 1 (1972) 271–281.

[47] R. Milner, An algebraic definition of simulation between programs, in: D. C. Cooper (Ed.), IJCAI, William Kaufmann, 1971, pp. 481–489.

[48] D. R. Smith, Designware: Software development by refinement, in: Proc. 8th Conference on Category Theory and Computer Science, CTCS'99, Vol. 29 of Electronic Notes in Theoretical Computer Science, 1999, pp. 275–287.

[49] M. Bidoit, D. Sannella, A. Tarlecki, Global development via local observational construction steps, in: Proc. 27th Intl. Symp. on Mathematical Foundations of Computer Science, Vol. 2420 of Lecture Notes in Computer Science, Springer, 2002, pp. 1–24.

[50] M. Bidoit, D. Sannella, A. Tarlecki, Observational interpretation of CASL specifications, Mathematical Structures in Computer Science 18 (2) (2008) 325–371.

Appendix A. Proofs

Proof of Theorem 5.2:

Assume that **arch spec** ASP = **units** UDD^+ **result** UE. Moreover, assume $\Gamma_v^\emptyset \vdash UDD^+ ::_c \Gamma_v$ and $\vdash UDD^+ :: \Gamma_{gen}, \Gamma$. We prove that if $USP = \mathcal{S}_{ASP}$ then $\Gamma_{gen}, \Gamma \vdash UE :: USP$.

- If $UE = UT$, it suffices to show that for any family of models $\mathcal{M}$ compatible with the diagram Γ', $M_B \models \mathcal{S}_{ASP}(UT)$, where $\Gamma_{gen}, \Gamma \vdash UT :: \Gamma', B$. We will proceed by induction on the structure of UT. Notice that in the case of unit application and union, $M_B \models \mathcal{S}_{amalg}(B)$ using the amalgamation properties ensured by the successful run of the extended static semantics.
  - If $UT = UN$, then by definition $\mathcal{S}_{ASP}(UT) = SP$ where $UN : SP$ in ASP and by construction of Γ', the node UN is labeled with SP. This means that M_{UN} must satisfy $SP = \mathcal{S}_{ASP}(UT)$.
  - If $UT = UT_1$ **and** UT_2, then by construction B has two incoming edges from A_1 and A_2, where $\Gamma_{gen}, \Gamma \vdash UT_i :: \Gamma_i, A_i, i = 1, 2$. By the inductive hypothesis $M_{A_i} \models \mathcal{S}_{ASP}(UT_i)$. It follows easily that $M_B \models \mathcal{S}_{ASP}(UT_1)$ **and** $\mathcal{S}_{ASP}(UT_2)$.
  - If $UT = F[UT'$ **fit** $\sigma]$, then, using the notation of the rule, B has two incoming edges from A_r and A_a by construction, A_r is labeled with SP_r, and $M_{A_a} \models \mathcal{S}_{ASP}(UT')$ by inductive hypothesis. It is easy to check that $M_B \models \mathcal{S}_{ASP}(UT)$.
  - If $UT = UT'$ **with** σ then, using the notation of the rule, B has an incoming edge from A by construction and $M_A \models \mathcal{S}_{ASP}(UT')$, which implies $M_B \models \mathcal{S}_{ASP}(UT)$.
  - If $UT = UT'$ **hide** σ then, using the notation of the rule, B has an outgoing edge to A by construction and $M_A \models \mathcal{S}_{ASP}(UT')$, which implies $M_B \models \mathcal{S}_{ASP}(UT)$.
  - if UT = **local** `UDECL` **within** UT', then we must show that $M_B \models \mathcal{S}_{ASP}(UT)$ which by definition is $\mathcal{S}_{\Gamma_v}(UT')$ where Γ_v extends the verification context of UT' according to `UDECL`. By induction for UT', since the node of UT is the one of UT' where the local declarations in `UDECL` are deleted, we get that $M_B \models \mathcal{S}_{\Gamma_v}(UT')$.
- If $UE = \lambda X : SP\, .\, UT$, we have $M_B \models \mathcal{S}_{ASP}(UT)$ where $\Gamma_{gen}, \Gamma \vdash UT :: \Gamma', B$ and since SP is obviously equivalent with itself, we have that $\Gamma_{gen}, \Gamma \vdash UE :: SP \to \mathcal{S}_{ASP}(UT)$.

□

Proof of Theorem 5.4:

The inclusion from left to right follows from the generalized version of Thm. 5.1 and Thm. 5.3. Concerning the converse inclusion, let $\vdash ASP \rhd \Box$ and $\vdash ASP ::_c USP$.

We distinguish the cases of ASP being consistent or not. If ASP is inconsistent, the specification of some non-generic unit UN in ASP must be inconsistent, because all generic unit specifications have been assumed to be consistent. But then UN is being used in the result unit expression of ASP (because the framework requires that all non-generic units not being used in the result unit expression of ASP are consistent), causing USP to be inconsistent as well. Hence, both sides of the equation are the empty set.

Let us now assume that ASP is consistent, and let M be a model of USP. For simplicity, we consider M to be non-parametric; the parametric case is treated similarly by considering all the possible applications to argument units. Let UT be the result unit term of ASP. For each occurrence of a subterm oT of UT in UT, we construct a model M_{oT} satisfying $\mathcal{S}_{ASP}(oT)$, where the latter is built in the context of the unit definitions of ASP. In parallel, we define partial functions M_F for parametric units F for one or more argument tuples. Let h compute the height of a unit term. We proceed by nested induction over $h(UT) - h(oT)$ within unit terms and the dependency structure of the unit definitions, which means by moving from each subterm to its immediate subterms and in the order of occurrence of defined units in the unit term.

The induction base is the result unit term UT; we put $M_{UT} := M$. For the induction step, we make a case distinction:

- if T is a unit name coming from a unit declaration, then there are no subterms;

- if T is a unit name coming from a unit definition $T = UT'$, we proceed with UT' with the model $M_{UT'} = M_T$;
- if T is T_1 **and** T_2, let $M_{oT_i} := M_T|_{\sigma_i}$, where σ_i is the inclusion of the signature of T_i into that of T. From $M_T \in \mathbf{Mod}(\mathcal{S}_{ASP}(T))$, we easily get $M_{oT_i} \in \mathbf{Mod}(\mathcal{S}_{ASP}(T_i))$;
- if T is T_1 **with** σ, let $M_{oT_1} := M_T|_{\sigma}$. Again, from $M_T \in \mathbf{Mod}(\mathcal{S}_{ASP}(T))$, we easily get $M_{oT_1} \in \mathbf{Mod}(\mathcal{S}_{ASP}(T_1))$;
- if T is T_1 **hide** σ, $\mathcal{S}_{ASP}(T)$ is $\mathcal{S}_{ASP}(T_1)$ **hide** σ. Hence, by $M_T \in \mathbf{Mod}(\mathcal{S}_{ASP}(T))$, there is some (not necessarily unique) $M_1 \in \mathbf{Mod}(\mathcal{S}_{ASP}(T_1))$ with $M_1|_{\sigma} = M_T$. Put $M_{oT_1} := M_1$;
- if $T = F[T_1 \textbf{ fit } \sigma_1] \ldots [T_n \textbf{ fit } \sigma_n]$, then we know that $\mathcal{S}_{ASP}(T) = \{SP \textbf{ with } \sigma\}$ **and** $\mathcal{S}_{ASP}(UT_1)$ **with** $\iota_1; \iota'$ **and** $\ldots$ **and** $\mathcal{S}_{ASP}(UT_n)$ **with** $\iota_n; \iota'$ **and** $\mathcal{S}_{amalg}(UT)$. Hence, for $i = 1, \ldots, n$, we define $M_{oT_i} := M|_{\iota_i;\iota'}$. We also define the action of M_F on the arguments thus defined: $M_F(M_{oT_1}|_{\sigma_1}, \ldots, M_{oT_n}|_{\sigma_n}) := M_T$. (Note that by assumption, F is applied only once.) If F is the name of a generic unit coming from a unit definition, we repeat the procedure for the term FM, defining F with $F_{FM} := M_T$.

