Specification refinements: calculi, tools, and applications

Citation for published version:

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

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
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

1 Introduction 2

2 Refinements in CASL 3

2.1 A Brief Summary of Institution-Independent Specifications in CASL 3

2.2 Architectural Specifications: Motivation 5

2.3 CASL Architectural Specifications 5

2.4 Refinements 8

2.5 Examples and Methodology 13

3 Foundations 15

4 Semantics of CASL Refinements 19

4.1 Semantics of CASL Structured Specifications 19

4.2 Semantics of Unit Specifications 19

4.3 Semantics of Architectural Specifications 20

4.4 Static Semantics of Refinements 22

4.5 Model Semantics of Refinements 25

5 Calculi for Refinements 28

5.1 Proof Calculus for Architectural Specifications 29

5.2 Unit Imports in CASL 35

5.3 Proof Calculus for Refinements 36

5.4 Completeness of the Proof Calculus 40

5.5 Checking Consistency of Refinements 43

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)

Preprint submitted to Elsevier April 18, 2017
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.

\[
SP \rightsquigarrow \begin{cases}
SP_1 \rightsquigarrow P_1 \\
\vdots \\
SP_n \rightsquigarrow \begin{cases}
SP_{n1} \rightsquigarrow \ldots \rightsquigarrow P_{n11} \\
\ldots \\
SP_{nm} \rightsquigarrow P_{nm}
\end{cases}
\end{cases}
\]

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 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 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.

3
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 \( \varphi \) is a \( \Sigma_1 \)-sentence and \( \sigma : \Sigma_1 \to \Sigma_2 \) is a signature morphism, then \( \sigma(\varphi) \) is a \( \Sigma_2 \)-sentence. Moreover, if \( M \) is a \( \Sigma_2 \)-model, then it can be reduced to a \( \Sigma_1 \)-model \( M\sigma \), called the \( \sigma \)-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 \( (\Sigma, E) \) essentially consists of a signature \( \Sigma \) 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 \textbf{and}, resulting in a conjunctive combination of the constraints expressed by the individual specifications. A specification \( \text{SPEC} \) can be renamed along a signature morphism \( \sigma \) using \text{SPEC \ with \ \( \sigma \)}, resulting in a specification over the target signature of the signature morphism. Finally, a specification \( \text{SPEC} \) can be restricted to an export interface using \text{SPEC \ hide \ \( \sigma \)}, where \( \sigma \) is the inclusion of the export interface signature into that of \( \text{SPEC} \).

A semantics of structured specifications will be given in Sect. 4. For now, let us just note that a structured specification \( \text{SP} \) denotes a signature (written \( \text{Sig}(\text{SP}) \)) 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 \( \text{SP}_1, \text{SP}_2 \) are structured specifications and \( \sigma : \text{Sig}(\text{SP}_1) \to \text{Sig}(\text{SP}_2) \) is a signature morphism, we say that \( \text{SP}_1 \) refines along \( \sigma \) to \( \text{SP}_2 \), denoted \( \text{SP}_1 \overset{\sigma}{\to} \text{SP}_2 \), if for each model of \( \text{SP}_2 \), its reduct along \( \sigma \) is a model of \( \text{SP}_1 \). If \( \sigma \) is the identity, it can be omitted. CASL already allows such refinements to be postulated using \textit{views}: if \( \text{SP}_1 \) and \( \text{SP}_2 \) are structured specifications, then \textit{view} \( \text{V : SP}_1 \text{ to SP}_2 = \sigma \) requires each model of \( \text{SP}_2 \) reduced along \( \sigma \) to be a model of \( \text{SP}_1 \). If this holds, then \( \sigma : \text{SP}_1 \to \text{SP}_2 \) is called a \textit{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] uses a similar syntax to the one in Figs. 1 and 2.

---

1See \url{http://www.omg.org/spec/DOL/} and \url{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),
- \( UN_1, \ldots, UN_n \) are units (i.e. named models) with their specifications \( USP_1, \ldots, USP_n \),
- \( \kappa \) is the linking procedure involving the units, i.e. a function taking models \( A_i \) of \( USP_i \), \( i = 1, \ldots, n \) to a model \( \kappa(A_1, \ldots, A_n) \) over the signature of \( SP \), and
- the refinement relation is denoted \( \leadsto \), expressing that the model given by \( \kappa(A_1, \ldots, A_n) \) is indeed a model of \( SP \).

Intuitively, the refinement is correct if for any models \( A_i \) of \( USP_i \), \( i = 1, \ldots, n \), \( \kappa(A_1, \ldots, A_n) \) 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 \( A_1, \ldots, A'_j, \ldots, A_n \) as prescribed by \( \kappa \) to get a realization of \( SP \), without having to modify the models \( A_1, \ldots, A_{j-1}, A_{j+1}, \ldots, A_n \) 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 Fig. 3 is written:

```
arch spec ASP_NAME =
  units
  UN_1 : USP_1;
  \ldots
  UN_n : USP_n;
result UE
```

where \( UE \) is a unit expression describing the linking procedure \( \kappa \) and possibly involving the units \( UN_1, \ldots, UN_n \). 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
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:

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:

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 `float`, should be provided from outside the architectural specification `TYPES`, we could make use of a unit expression: `R = \text{lambda } X:\text{ float} \cdot N \text{ and } S \text{ and } B \text{ 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

\begin{figure}[h]
\centering
\begin{verbatim}
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 | A 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’
\end{verbatim}
\caption{Syntax of CASL architectural specifications.}
\end{figure}
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: 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:

<table>
<thead>
<tr>
<th>Component diagrams</th>
<th>Architectural specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>component</td>
<td>unit</td>
</tr>
<tr>
<td>required interface</td>
<td>argument specification</td>
</tr>
<tr>
<td>provided interface</td>
<td>result specification</td>
</tr>
<tr>
<td>delegation connector to required interface</td>
<td>lambda unit expression</td>
</tr>
<tr>
<td>delegation connector to provided interface</td>
<td>result unit expression</td>
</tr>
<tr>
<td>assembly connector</td>
<td>unit term in an application</td>
</tr>
</tbody>
</table>

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

2The complete specification of the SBCS example can be found under https://ontohub.org/hets-lib/UserManual/Sbcs.casl
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.\footnote{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.}

\begin{itemize}
  \item [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 $\sigma$ 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:
  \[
  \begin{array}{c}
  SP \\
  \sigma \\
  \rightarrow \\
  SP'
  \end{array}
  \]

  To save space, we sometimes write refinements horizontally instead of vertically. The arrow indicates which direction is “down”.
\end{itemize}
B Branching refinements introduce architectural decompositions. An architectural specification $ASP_{\text{NAME}}$ with $n$ units $UN_1, \ldots, UN_n$ 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 $ASP_{\text{NAME}}$ is

$$
\text{ASP}_{\text{NAME}} -> UN_1 -> \cdots -> UN_n
$$

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 $ASP_{\text{NAME}}$ introduces a new root node, labeled with $ASP_{\text{NAME}}$, 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 $UN_1, \ldots, UN_k$. 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 $\sigma$ to $USP'$, where $USP$ and $USP'$ are generic unit specifications of the form $SP_1 \times \ldots \times SP_n \mapsto SP$ and $SP_1 \times \ldots \times SP_n \mapsto SP'$ respectively and $\sigma : \text{Sig}(SP) \rightarrow \text{Sig}(SP')$. When $\sigma$ is omitted, $\sigma$ 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 $U' \in \text{Unit}(USP')$ as well; in this case, the notation denotes the unit function obtained by reducing the result w.r.t. $\sigma$ 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 $\sigma$ to $USP'$ as

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

and denote this by $USP \xrightarrow{\sigma} USP'$. If $USP$ and $USP'$ are non-generic, we can express the refinement $USP$ refined via $\sigma$ to $USP'$ equivalently as view $V : USP$ to $USP' = \sigma$. 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 \( USP \xrightarrow{\sigma} USP' \xrightarrow{\tau} USP'' \). We write this \( USP \) refined via \( \sigma \) to \( USP' \) refined via \( \tau \) to \( USP'' \), or using named refinements as one of the three equivalent constructions \( C_1 \), \( C_2 \) or \( C_3 \) below:

- \( \text{refinement } R = USP \) refined via \( \sigma \) to \( USP' \)
- \( \text{refinement } R' = USP' \) refined via \( \tau \) to \( USP'' \)
- \( \text{refinement } C_1 = USP \) refined via \( \sigma \) to \( R' \)
- \( \text{refinement } C_2 = USP \) refined via \( \sigma \) to \( USP' \) then \( R' \)
- \( \text{refinement } C_3 = R \) then \( R' \)

where the last two alternatives make use of an operation of composition of refinements, using the keyword \( \text{then} \). 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:

```plaintext
spec Monoid =
  sort Elem
  ops 0 : Elem;
  _+_ : Elem × Elem → Elem, assoc, unit 0
```

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

```plaintext
spec NatWithSuc =
  free type Nat ::= 0 | suc(Nat) \% \% CASL shorthand for Peano axioms
  op _+_ : Nat × Nat → 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:

```plaintext
refinement R1 = Monoid refined via Elem ⇔ Nat to Nat
```

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.