We need to show that if oT_1 and oT_2 are two occurrences of a term T in UT, then $M_{oT_1} = M_{oT_2}$. Thus, we can define $M_T = M_{oT}$ for each subterm T of UT and for an arbitrary occurrence oT of T in UT and we also ensure that M_F is well-defined, for any generic unit F.

Assume that oT_1 and oT_2 are two occurrences of T in UT. Let UT' be the least subterm of UT that contains both oT_1 and oT_2, and notice that UT' can be either a unit application $F[UT_1] \ldots [UT_n]$ with oT_1, oT_2 subterms of UT_i and UT_j respectively, for some $i, j \in 1, \ldots, n$ or a unit amalgamation UT_1 **and** $\ldots$ **and** UT_n with oT_1, oT_2 subterms of UT_i and UT_j respectively, for some $i, j \in 1, \ldots, n$. In both cases, by Def. 5.3 we have that $M_{UT'} \models \mathcal{S}_{amalg}(UT')$. Since the framework assumes that no generic unit can be applied more than once, T cannot be a unit application $G[T']$. If that were the case, the diagram $D_{UT'}$ of the unit term UT' would contain a sub-diagram like the following:

$$\begin{tikzcd}
 & \Sigma_a(\Delta) & & \Sigma_a(\Delta) & \\
\Sigma_r \arrow[ur, "\sigma(\Delta)"] & & \Sigma_a \arrow[ul, "\iota"'] \arrow[ur, "\iota"] & & \Sigma_r \arrow[ul, "\sigma(\Delta)"'] \\
 & \Sigma_f \arrow[ul, "\Delta"] \arrow[ur, "\sigma"'] & & \Sigma_f \arrow[ul, "\sigma"] \arrow[ur, "\Delta"'] &
\end{tikzcd}$$

where the nodes labeled with $\Sigma_a(\Delta)$ correspond to the two occurences oT_1 and oT_2, $\Delta : \Sigma_f \to \Sigma_r$ is the inclusion of the signature of the formal parameter of G in the signature of the result, σ is the fitting morphism to the signature of the actual parameter T' and $(\Sigma_a(\Delta), \sigma(\Delta), \iota)$ is the selected pushout of the span (cf. Def. 4.1). Then the symbols of Σ_r that are not in Σ_f do not have a common origin in the diagram, and therefore their image along $\sigma(\Delta)$ gives symbols that do not neccesarily share. In all other possible cases, we can observe that the nodes corresponding to oT_1 and oT_2 in the diagram $D_{UT'}$ of the unit term UT' have symbols with the same origin. This implies that M_{oT_1} and M_{oT_2} must be equal.

We now construct a model of ASP as follows: non-parametric unit names A are interpreted as M_A (if this is defined), while parametric units F are interpreted as M_F whenever this is defined for specific arguments. The interpretations of the remaining non-parametric units and remaining applications of parametric units to arguments does not affect UT at all, we can hence take them from any model of ASP (which exists by consistency of ASP). This yields a model of ASP that interprets UT as M. □

Proof of Theorem 5.5:

Assume that **arch spec** ASP = **units** UDD^+ **result** UE. Moreover, assume $\vdash UDD^+ :: \Gamma_{gen}, \Gamma$. We prove that for any unit expression UE' occurring in ASP (either in a unit definition or in the result unit expression) if we have $\Gamma_{gen}, \Gamma \vdash UE' :: USP''$ for some USP'', then $\Gamma_v^{\emptyset} \vdash UDD^+ ::_c \Gamma_v$, $\Gamma_v \vdash UE' ::_c USP'$ where $USP' = \mathcal{S}_{\Gamma_v}(UE')$ and $USP' \rightsquigarrow USP''$. Then the theorem follows by taking $UE' = UE$.

We proceed by induction on the sequence of unit definitions in UDD^+ and then by induction on the structure of UE'.

So, suppose that the claim holds for all unit expressions in the preceding unit definitions and for all subexpressions of UE'. We make a case analysis on the form of UE'.

If UE' is just a unit name UN, we have two possibilities: either UN is the name of a defined unit and then the claim follows directly from the inductive hypothesis, or UN is the name of a declared unit. In the latter case let USP' be its specification in ASP. We have by the hypothesis that $\Gamma_{gen}, \Gamma \vdash UN :: USP$ for some USP. By definition of $\mathcal{S}_{ASP}(UN)$, we have that $\Gamma_v \vdash UN ::_c USP'$. We have to show that $USP \rightsquigarrow USP'$. Let ASP' be the architectural specification obtained from ASP by replacing its result unit expression with UN. Since $\vdash ASP' ::_c USP'$, by Thm. 5.4 we have $ProjRes(\mathbf{Mod}(ASP')) = \mathbf{Mod}(USP')$. By $\vdash ASP' :: USP$ and soundness of $\vdash \_ :: \_$ we have $ProjRes(\mathbf{Mod}(ASP')) \subseteq \mathbf{Mod}(USP)$. Hence, $\mathbf{Mod}(USP') \subseteq \mathbf{Mod}(USP)$.

Of the other cases, we present here only the case of one argument unit application, as the others are similar. Therefore, let us assume that $UE = F[UE']$ where $F : SP \to SP'$ in Γ_{gen}. Since $\Gamma_{gen}, \Gamma \vdash UE :: USP$ for some USP, we can easily show that $\Gamma_{gen}, \Gamma \vdash UE' :: SP$ (this follows directly from the verification condition of the proof calculus $\vdash \_ :: \_$ for unit applications). By the inductive hypothesis, we get that $\Gamma_v \vdash UE' ::_c \mathcal{S}_{ASP}(UE')$ and moreover $SP \rightsquigarrow \mathcal{S}_{ASP}(UE')$, which is precisely the verification condition of the proof calculus $\vdash \_ ::_c \_$ for $F[UE']$. This means that $\Gamma_v \vdash UE ::_c \mathcal{S}_{ASP}(UE)$, where $\mathcal{S}_{ASP}(UE)$ is as given in Def. 5.3. We need to show that $USP \rightsquigarrow \mathcal{S}_{ASP}(UE)$, and this is done with the same argument as for the case of unit names. □

Proof of Theorem 5.7:

The proof follows by induction on the structure of SPR.

If $SPR = USP$, then according to the rules of the model semantics in Fig. 12, we have $\vdash SPR \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U, U) | U \in Unit(USP)\}$. With the proof calculus rule we have $\vdash SPR ::_c (USP, USP)$ and it is obvious that $\mathcal{R} \models (USP, USP)$.

If $SPR = USP$ **refined via** σ **to** SPR' with $\vdash SPR ::_c S$, then according to the proof calculus rule we have that $\vdash SPR' ::_c (USP', BSP)$, $USP \overset{\sigma}{\rightsquigarrow} USP'$ and $S = (USP, BSP)$. By the inductive hypothesis we get that there is $\mathcal{R}'$ such that $\vdash SPR' \Rightarrow \mathcal{R}'$ and $\mathcal{R}' \models (USP', BSP)$. Then $\vdash SPR \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U|_\sigma, BM) | (U, BM) \in \mathcal{R}'\}$ and the refinement condition $USP \overset{\sigma}{\rightsquigarrow} USP'$ ensures that $\mathcal{R} \models S$.

If $SPR =$ **arch spec** ASP, then according to the proof calculus rule we have that for any unit declaration of the form $UN_i : SPR_i$ in ASP, $\vdash SPR_i ::_c (USP_i, BSP_i)$. Let ASP' be the architectural specification obtained by replacing SPR_i with USP_i for each unit declaration $UN_i : SPR_i$ in ASP and let UE denote the result unit of ASP. Then we define $\mathcal{SPM}(UN)$ for any unit UN of ASP as $\mathcal{SPM}(UN) = BSP_i$, if UN is a declared unit UN_i of ASP, and $\mathcal{SPM}(UN) = \mathcal{S}_{\Gamma_v}(UE')$ if $UN = UE'$ is a unit definition of ASP, where Γ_v is the verification environment of ASP'. This allows us to define $S = (\mathcal{S}_{\Gamma_v}(UE), \mathcal{SPM})$. By inductive hypothesis we have $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and $\mathcal{R}_i \models (USP_i, BSP_i)$ for each i. Then $\vdash SPR \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U, \pi_2(RE)) | \; RE(UN_i) \in \mathcal{R}_i$ for the defined units UN_i, U combines the units in $\pi_1(RE)$ according to $UE\}$ and $\mathcal{R} \models S$ by Thm. 5.3, where $\mathcal{AM} = \mathcal{R}$ and $\vdash ASP ::_c \mathcal{S}_{\Gamma_v}(UE)$ by the rules of the proof calculus (Fig. 16).

If $SPR = \{UN_i$ **to** $SPR_i\}_{i \in \mathcal{J}}$ with $\vdash SPR ::_c S$, then according to the proof calculus rule we have that $S = \{UN_i \mapsto S_i\}_{i \in \mathcal{J}}$ and $\vdash SPR_i ::_c S_i$ for $i \in \mathcal{J}$. By the inductive hypothesis we get that $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and $\mathcal{R}_i \models S_i$. Then $\vdash SPR \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{R \mid dom(R) = \{UN_i\}_{i \in \mathcal{J}}, R(UN_i) \in \mathcal{R}_i$ for each $i \in \mathcal{J}\}$ and notice that $\mathcal{R} \models S$ by definition.

Finally, if $SPR = SPR_1$ **then** SPR_2, with the proof calculus rule we get $\vdash SPR_i ::_c S_i$ for $i = 1, 2$ such that $S = S_1; S_2$ and by inductive assumption we have $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and $\mathcal{R}_i \models S_i$, for $i = 1, 2$. It follows (by case analysis, sketched below) that $\mathcal{R} = \mathcal{R}_1; \mathcal{R}_2$ is defined and $\mathcal{R} \models S$.

Let us assume that $\vdash SPR_1 \rhd (U\Sigma, BstC)$; the other cases are similar. Then $S_1 = (USP, \mathcal{SPM})$, $\vdash SPR_2 \rhd \{UN_i \mapsto R\Sigma_i\}_{i \in \mathcal{J}}$ with $UN_i \in dom(BstC)$ for $i \in \mathcal{J}$ and $S_2 = \{UN_i \to S_i\}_{i \in \mathcal{J}}$. Let $R_2 = \{UN_i \to R_i\}_{i \in \mathcal{J}} \in \mathcal{R}_2$ and assume there exists $R_1 = (U, BE) \in \mathcal{R}_1$. (If $\mathcal{R}_1$ were empty, the result follows by noticing that $\mathcal{R} = \mathcal{R}_1; \mathcal{R}_2$ is empty as well [25] and that $\mathcal{R} \models S_1; S_2$ by distinguishing the subcases when an inconsistent specification in $\mathcal{SPM}$ is further refined or not.) To ease understanding, we can assume that the second refinement further refines only one component UN of the first refinement, and that the component UN corresponds to a leaf in the branching tree. This means that $dom(R_2) = \{UN\}$ and moreover $BstC(UN) = U\Sigma$. Then $\mathcal{SPM}(UN) = USP$ and $S_2(UN) = (USP', BSP')$. Since $R_2 \models S_2$ we have that $R_2(UN) = (U', BM'')$ such that $U' \in Unit(USP')$. Since $S_1; S_2$ is defined, the associated verification condition

[25] In the case of simple refinement, this does not follow immediately from the definition of composition of refinement relations, but requires to notice that if $\mathcal{R}_1$ is empty, then so is $\mathcal{R}_2$, and their composition is as well empty.

$USP \rightsquigarrow USP'$ holds and thus U' is also in $Unit(USP)$. Thus we can replace $BE(UN)$ with U' in BE to obtain a branching environment BE' that satisfies $\mathcal{SPM}$ and therefore there exists $(U'', BE') \in \mathcal{R}_1$. The composition $\mathcal{R}_1;\mathcal{R}_2$ is then defined and it is easy to see that $\mathcal{R}_1;\mathcal{R}_2 \models S_1;S_2$. □

Proof of Theorem 5.8: By induction on the structure of SPR.

If $SPR = USP$, then $\vdash USP ::_e ASP$ where **arch spec** $ASP =$ **units** $UN : USP$ **result** UN. By definition, $\vdash USP \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U, U) | U \in Unit(USP)\}$. From the rules of the model semantics for architectural specifications, we know that $\vdash ASP \Rightarrow \mathcal{AM}$, where $\mathcal{AM} = \{(U, \{UN \mapsto U\}) | U \in Unit(USP)\}$. It is obvious that $\mathcal{AM}$ and $\mathcal{R}$ correspond up to unit names.

If $SPR = USP$ **refined via** σ **to** SPR' and by the hypothesis $\vdash SPR ::_e ASP$, then according to the proof calculus we know that $\vdash SPR' ::_e ASP'$, where ASP' is of the form **units** UN : **arch spec** ASP'' **result** (UN **hide** σ), as in the rule for simple refinements. By the induction hypothesis we get $\vdash SPR' \Rightarrow \mathcal{R}'$ and $\mathcal{R}'$ and $\mathcal{AM}'$ correspond up to unit names, where $\vdash ASP' \Rightarrow \mathcal{AM}'$. Moreover, the refinement condition $USP \overset{\sigma}{\rightsquigarrow} \mathcal{S}_{ASP'}$ holds. Let $(U, BM) \in \mathcal{R}'$. Since $\mathcal{R}'$ and $\mathcal{AM}'$ correspond up to unit names we get $U \models \mathcal{S}_{ASP'}$ and therefore $U|_\sigma \models USP$. Then by definition we have $\vdash SPR \Rightarrow \mathcal{R}$, where $\mathcal{R} = \{(U|_\sigma, BM) | (U, BM) \in \mathcal{R}'\}$. From the rules of the model semantics for architectural specifications, we get $\vdash ASP \Rightarrow \mathcal{AM}$ where $\mathcal{AM} = \{(U|_\sigma, BM) | (U, BM) \in \mathcal{AM}'\}$ and since $\mathcal{R}'$ and $\mathcal{AM}'$ correspond up to unit names, so do $\mathcal{R}$ and $\mathcal{AM}$.

If $SPR =$ **arch spec** ASP, then by the hypothesis for any unit declaration $UN_i : SPR_i$ in ASP, $\vdash SPR_i ::_e ASP_i$. By the induction hypothesis this means $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and $\mathcal{R}_i$ corresponds up to unit names with the model class $\mathcal{AM}_i$ of ASP_i. This means that for any $(U, BM) \in \mathcal{R}_i$, $U \models \mathcal{S}_{ASP_i}$. Then for any choice of models $(U_i, BM_i) \in \mathcal{R}_i$ for each i, we obtain on one hand a unit environment for ASP by projecting the models on the first component and we can combine them according to the result unit expression of ASP to obtain a model U. On the other hand, with the same construction we get a unit environment for ASP' (where ASP' is the architectural specification defined in the corresponding proof rule of the calculus) and since the result unit expression is the same, we get the same model U. Since the choice of models for $\mathcal{R}_i$ was arbitrary, we get this for all possible refined-unit static contexts, and thus for all models of ASP and ASP'. Then the models of ASP and ASP' correspond up to unit names.

If $SPR = \{UN_i$ **to** $SPR_i\}_{i \in \mathcal{J}}$, then by the hypothesis we know $\vdash SPR_i ::_e \mathcal{VS}_i$ and by the inductive hypothesis we have $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and $\mathcal{R}_i$ corresponds with the model class of $\mathcal{VS}_i$. Then using the rules of the model semantics we get $\vdash SPR \Rightarrow \mathcal{R} = \{UN_i \to \mathcal{R}_i\}_{i \in \mathcal{J}}$ and $\mathcal{R}$ corresponds with the model class of $\{UN_i \to \mathcal{VS}_i\}_{i \in \mathcal{J}}$.