```plaintext
spec NatBin =
  generated type Bin ::= 0 | 1 | _0(Bin) | _1(Bin)
  ops _+_ : Bin × Bin → Bin
  \forall x, y : Bin
  \bullet 0 + 0 = 0
  \bullet 0 + 1 = 1
  \bullet 1 + 0 = 1
  \bullet 1 + 1 = x = y
  \bullet x + 0 = y 1
  \bullet x + 1 = y 1
  \bullet 0 + 0 = 0
  \bullet 0 + 1 = 1
  \bullet 0 + 0 = 1
```

We obtain a new refinement:

```plaintext
refinement R2 = Nat refined via Nat ⇔ Bin to NatBin
```

which can be composed with the first one:

```plaintext
refinement R3 = R1 then R2
```

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

\[\]
refinement $R = USP$ refined via $\sigma$ 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 $\sigma$ 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 $\sigma$.

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 $AdditionFirst =$ {
    units $N : Nat$;
    $F : Nat \rightarrow NatWithSuc$
    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 $AdditionFirst$

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 $AdditionFirstRef =$ {
    units $N : Nat$ refined via $Nat \rightarrow Bin$ to $NatBin$ ;
    $F : Nat \rightarrow NatWithSuc$
    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 $A_1$, \ldots, $A_N$ and their specifications have been refined as captured by the refinements $R_1$, \ldots, $R_N$, 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{I}} where \mathcal{I} is a set of indices and for each i \in \mathcal{I}, UN_i stands for a unit name and SPR_i for a refinement. This expresses that for each i \in \mathcal{I}, 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_1, for an index i \in \{1, ..., K\}, 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 R_{COMP} = [N to R2]
refinement R5 = R4 then R_{COMP}

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.

12
The complete syntax for CASL refinements is presented in Fig. 6, where $A, A_1, \ldots, A_n$ stand for unit names and $\sigma$ 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

$$\text{refinement } \text{Ref}_\text{SBCS} = \text{SBCS}_\text{OPEN} \text{ refined to } \text{arch spec } \text{ARCH}_\text{SBCS}$$

We proceed with refining individual units. The specifications of $C$ and $S$ in $\text{ARCH}_\text{SBCS}$ above do not require further architectural decomposition. The specification of $S$, recorded in the unit specification $\text{STATE}_\text{ABSTR}$, can be refined by providing an implementation of states as a record of all observable values. This is done in $\text{SBCS}_\text{STATE}_\text{Impl}$, assuming an implementation of $\text{PRELIMINARY}$; we record this development in the unit specification $\text{UNIT}_\text{SBCS}_\text{STATE}$.

The refinement of $S$ is then written in $\text{STATE}_\text{Ref}$.

```
unit spec \text{STATE}_\text{ABSTR} =
\text{PRELIMINARY} \rightarrow \text{SBCS}_\text{STATE}

unit spec \text{UNIT}_\text{SBCS}_\text{STATE} =
\text{PRELIMINARY} \rightarrow \text{SBCS}_\text{STATE}_\text{Impl}

refinement \text{STATE}_\text{Ref} =
\text{STATE}_\text{ABSTR} \text{ refined to } \text{UNIT}_\text{SBCS}_\text{STATE}
```

For the units $A$ and $P$, we proceed with designing their architecture. This is recorded in the architectural specifications $\text{ARCH}_\text{Analysis}$ and $\text{ARCH}_\text{PRELIMINARY}$.

```
arch spec \text{ARCH}_\text{Analysis} =
units
FD : \text{SBCS}_\text{STATE} \rightarrow \text{FAILURE} \text{\_DETECTION};
PR : \text{FAILURE} \text{\_DETECTION} \rightarrow \text{PU} \text{\_PREDICTION};
ME : \text{PU} \text{\_PREDICTION} \rightarrow \text{MODE} \text{\_EVOLUTION}[\text{PU} \text{\_PREDICTION}];
MTS : \text{MODE} \text{\_EVOLUTION}[\text{PU} \text{\_PREDICTION}] \rightarrow \text{SBCS}_\text{ANALYSIS}
result \lambda S : \text{SBCS}_\text{STATE} \bullet \text{MTS} [\text{ME} [\text{PR} [\text{FD} [S]]]]
```

\[\text{See } \text{https://spechub.org/casl/hets-lib/UserManual/SBCS.casl} \text{ for } \text{SBCS}_\text{STATE}_\text{Impl} \text{ and other specifications in the steam boiler system example that are referred to here.}\]

\[\text{This can be equivalently written refinement } \text{STATE}_\text{Ref} = \text{PRELIMINARY} \rightarrow \text{SBCS}_\text{STATE}_\text{PRELIMINARY} \rightarrow \text{SBCS}_\text{STATE}_\text{Impl}.\]

\[\text{The specification MODE} \text{\_EVOLUTION is generic, and is instantiated with the actual parameter PU} \text{\_PREDICTION. Although the syntax is similar, the concepts of (applications of) generic units and generic specifications are different, see } 4.\]

\[\text{Since two of the units of ARCH\_PRELIMINARY have imports, its refinement tree is actually slightly different than the one we show here. This will be clarified in Sect. } 5.2\text{ and the correct refinement (sub)tree is presented in Fig. } 20.\]
arch spec \texttt{ARCH\_PRELIMINARY} = \\
units \\
\text{SET : } \text{sort} \ E \times \text{Nat} \rightarrow \text{SET}[\text{sort} \ E]; \\
\text{B : Basics}; \\
\text{MS : Messages\_Sent given } B; \\
\text{MR : Value } \rightarrow \text{Messages\_Received given } B; \\
\text{CST : Value } \rightarrow \text{Sbcs\_CONSTANTS} \\
\text{result } \lambda V : \text{Value} \\
\text{• SET } [\text{MS fit} \ E \rightarrow S \_Message] \ [V] \\
\text{and SET } [\text{MR } [V] \text{ fit } \ E \rightarrow R \_Message] \ [V] \\
\text{and CST } [V] \\
\text{We can now record the component refinement:} \\
\text{refinement } \text{REF\_SBCS’ = REF\_SBCS then} \\
\{ \text{P to arch spec } \text{ARCH\_PRELIMINARY, S to } \text{STATE\_Ref,} \\
\text{A to arch spec } \text{ARCH\_ANALYSIS} \} \\
\text{The refinement tree of the component refinement following then is:} \\
\text{The refinement tree of } \text{REF\_SBCS’ consists of the first five levels of the tree in Fig. 20 in Sect. 6.2} \\
\text{Moreover, the components } \text{FD and PR of } \text{ARCH\_ANALYSIS are further refined} \text{ (the architectural specifications } \text{ARCH\_Failure\_Detection and ARCH\_Prediction are omitted here)} \\
\text{refinement } \text{REF\_SBCS” = } \\
\text{REF\_SBCS’} \\
\text{then } \{ \text{A to} \\
\{ \text{FD to arch spec } \text{ARCH\_Failure\_Detection,} \\
\text{PR to arch spec } \text{ARCH\_Prediction } \} \} \\
\text{The refinement tree of } \text{REF\_SBCS” is shown in Fig. 20 For the component refinement following then we have the following tree:} \\
\text{FD} \\
\text{PR} \\
\text{refinement } \text{REF\_SBCS’} = \text{REF\_SBCS’} \\
\text{then } \{ \text{A to} \\
\{ \text{FD to arch spec } \text{ARCH\_Failure\_Detection,} \\
\text{PR to arch spec } \text{ARCH\_Prediction } \} \} \\
\text{The refinement tree of } \text{REF\_SBCS” is shown in Fig. 20 For the component refinement following then we have the following tree:} \\
\text{FD} \\
\text{PR}
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 \( R = \text{arch spec ASP_NAME then USP} \)

The static and model semantics rules for refinements that we will introduce in Sect. 3 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 \( I = (\text{Sig, Sen, Mod}, \models) \) consists of:

- a category \( \text{Sig} \) of signatures,
- a functor \( \text{Sen} : \text{Sig} \to \text{Set} \), giving for each signature \( \Sigma \) a set of sentences \( \text{Sen}(\Sigma) \), and for each signature morphism \( \sigma : \Sigma \to \Sigma' \), a sentence translation map \( \text{Sen}(\sigma) : \text{Sen}(\Sigma) \to \text{Sen}(\Sigma') \), where we may write \( \text{Sen}(\sigma)(e) \) as \( \sigma(e) \),