Finally, if $SPR = SPR_1$ **then** SPR_2 and $\vdash SPR ::_c \mathcal{VS}$, then by the corresponding proof rule $\vdash SPR_i ::_e \mathcal{VS}_i$ for $i = 1, 2$ and $\mathcal{VS} = \mathcal{VS}_1;\mathcal{VS}_2$. By the induction hypothesis we get $\vdash SPR_i \Rightarrow \mathcal{R}_i$ and $\mathcal{R}_i$ corresponds up to unit names to the model class of $\mathcal{VS}_i$, for $i = 1, 2$. We show that $\mathcal{R}_1;\mathcal{R}_2$ is defined and it corresponds up to unit names to the model class of $\mathcal{VS}$ by case analysis on the refinement signature $R\Sigma_1$ of SPR_1.

If $R\Sigma_1 = (U\Sigma, U\Sigma')$ then $R\Sigma_2 = (U\Sigma', B\Sigma)$. Then, $\mathcal{VS}_1$ and $\mathcal{VS}_2$ are both architectural specifications and $\mathcal{VS}_1$ has only one unit UN labelled with a unit specification USP (possibly inside several architectural levels). Therefore the assignments in $\mathcal{R}_2$ are branching assignments of the form (U, BM) and by the inductive hypothesis, we know that $U \models \mathcal{S}_{\mathcal{VS}_2}$. Since $\mathcal{VS}_1;\mathcal{VS}_2$ is defined, this implies that $U \models USP$ and there must be an assignment (U_0, U) of $\mathcal{R}_1$, because $\mathcal{VS}_1$ is architectural and therefore U generates a unit environment for $\mathcal{VS}_1$ in which the result unit of $\mathcal{VS}_1$ is evaluated to a model U_0, and $\mathcal{AM}_1$ and $\mathcal{R}_1$ correspond up to unit names. This means that $\mathcal{R}_1;\mathcal{R}_2$ is defined. Moreover, if we replace USP in $\mathcal{VS}_1$ with the result of evaluating $\mathcal{S}_{\mathcal{VS}_2}$ in the context given by $\mathcal{VS}_2$, we get that the model U_0 satisfies $\mathcal{S}_{\mathcal{VS}_1}[UN/\mathcal{S}_{\mathcal{VS}_2}]$, which ensures that $\mathcal{R}_1;\mathcal{R}_2$ corresponds up to unit names to the model class of $\mathcal{VS}_1;\mathcal{VS}_2$. The other cases are similar but tedious and we omit them. □

Proof of Theorem 5.9:

By induction on the structure of SPR.

If $SPR = USP$, then $\vdash SPR ::_e ASP$ and $\vdash ASP \Rightarrow \mathcal{AM}$, where $\mathcal{AM} = \{(U, \{UN \mapsto U\}) | U \in Unit(USP)\}$. By definition, $\vdash USP \Rightarrow \mathcal{R}$ where $\mathcal{R} = \{(U, U) | U \in Unit(USP)\}$ and therefore the one-to-one correspondence up to unit names is obvious.

If $SPR = USP$ **refined via** σ **to** SPR' and we know that $\vdash SPR \Rightarrow \mathcal{R}$, then from the rules of the model semantics we get that $\vdash SPR' \Rightarrow \mathcal{R}'$ and for every $(U, BM) \in \mathcal{R}'$, $U|_\sigma \in Unit(USP)$. By the inductive hypothesis we have that $\vdash SPR' ::_e ASP'$ and there is a one-to-one correspondence between $\mathcal{R}'$ and $\mathcal{AM}'$, where $\vdash ASP' \Rightarrow \mathcal{AM}'$. We need to prove that $USP \overset{\sigma}{\rightsquigarrow} \mathcal{S}_{ASP'}$, so let $V \models \mathcal{S}_{ASP'}$. We know that there is a branching model BM such that $(V, BM) \in \mathcal{AM}'$

and we know that (V, BM) corresponds up to unit names with an assignment of $\mathcal{R}'$, which gives us that $V|_\sigma \models USP$. Thus $\vdash SPR ::_e ASP$. Finally, notice that the model class of ASP and $\mathcal{R}$ correspond up to unit names by construction of ASP, definition of $\mathcal{R}$ and correspondence up to unit names between $\mathcal{AM}'$ and $\mathcal{R}'$.

If $SPR =$ **arch spec** ASP and $\vdash ASP \Rightarrow \mathcal{R}$, then from the rule in the model semantics we get that $\vdash SPR_i \Rightarrow \mathcal{R}_i$ for each unit declaration $UN_i : SPR_i$ in ASP. By induction we get that $\vdash SPR_i ::_e ASP_i$ and model class of ASP_i correspond up to unit names with $\mathcal{R}_i$. It follows that $\vdash SPR ::_e ASP'$ and the model of ASP' corresponds up to unit names with $\mathcal{R}$ with the same argument as in the soundness proof.

If $SPR = \{UN_i$ **to** $SPR_i\}_{i\in\mathcal{J}}$ and $\vdash SPR \Rightarrow \mathcal{R}$, then from the rule in the model semantics we get $\vdash SPR_i \Rightarrow \mathcal{R}_i$. By induction we get $\vdash \mathcal{R}_i ::_e \mathcal{VS}_i$ and with the proof calculus rule we get $\vdash \mathcal{R} ::_e \mathcal{VS} = \{UN_i$ **to** $\mathcal{VS}_i\}_{i\in\mathcal{J}}$. The correspondence between $\mathcal{R}$ and the model of $\mathcal{VS}$ follows immediately by definition and the correspondence between $\mathcal{R}_i$ and the model of $\mathcal{VS}_i$, for each $i \in \mathcal{J}$.

Finally, if $SPR = SPR_1$ **then** SPR_2, and $\vdash SPR \Rightarrow \mathcal{R}$. then according to the rule in the model semantics we get that $\vdash SPR_i \Rightarrow \mathcal{R}_i$ for $i = 1, 2$ and $\mathcal{R} = \mathcal{R}_1; \mathcal{R}_2$. By induction we get that $\vdash SPR_i ::_e \mathcal{VS}_i$ and $\mathcal{R}_i$ corresponds with the model class of $\mathcal{VS}_i$ for $i = 1, 2$. We need show that $\mathcal{VS} = \mathcal{VS}_1; \mathcal{VS}_2$ is defined.

We will consider only the case when $\mathcal{VS}_1$ is an architectural specification with more than one component; the other cases are similar. We know that $\mathcal{VS}_2$ is an enhanced verification map since the refinements compose. Let $UN \in dom(\mathcal{VS}_2)$ and assume the specification of UN in $\mathcal{VS}_1$ is a unit specification USP (the case when the specification of U is architectural reduces to this case) and $\mathcal{VS}_2(UN) = ASP'$. Let $U \models \mathcal{S}_{ASP'}$ and then by Thm. 5.4 we know that there is a BM such that (U, BM) is an architectural model of ASP'. Since the models of $\mathcal{VS}_2$ correspond up to unit names with $\mathcal{R}_2$ and $\mathcal{R}_1; \mathcal{R}_2$ is defined, there must be an assignment (U', BM') in $\mathcal{R}_1$ such that $BM'(UN) = U$. We know that (U, BM) corresponds up to unit names with a $\mathcal{VS}_1$-model, which implies $U \models USP$, so the refinement condition holds. To prove the correspondence up to unit names between $\mathcal{R}$ and architectural models of $\mathcal{VS}$, let $(U, BM) \in \mathcal{VS}$. Notice that by construction of $\mathcal{VS}$, BM not only produces a unit environment for $\mathcal{VS}$ but also a unit environment for $\mathcal{VS}_1$ such that evaluating the result unit expression of $\mathcal{VS}$ and $\mathcal{VS}_1$ yields the same result. This shows that we can write (U, BM) as a composition between a model corresponding to an assignment in $\mathcal{R}_1$ and a model corresponding up to unit names to an assignment in $\mathcal{R}_2$. □

Proof of Theorem 5.10:

By induction on the structure of SPR.

If $SPR = USP$, the result follows directly from the soundness of the calculi for consistency and conservativity.

If $SPR = USP$ **refined via** σ **to** SPR', by definition, $\vdash cons(SPR)$ holds if $\vdash cons(SPR')$ holds. By the induction hypothesis, the refinement relation $\mathcal{R}'$ of SPR' is non-empty. Since $\vdash SPR ::_c \square$, it follows that $USP \overset{\sigma}{\rightsquigarrow} USP'$, where $\vdash SPR' ::_c (USP', BSP)$. This means that $U|_\sigma \in Unit(USP)$ for any $(U, BM) \in \mathcal{R}'$. It follows that the refinement relation $\mathcal{R} = \{(U|_\sigma, BM) \mid (U, BM) \in \mathcal{R}'\}$ of SPR is non-empty.