- a functor \( \text{Mod} : \text{Sig} \to \text{Cat} \), giving for each signature \( \Sigma \) a category of models \( \text{Mod}(\Sigma) \) and for each signature morphism \( \sigma : \Sigma \to \Sigma' \), a reduct functor \( \text{Mod}(\sigma) : \text{Mod}(\Sigma') \to \text{Mod}(\Sigma) \), where we may write \( \text{Mod}(\sigma)(M') \) as \( \text{Mod}(\sigma|M') \);
- a binary relation \( \models \subseteq [\text{Mod}(\Sigma)] \times \text{Sen}(\Sigma) \), for each signature \( \Sigma \), called the satisfaction relation such that the following satisfaction condition holds:

\[
M'|\rho \models_{\Sigma} e \iff M' \models_{\Sigma'} \sigma(e)
\]

for each signature morphism \( \sigma : \Sigma \to \Sigma' \), \( \Sigma \)-sentence \( e \) and \( \Sigma' \)-model \( M' \). \( \square \)

Example 3.1. [15] First-order Logic. In the institution \( \text{FOL}^n \) 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. \( \square \)

Example 3.2. [15] Multi-sorted first-order logic. In the institution \( \text{MSFOL}^n \) 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 \( f : s_1 \times s_2 \times \ldots \times s_n \to s \)), and for a predicate symbols \( p \) with \( n \) argument we give the sorts \( s_1, \ldots, s_n \) of the arguments.

---

\( ^9 \) As usual, \( \text{Set} \) is the category having all small sets as objects and functions as arrows and \( \text{Cat} \) 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 \( \Sigma \), then its image under \( \sigma \), 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 \( t \) 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 \( \Phi, \Phi_1, \Phi_2 \) are formulas, then \( \neg \Phi, \Phi_1 \land \Phi_2, \Phi_1 \lor \Phi_2, \Phi_1 \implies \Phi_2, \Phi_1 \iff \Phi_2, \forall X \bullet \Phi \) 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. \( \square \)

**Example 3.3.** [6] Multi-sorted first order logic with sort generation constraints. This logic extends MSFOL\(^*\) with a new kind of sentences, sort generation constraints. A sort generation constraint over a signature \( \Sigma \) is a triple \((S', F', \sigma')\) such that \( \sigma' : \Sigma' \to \Sigma \) and \( S' \) and \( F' \) are sets of, respectively, sort and function symbols of \( \Sigma' \). Sort generation constraints are translated along a signature morphism \( \varphi : \Sigma \to \Sigma'' \) by composing the morphism \( \sigma' \) in their third component with \( \varphi \). A sort generation constraint \((S', F', \sigma')\) holds in a model \( M \) if the carriers of \( M|_{S'} \) of the sorts in \( S' \) are generated by function symbols in \( F' \), which means that for every sort \( s' \in S' \) and each \( a \in (M|_{S'})_s' \), there must exist a \( \Sigma' \)-term \( t \) with function symbols only from \( F' \) and variables of sorts not in \( S' \) and a variable assignment \( \rho \) in \( M|_{S'} \) such that the interpretation of \( t \) in \( M|_{S'} \) under \( \rho \) 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”). \( \square \)

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:

\[
\begin{array}{ccc}
A \cap B & \rightarrow & A \\
\downarrow & & \downarrow \\
B & \rightarrow & A \cup B
\end{array}
\]

\( \square \)

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 $I = (\text{Sig}, \text{Sen}, \text{Mod}, \models)$ is inclusive\textsuperscript{11} if

- $\text{Sig}$ is an inclusive category,
- $\text{Sen}$ is inclusive and preserves intersections\textsuperscript{12}

Moreover, we assume that reducts w.r.t. signature inclusions are surjective on objects. \textsuperscript{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 $\iota: \Sigma_1 \hookrightarrow \Sigma_2$ and $M \in \text{Mod}(\Sigma_2)$, we write $M|_{\Sigma_1}$ for $M|_{\iota}$. Note that $\text{Sen}(\iota): \text{Sen}(\Sigma_1) \to \text{Sen}(\Sigma_2)$ 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\textsuperscript{14}) 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 $\Delta \subseteq \Delta'$ 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.