If $SPR = \{UN_i$ **to** $SPR_i\}_{i\in\mathcal{J}}$, by the hypothesis we get that $\vdash cons(SPR_i)$ and $\vdash SPR_i ::_c \square$, for each $i \in \mathcal{J}$. By the induction hypothesis, for each $i \in \mathcal{J}$, the refinement relation $\mathcal{R}_i$ of SPR_i is non-empty. We can therefore take an assignment from each such refinement relation and combine them as in the rule for component refinements in Fig. 12 to obtain a refinement assignment in the refinement relation of SPR.

If $SPR = SPR_1$ **then** SPR_2 we have two sub-cases.

Let us first assume that SPR_1 contains branchings. By the hypothesis we get that $\vdash cons(SPR_i)$ and $\vdash SPR_i ::_c \square$, for any $i \in \{1, 2\}$. By the induction hypothesis, we get that the model classes $\mathcal{R}_1$ and $\mathcal{R}_2$ of SPR_1 and SPR_2 respectively are non-empty and let $R_i \in \mathcal{R}_i$, for $i = 1, 2$. Since $\vdash SPR ::_c \square$ and SPR_1 and SPR_2 are consistent, using the soundness of the proof calculus we have that $\mathcal{R}_1; \mathcal{R}_2$ is defined and $\vdash SPR \Rightarrow \mathcal{R}_1; \mathcal{R}_2$. Since the composition $\mathcal{R}_1; \mathcal{R}_2$ is defined, there must be a model $R'_1 = (U', BE[R_2]) \in \mathcal{R}_1$, where $R_1 = (U, BE)$ and the definition of $BE[R_2]$ has been given in the second main case of Def. 4.6. Using the definition of the composition, we get that $(U', BE\langle R_2\rangle)$ (as defined in the second main case of Def. 4.6) is a refinement assignment in $\mathcal{R}_1; \mathcal{R}_2$, which is thus non-empty.

Let us then consider the case when SPR_1 does not contain branchings. By hypothesis we get that $\vdash cons(SPR_2)$ and $\vdash SPR_i ::_c \square$, for any $i \in \{1, 2\}$. By induction we get that the refinement relation $\mathcal{R}_2$ of SPR_2 is non-empty. Notice that since SPR_1 does not contain branchings, it must be the case that $\vdash SPR_1 \rhd (U\Sigma, U\Sigma')$ and $\vdash SPR_2 \rhd (U\Sigma', B\Sigma)$. Then $\vdash SPR_1 ::_c (USP, USP')$ and $\vdash SPR_2 ::_c (USP'', BSP)$ and moreover $USP' \rightsquigarrow USP''$. This means that if (U, BM) is an assignment in $\mathcal{R}_2$, U is not only a USP''-model but also a USP'-model and, since $\mathcal{R}_1 \models (USP, USP')$, the constructor κ associated with SPR_1 is defined on U. We get thus the assignment $(\kappa(U), BM)$ in $\mathcal{R}_1; \mathcal{R}_2$.

If $SPR =$ **arch spec** ASP, by the induction hypothesis, SPR_i has a model, for all $UN_i : SPR_i$ in ASP. This gives us a unit environment for ASP by projecting each of the models of SPR_i to the first component and we can combine the units in the way described by the result unit of ASP to get a model of ASP because ASP is statically correct.

□

Proof of Theorem 5.11:

By structural induction on SPR.

If $SPR = USP$, the result follows from the completeness of the calculus for structured specifications and conservativity of extensions of structured specifications.

If $SPR = USP$ **refined via** σ **to** SPR', let $\mathcal{R}$ be the refinement relation of SPR. From the corresponding rule of the model semantics, $\mathcal{R} = \{(U|_\sigma, BM) \mid (U, BM) \in \mathcal{R}'\}$ where $\mathcal{R}'$ is the refinement relation of SPR'. Since $\vdash SPR ::_c \square$ we also get $\vdash SPR' ::_c \square$ and non-emptiness of $\mathcal{R}$ implies non-emptiness of $\mathcal{R}'$. We can apply the inductive hypothesis to get $\vdash cons(SPR')$, which implies $\vdash cons(SPR)$.

If $SPR = \{UN_i$ **to** $SPR_i\}_{i \in \mathcal{J}}$ and SPR has a non-empty refinement relation, then we can define refinement assignments of SPR_i by projecting a refinement assignment of SPR to the i-th component. Since correctness of SPR implies correctness of SPR_i, by the induction hypothesis we obtain $\vdash cons(SPR_i)$. With the rule for component refinements we get $\vdash cons(SPR)$.

If $SPR = SPR_1$ **then** SPR_2 and SPR has a non-empty refinement relation $\mathcal{R}$, let $R \in \mathcal{R}$. With the corresponding rule of the model semantics there must be assignments $R_1 \in \mathcal{R}_1$ and $R_2 \in \mathcal{R}_2$ such that $R = R_1; R_2$. By the induction hypothesis we get $\vdash cons(SPR_1)$ and $\vdash cons(SPR_2)$ and with the calculus rule for compositions of refinements we get $\vdash cons(SPR)$. Note that if SPR_1 has no branching, it suffices to use $\vdash cons(SPR_2)$.

If $SPR =$ **arch spec** ASP and ASP has a model class $\mathcal{AM}$, then by the rule of the model semantics we know that for any unit declaration $UN_i : SPR_i$ of ASP, SPR_i has a non-empty refinement relation $\mathcal{R}_i$. We can apply the induction hypothesis to get $\vdash cons(SPR_i)$ and with the rule for architectural specifications we get $\vdash cons(ASP)$.

□

Appendix B. The Specification of the Steam Boiler Control System To be included in the electronic version only

```
library UserManual/Sbcs

%author Michel Bidoit <bidoit@lsv.ens−cachan.fr>
%date 20 Oct 2003
%display(__half %LATEX __/2)%
%display(__square %LATEX __^2)%

from Basic/Numbers get Nat

from Basic/StructuredDatatypes get Set

from Basic/StructuredDatatypes get TotalMap

spec Value =
    %% At this level we don't care about the exact specification of values.
    Nat
then sort   Nat < Value
    ops     __+__ : Value × Value → Value, assoc, comm, unit 0;
            __−__ : Value × Value → Value;
            __*__ : Value × Value → Value, assoc, comm, unit 1;
            __/2, __^2 : Value → Value;
            min, max : Value × Value → Value
    preds __<__, __<=__ : Value × Value
end

spec Basics =
    free type PumpNumber ::= Pump1 | Pump2 | Pump3 | Pump4
    free type PumpState ::= Open | Closed
    free type PumpControllerState ::= Flow | NoFlow
    free type PhysicalUnit ::= Pump(PumpNumber) | PumpController(PumpNumber) | SteamOutput | WaterLevel
    free type Mode ::= Initialization | Normal | Degraded | Rescue | EmergencyStop
end

spec Messages_Sent =
    Basics
then free type S_Message ::= MODE(Mode)
                            | PROGRAM_READY
                            | VALVE
                            | OPEN_PUMP(PumpNumber)
                            | CLOSE_PUMP(PumpNumber)
                            | FAILURE_DETECTION(PhysicalUnit)
                            | REPAIRED_ACKNOWLEDGEMENT(PhysicalUnit)
end

spec Messages_Received =
    Basics and  Value
then free type
    R_Message ::= STOP
                | STEAM_BOILER_WAITING
```

```
                  | PHYSICAL_UNITS_READY
                  | PUMP_STATE(PumpNumber; PumpState)
                  | PUMP_CONTROLLER_STATE(PumpNumber;
                                                  PumpControllerState)
                  | LEVEL(Value)
                  | STEAM(Value)
                  | REPAIRED(PhysicalUnit)
                  | FAILURE_ACKNOWLEDGEMENT(PhysicalUnit)
                  | junk
end

spec SBCS_CONSTANTS =
     VALUE
then ops   C, M1, M2, N1, N2, W, U1, U2, P : Value;
           dt : Value
           %% Time duration between two cycles (5 sec.)
           %% These constants must verify some obvious properties:
     • 0 < M1
     • M1 < N1
     • N1 < N2
     • N2 < M2
     • M2 < C
     • 0 < W
     • 0 < U1
     • 0 < U2
     • 0 < P
end

spec PRELIMINARY =
     SET[MESSAGES_RECEIVED fit Elem ↦ R_Message]
and  SET[MESSAGES_SENT fit Elem ↦ S_Message]
and  SBCS_CONSTANTS
end

spec SBCS_STATE_1 =
     PRELIMINARY
then sort  State
     ops   mode : State → Mode;
           numSTOP : State → Nat
end

spec MODE_EVOLUTION
     [preds Transmission_OK : State × Set[R_Message];
             PU_OK : State × Set[R_Message] × PhysicalUnit;
             DangerousWaterLevel : State × Set[R_Message]]
     given SBCS_STATE_1 =
     local

     %% Auxiliary predicates to structure the specification of next_mode.
     preds Everything_OK, AskedToStop, SystemStillControllable,
           Emergency : State × Set[R_Message]
     ∀ s : State; msgs : Set[R_Message]
```

```
    • Everything_OK(s, msgs) ⇔ Transmission_OK(s, msgs) ∧ ∀ pu : PhysicalUnit • PU_OK(s, msgs, pu)
    • AskedToStop(s, msgs) ⇔ numSTOP(s) = 2 ∧ STOP eps msgs
    • SystemStillControllable(s, msgs)
      ⇔ PU_OK(s, msgs, SteamOutput)
         ∧ ∃ pn : PumpNumber
            • PU_OK(s, msgs, Pump(pn))
              ∧ PU_OK(s, msgs, PumpController(pn))
    • Emergency(s, msgs)
      ⇔ mode(s) = EmergencyStop ∨ AskedToStop(s, msgs)
         ∨ ¬ Transmission_OK(s, msgs)
         ∨ DangerousWaterLevel(s, msgs)
         ∨ (¬ PU_OK(s, msgs, WaterLevel)
            ∧ ¬ SystemStillControllable(s, msgs))
    within
    ops    next_mode : State × Set[R_Message] → Mode;
           next_numSTOP : State × Set[R_Message] → Nat
           %% Emergency stop mode:
    ∀ s : State; msgs : Set[R_Message]
    • Emergency(s, msgs) ⇒ next_mode(s, msgs) = EmergencyStop
    %% Normal mode:
    • ¬ Emergency(s, msgs) ∧ Everything_OK(s, msgs) ⇒ next_mode(s, msgs) = Normal
    %% Degraded mode:
    • ¬ Emergency(s, msgs) ∧ ¬ Everything_OK(s, msgs)
      ∧ PU_OK(s, msgs, WaterLevel)
      ∧ Transmission_OK(s, msgs)
      ⇒ next_mode(s, msgs) = Degraded
    %% Rescue mode:
    • ¬ Emergency(s, msgs) ∧ ¬ PU_OK(s, msgs, WaterLevel)
      ∧ SystemStillControllable(s, msgs)
      ∧ Transmission_OK(s, msgs)
      ⇒ next_mode(s, msgs) = Rescue
    %% next_numSTOP:
    • next_numSTOP(s, msgs) = numSTOP (s) + 1 when STOP eps msgs else 0
end

spec SBCS_STATE_2 =
    SBCS_STATE_1
then free type
    Status ::= OK | FailureWithoutAck | FailureWithAck
    op     status : State × PhysicalUnit → Status
end

spec STATUS_EVOLUTION
    [pred PU_OK : State × Set[R_Message] × PhysicalUnit]
    given SBCS_STATE_2 =
    op     next_status : State × Set[R_Message] × PhysicalUnit → Status
    ∀ s : State; msgs : Set[R_Message]; pu : PhysicalUnit
    • status(s, pu) = OK ∧ PU_OK(s, msgs, pu) ⇒ next_status(s, msgs, pu) = OK
    • status(s, pu) = OK ∧ ¬ PU_OK(s, msgs, pu) ⇒ next_status(s, msgs, pu) = FailureWithoutAck
    • status(s, pu) = FailureWithoutAck ∧ FAILURE_ACKNOWLEDGEMENT (pu) eps msgs
      ⇒ next_status(s, msgs, pu) = FailureWithAck
    • status(s, pu) = FailureWithoutAck ∧ ¬ FAILURE_ACKNOWLEDGEMENT (pu) eps msgs
```

```
        ⇒ next_status(s, msgs, pu) = FailureWithoutAck
    • status(s, pu) = FailureWithAck ∧ REPAIRED (pu) eps msgs ⇒ next_status(s, msgs, pu) = OK
    • status(s, pu) = FailureWithAck ∧ ¬ REPAIRED (pu) eps msgs ⇒ next_status(s, msgs, pu) = FailureWithAck
end

spec MESSAGE_TRANSMISSION_SYSTEM_FAILURE =
    SBCS_STATE_2
then local

    %% Static analysis:
    pred __is_static_OK : Set[R_Message]
    ∀ msgs : Set[R_Message]
    • msgs is_static_OK
      ⇔ ¬ junk eps msgs ∧ (∃! v : Value • LEVEL (v) eps msgs)
         ∧ (∃! v : Value • STEAM (v) eps msgs)
         ∧ (∀ pn : PumpNumber
            • ∃! ps : PumpState
              • PUMP_STATE (pn, ps) eps msgs)
         ∧ (∀ pn : PumpNumber
            • ∃! pcs : PumpControllerState
              • PUMP_CONTROLLER_STATE (pn, pcs) eps
                msgs)
         ∧ ∀ pu : PhysicalUnit
           • ¬ FAILURE_ACKNOWLEDGEMENT (pu) eps msgs
             ∧ REPAIRED (pu) eps msgs
    %% Dynamic analysis:
    pred __is_NOT_dynamic_OK_for__
          : Set[R_Message] × State
    ∀ s : State; msgs : Set[R_Message]
    • msgs is_NOT_dynamic_OK_for s
      ⇔ (¬ mode(s) = Initialization
          ∧ (STEAM_BOILER_WAITING eps msgs
              ∨ PHYSICAL_UNITS_READY eps msgs))
         ∨ (∃ pu : PhysicalUnit
            • FAILURE_ACKNOWLEDGEMENT (pu) eps msgs
              ∧ (status(s, pu) = OK
                  ∨ status(s, pu) = FailureWithAck))
         ∨ ∃ pu : PhysicalUnit
           • REPAIRED (pu) eps msgs
             ∧ (status(s, pu) = OK
                 ∨ status(s, pu) = FailureWithoutAck)
    within
    pred Transmission_OK : State × Set[R_Message]
    ∀ s : State; msgs : Set[R_Message]
    • Transmission_OK(s, msgs)
      ⇔ msgs is_static_OK
         ∧ ¬ msgs is_NOT_dynamic_OK_for s
end

spec SBCS_STATE_3 =
    SBCS_STATE_2
then free type
```

```
    ExtendedPumpState ::= sort PumpState | Unknown_PS
    op      PS_predicted : State × PumpNumber → ExtendedPumpState
            %{
            status(s,Pump(pn)) = OK <=>
            not (PS_predicted(s,pn) = Unknown_PS) }%
end

spec PUMP_FAILURE =
    SBCS_STATE_3
then pred  Pump_OK : State × Set[R_Message] × PumpNumber
    ∀ s : State; msgs : Set[R_Message]; pn : PumpNumber
    • Pump_OK(s, msgs, pn)
      ⇔ PS_predicted(s, pn) = Unknown_PS ∨ PUMP_STATE (pn, PS_predicted (s, pn) as PumpState) eps msgs
end

spec SBCS_STATE_4 =
    SBCS_STATE_3
then free type
    ExtendedPumpControllerState ::= sort PumpControllerState | SoonFlow | Unknown_PCS
    op      PCS_predicted : State × PumpNumber → ExtendedPumpControllerState
            %{
            status(s,PumpController(pn)) = OK =>
            not (PCS_predicted(s,pn) = Unknown_PCS) }%
end

spec PUMP_CONTROLLER_FAILURE =
    SBCS_STATE_4
then pred  Pump_Controller_OK : State × Set[R_Message] × PumpNumber
    ∀ s : State; msgs : Set[R_Message]; pn : PumpNumber
    • Pump_Controller_OK(s, msgs, pn)
      ⇔ PCS_predicted(s, pn) = Unknown_PCS
         ∨ PCS_predicted(s, pn) = SoonFlow
         ∨ PUMP_CONTROLLER_STATE
           (pn, PCS_predicted (s, pn) as PumpControllerState)
           eps msgs
end

spec SBCS_STATE_5 =
    SBCS_STATE_4
then free type Valpair ::= pair(low : Value; high : Value)
    ops     steam_predicted, level_predicted : State → Valpair
            %{
            low(steam_predicted(s)) is the minimal steam output predicted,
            high(steam_predicted(s)) is the maximal steam output predicted,
            and similarly for level_predicted. }%
end

spec STEAM_FAILURE =
    SBCS_STATE_5
then pred  Steam_OK : State × Set[R_Message]
    ∀ s : State; msgs : Set[R_Message]
    • Steam_OK(s, msgs)
```

```
          ⇔ ∀ v : Value
             • STEAM (v) eps msgs
               ⇒ low (steam_predicted(s)) <= v
                  ∧ v <= high (steam_predicted(s))
end

spec LEVEL_FAILURE =
     SBCS_STATE_5
then pred Level_OK : State × Set[R_Message]
     ∀ s : State; msgs : Set[R_Message]
     • Level_OK(s, msgs)
       ⇔ ∀ v : Value
          • LEVEL (v) eps msgs
            ⇒ low (level_predicted(s)) <= v
               ∧ v <= high (level_predicted(s))
end

spec FAILURE_DETECTION =
     {MESSAGE_TRANSMISSION_SYSTEM_FAILURE
     and PUMP_FAILURE
     and PUMP_CONTROLLER_FAILURE
     and STEAM_FAILURE
     and LEVEL_FAILURE
     then pred PU_OK : State × Set[R_Message] × PhysicalUnit
          ∀ s : State; msgs : Set[R_Message]; pn : PumpNumber
          • PU_OK(s, msgs, Pump(pn))
            ⇔ Pump_OK(s, msgs, pn)
          • PU_OK(s, msgs, PumpController(pn))
            ⇔ Pump_Controller_OK(s, msgs, pn)
          • PU_OK(s, msgs, SteamOutput)
            ⇔ Steam_OK(s, msgs)
          • PU_OK(s, msgs, WaterLevel) ⇔ Level_OK(s, msgs)
     }
     hide preds Pump_OK, Pump_Controller_OK, Steam_OK,
                Level_OK
end

spec STEAM_AND_LEVEL_PREDICTION =
     FAILURE_DETECTION
and  SET[sort PumpNumber fit Elem ↦ PumpNumber]
then local
     ops  received_steam : State × Set[R_Message] → Value;
          adjusted_steam : State × Set[R_Message] → Valpair;
          received_level : State × Set[R_Message] → Value;
          adjusted_level : State × Set[R_Message] → Valpair;
          broken_pumps : State × Set[R_Message] → Set[PumpNumber];
          reliable_pumps : State × Set[R_Message] × PumpState → Set[PumpNumber]
          %% Axioms for STEAM:
     ∀ s : State; msgs : Set[R_Message]; pn : PumpNumber;
     ps : PumpState
     • Transmission_OK(s, msgs) ⇒ STEAM (received_steam(s, msgs)) eps msgs
     • adjusted_steam(s, msgs) = pair(received_steam(s, msgs), received_steam(s, msgs))
```

```
                  when Transmission_OK(s, msgs) ∧ PU_OK(s, msgs, SteamOutput)
                  else steam_predicted(s)
%% Axioms for LEVEL:
• Transmission_OK(s, msgs)
  ⇒ LEVEL (received_level(s, msgs)) eps msgs
• adjusted_level(s, msgs)
  = pair(received_level(s, msgs), received_level(s, msgs))
     when Transmission_OK(s, msgs)
            ∧ PU_OK(s, msgs, WaterLevel)
     else level_predicted(s)
%% Axioms for auxiliary pumps operations:
• pn eps broken_pumps (s, msgs)
  ⇔ ¬ PU_OK(s, msgs, Pump(pn))
      ∧ PU_OK(s, msgs, PumpController(pn))
• pn eps reliable_pumps (s, msgs, ps)
  ⇔ ¬ pn eps broken_pumps (s, msgs)
      ∧ PUMP_STATE (pn, ps) eps msgs
within
ops    next_steam_predicted
         : State × Set[R_Message] → Valpair;
         chosen_pumps : State × Set[R_Message] × PumpState →
                          Set[PumpNumber];
         minimal_pumped_water, maximal_pumped_water : State × Set[R_Message] → Value;
         next_level_predicted : State × Set[R_Message] → Valpair
pred  DangerousWaterLevel : State × Set[R_Message]
         %% Axioms for STEAM:
∀ s : State; msgs : Set[R_Message]; pn : PumpNumber
• low(next_steam_predicted(s, msgs))
  = max(0, low (adjusted_steam(s, msgs)) − (U2 ∗ dt))
• high(next_steam_predicted(s, msgs))
  = min(W, high (adjusted_steam(s, msgs)) + (U1 ∗ dt))
%% Axioms for PUMPS:
• pn eps chosen_pumps (s, msgs, Open)
  ⇒ pn eps reliable_pumps (s, msgs, Closed)
• pn eps chosen_pumps (s, msgs, Closed)
  ⇒ pn eps reliable_pumps (s, msgs, Open)
• minimal_pumped_water(s, msgs)
  = (dt ∗ P) ∗ #
     (reliable_pumps (s, msgs, Open) − chosen_pumps
      (s, msgs, Closed))
• maximal_pumped_water(s, msgs)
  = (dt ∗ P) ∗ #
     (((reliable_pumps (s, msgs, Open) union chosen_pumps
         (s, msgs, Open))
       union broken_pumps (s, msgs))
      − chosen_pumps (s, msgs, Closed))
%% Axioms for LEVEL:
• low(next_level_predicted(s, msgs))
  = max
     (0,
      (low (adjusted_level(s, msgs)) +
       minimal_pumped_water (s, msgs))
```

```
           −
           ((dt square ∗ U1 half) +
            (dt ∗ high (adjusted_steam(s, msgs)))))
     • high(next_level_predicted(s, msgs))
         = min
           (C,
            (high (adjusted_level(s, msgs)) +
             maximal_pumped_water (s, msgs))
            −
           ((dt square ∗ U2 half) +
            (dt ∗ low (adjusted_steam(s, msgs)))))
     • DangerousWaterLevel(s, msgs)
         ⇔ low (next_level_predicted(s, msgs)) <= M1
             ∨ M2 <= high (next_level_predicted(s, msgs))
     hide ops minimal_pumped_water, maximal_pumped_water
end

spec PUMP_STATE_PREDICTION =
     STATUS_EVOLUTION[FAILURE_DETECTION]
and  STEAM_AND_LEVEL_PREDICTION
then op      next_PS_predicted
               : State × Set[R_Message] × PumpNumber →
                 ExtendedPumpState
     ∀ s : State; msgs : Set[R_Message]; pn : PumpNumber
     • next_PS_predicted(s, msgs, pn)
         = Unknown_PS
           when ¬ next_status(s, msgs, Pump(pn)) = OK
           else Open
                when (PUMP_STATE (pn, Open) eps msgs
                        ∧ ¬ pn eps chosen_pumps (s, msgs, Closed))
                       ∨ pn eps chosen_pumps (s, msgs, Open)
                else Closed
end

spec PUMP_CONTROLLER_STATE_PREDICTION =
     STATUS_EVOLUTION[FAILURE_DETECTION]
and  STEAM_AND_LEVEL_PREDICTION
then op      next_PCS_predicted
               : State × Set[R_Message] × PumpNumber →
                 ExtendedPumpControllerState
     ∀ s : State; msgs : Set[R_Message]; pn : PumpNumber
     • next_PCS_predicted(s, msgs, pn)
         = Unknown_PCS
           when ¬ next_status(s, msgs, PumpController(pn)) = OK
                 ∧ next_status(s, msgs, Pump(pn)) = OK
           else Flow
                when (PUMP_CONTROLLER_STATE (pn, Flow) eps
                        msgs
                        ∨ (PUMP_CONTROLLER_STATE
                            (pn, NoFlow) eps msgs
                            ∧ PCS_predicted(s, pn) = SoonFlow))
                       ∧ ¬ pn eps chosen_pumps (s, msgs, Closed)
            else NoFlow
                when pn eps chosen_pumps (s, msgs, Closed)
                     ∨ (PUMP_CONTROLLER_STATE
                        (pn, NoFlow) eps msgs
                        ∧ ¬ PCS_predicted(s, pn) = SoonFlow
                        ∧ ¬ pn eps chosen_pumps
                            (s, msgs, Open))
                else SoonFlow
end

spec PU_Prediction =
     Pump_State_Prediction and  Pump_Controller_State_Prediction
end

spec Sbcs_Analysis =
     Mode_Evolution[PU_Prediction]
then local
     ops    PumpMessages, FailureDetectionMessages
            : State × Set[R_Message] → Set[S_Message];
            RepairedAcknowledgementMessages
            : Set[R_Message] → Set[S_Message]
     ∀ s : State; msgs : Set[R_Message]; Smsg : S_Message
     • Smsg eps PumpMessages (s, msgs)
       ⇔ ∃ pn : PumpNumber
          • (pn eps chosen_pumps (s, msgs, Open)
             ∧ Smsg = OPEN_PUMP(pn))
            ∨ (pn eps chosen_pumps (s, msgs, Closed)
               ∧ Smsg = CLOSE_PUMP(pn))
     • Smsg eps FailureDetectionMessages (s, msgs)
       ⇔ ∃ pu : PhysicalUnit
          • Smsg = FAILURE_DETECTION(pu)
            ∧ next_status(s, msgs, pu) = FailureWithoutAck
     • Smsg eps RepairedAcknowledgementMessages (msgs)
       ⇔ ∃ pu : PhysicalUnit
          • Smsg = REPAIRED_ACKNOWLEDGEMENT(pu)
            ∧ next_status(s, msgs, pu) = FailureWithAck
     within
     op     messages_to_send
            : State × Set[R_Message] → Set[S_Message]
     ∀ s : State; msgs : Set[R_Message]
     • messages_to_send(s, msgs)
       = ((PumpMessages (s, msgs) union
           FailureDetectionMessages (s, msgs))
          union RepairedAcknowledgementMessages (msgs))
         + MODE (next_mode(s, msgs))
end

spec Sbcs_State =
     Preliminary
then sort   State
     free type
     Status ::= OK | FailureWithoutAck | FailureWithAck
     free type
     ExtendedPumpState ::= sort PumpState | Unknown_PS
     free type
     ExtendedPumpControllerState
     ::= sort PumpControllerState | SoonFlow | Unknown_PCS
     free type Valpair ::= pair(low : Value; high : Value)
     ops   mode : State → Mode;
           numSTOP : State → Nat;
           status : State × PhysicalUnit → Status;
           PS_predicted
           : State × PumpNumber → ExtendedPumpState;
           PCS_predicted
           : State × PumpNumber →
             ExtendedPumpControllerState;
           steam_predicted, level_predicted : State → Valpair
end

spec STEAM_BOILER_CONTROL_SYSTEM =
     SBCS_ANALYSIS
then op    init : State
     pred  is_step
           : State × Set[R_Message] × Set[S_Message] × State
           %% Specification of the initial state init:
     • mode(init) = Normal ∨ mode(init) = Degraded
     %% Specification of is_step:
     ∀ s, s' : State; msgs : Set[R_Message];
     Smsg : Set[S_Message]
     • is_step(s, msgs, Smsg, s')
       ⇔ mode(s') = next_mode(s, msgs)
          ∧ numSTOP(s') = next_numSTOP(s, msgs)
          ∧ (∀ pu : PhysicalUnit
               • status(s', pu) = next_status(s, msgs, pu))
          ∧ (∀ pn : PumpNumber
               • PS_predicted(s', pn)
                 = next_PS_predicted(s, msgs, pn)
                 ∧ PCS_predicted(s', pn)
                   = next_PCS_predicted(s, msgs, pn))
          ∧ steam_predicted(s') = next_steam_predicted(s, msgs)
          ∧ level_predicted(s') = next_level_predicted(s, msgs)
          ∧ Smsg = messages_to_send(s, msgs)
then
     %% Specification of the reachable states:
     free
     {pred  reach : State
     ∀ s, s' : State; msgs : Set[R_Message];
     Smsg : Set[S_Message]
     • reach(init)
     • reach(s) ∧ is_step(s, msgs, Smsg, s') ⇒ reach(s')
     }
end

arch spec 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

spec Elem =
    sort   Elem
end

arch spec Arch_Preliminary =
    units SET : Elem × Nat → Set[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

spec Sbcs_State_Impl =
    Preliminary
then free type Status ::= OK | FailureWithoutAck | FailureWithAck
    free type ExtendedPumpState ::= sort PumpState | Unknown_PS
    free type ExtendedPumpControllerState ::= sort PumpControllerState | SoonFlow | Unknown_PCS
    free type Valpair ::= pair(low : Value; high : Value)
then TotalMap[Basics fit S ↦ PhysicalUnit][sort Status]
and  TotalMap
    [Basics fit S ↦ PumpNumber][sort ExtendedPumpState]
and  TotalMap
    [Basics fit S ↦ PumpNumber]
    [sort ExtendedPumpControllerState]
then free type
    State ::= mk_state(mode : Mode;
                       numSTOP : Nat;
                       status : TotalMap[PhysicalUnit,Status];
                       PS_predicted : TotalMap[PumpNumber,ExtendedPumpState];
                       PCS_predicted : TotalMap [PumpNumber,ExtendedPumpControllerState];
                       steam_predicted, level_predicted : Valpair)
    ops    status(s : State; pu : PhysicalUnit) : Status = lookup(pu, status(s));
           PS_predicted (s : State; pn : PumpNumber) : ExtendedPumpState = lookup(pn, PS_predicted(s));
           PCS_predicted(s : State; pn : PumpNumber) : ExtendedPumpControllerState = lookup(pn, PCS_predicted(s))
end

unit spec Unit_Sbcs_State =
    Preliminary → Sbcs_State_Impl
end

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 lambda S : Sbcs_State • MTS [ME [PR [FD [S]]]]
end

arch spec 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 lambda S : Sbcs_State
          • PU [MTSF [S]] [PF [S]] [PCF [S]] [SF [S]] [LF [S]]
            hide Pump_OK, Pump_Controller_OK, Steam_OK, Level_OK
end

arch spec 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 lambda FD : Failure_Detection
          • local SEFD = SE [FD]; SLPFD = SLP [FD] within
            {PP [SEFD] [SLPFD] and PCP [SEFD] [SLPFD]}
end


%% We may record this initial refinement now:
spec Sbcs_System_Impl =
     Steam_Boiler_Control_System
end

unit spec Unit_Sbcs_System =
     Sbcs_Analysis → Sbcs_System_Impl
end

unit spec SBCS_Open =
     Value → Steam_Boiler_Control_System
end

refinement Ref_Sbcs =
          SBCS_Open refined to arch spec Arch_Sbcs
end

unit spec StateAbstr = Preliminary → Sbcs_State
end

refinement StateRef =
          StateAbstr refined to Unit_Sbcs_State
end
```

```
refinement REF_SBCS' =
        REF_SBCS
        then {P to arch spec Arch_Preliminary, S to STATEREF, A to arch spec Arch_Analysis}
end

refinement REF_SBCS'' =
        REF_SBCS'
        then {A to
            {FD to arch spec Arch_Failure_Detection, PR to arch spec Arch_Prediction }
        }
end
```