\[
\begin{array}{ccc}
A & \xrightarrow{\Delta} & X \\
\downarrow m & & \downarrow m(\Delta) \\
B & \xrightarrow{m(\Delta)} & B(\Delta)
\end{array}
\]

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 \textsuperscript{15}:

- when $m = \text{id}_A$, $A(\Delta) = X$ and $\text{id}_A(\Delta) = \text{id}_X$
- when $\Delta = \text{id}_A$, $B(\text{id}_A) = B$ and $m(\text{id}_A) = m$
- $(m; n)(\Delta) = m(\Delta); n(\Delta')$ 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

\[
\begin{array}{ccc}
Y & \xrightarrow{(m(\Delta))(\Delta')} & (B(\Delta))(\Delta') \\
\downarrow (m(\Delta))(\Delta') & & \downarrow (B(\Delta))(\Delta') \\
X & \xrightarrow{(B(\Delta))(\Delta')} & B(\Delta)
\end{array}
\]

\[
\begin{array}{ccc}
X & \xrightarrow{m(\Delta; \Delta')} & B(\Delta) \\
\downarrow m(\Delta; \Delta') & & \downarrow (m(\Delta; \Delta')) \\
A & \xrightarrow{m} & B
\end{array}
\]

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

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

\textsuperscript{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 Casi logic, [6] presents a set-theoretical construction of the selected pushout\textsuperscript{14}.

Let $\mathcal{C}$ be a weakly inclusive category. We say that $\mathcal{C}$ 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 $\mathcal{C}$, 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:

$$
\begin{array}{ccc}
\Sigma_1 \cup \Sigma_2 & \xrightarrow{\sigma_1 \cup \sigma_2} & \Sigma_1' \cup \Sigma_2' \\
\Sigma_1 & \xrightarrow{\sigma_1} & \Sigma_1' \\
\Sigma_2 & \xrightarrow{\sigma_2} & \Sigma_2' \\
\end{array}
$$

The union of the two arrows is defined only if the outer diagram is a cocone. Intuitively, this condition ensures that $\sigma_1$ and $\sigma_2$ agree on $\Sigma_1 \cap \Sigma_2$. \hfill $\blacksquare$

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 $M = \{M_p\}_{p \in \text{Nodes}(D)}$ indexed by the nodes of $D$ is compatible with $D$ if for each node $p$ of $D$, $M_p \in \text{Mod}(D(p))$ and for each edge $e : p \to q$, $M_p = M_q|_{D(e)}$. A sink $\eta$ on a subset $K$ of nodes consists of a signature $\Sigma$ together with a family of morphisms $[\eta_p : D(p) \to \Sigma]_{p \in K}$. We say that $D$ ensures amalgamability along $\eta = (\Sigma, [\eta_p : D(p) \to \Sigma]_{p \in K})$ if for every model family $M$ compatible with $D$ there is a unique model $M \in \text{Mod}(\Sigma)$ 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 $\eta$. Moreover, if $K$ consists of all nodes of $D$ and for each edge $i : p \to q$ in $D$ we have that $\eta_p = D(i) ; \eta_q$, $\eta$ is called a (weakly) amalgamable cocone for $D$. Given a family of signatures $\{\Sigma_i\}_{i = 1,...,n}$ and a family of $\Sigma_i$-models $M = \{M_i \in \text{Mod}(\Sigma_i)\}_{i = 1,...,n}$, we define $M_1 \oplus \cdots \oplus M_n = \{M \in \text{Mod}(\Sigma_1 \cup \cdots \cup \Sigma_n) \mid M_{\Sigma_i} \in M_i, i \in 1, \ldots, n\}$.

For an inclusive institution $I$, a presentation is a pair $(\Sigma, E)$ where $\Sigma$ is a signature and $E$ is a set of $\Sigma$-sentences. For a set $E$ of $\Sigma$-sentences, $\text{Mod}_E(\Sigma)$ denotes the class of all $\Sigma$-models satisfying $E$. If $E_1$ and $E_2$ are sets of $\Sigma$-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 $I^{\text{pre}}$, having as its category of signatures the category of $I$-presentations and their morphisms, as sentences over a presentation $(\Sigma, E)$ simply all $\Sigma$-sentences, and $\text{Mod}(\Sigma, E) = \text{Mod}_E(\Sigma)$; the satisfaction relation is the same as in $I$. Presentations provide the simplest form of specifications, and they provide the formalization of basic specifications.

Casi 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 \cite{3, 23, 24} and the formalism of development graphs \cite{25}. See also the discussion in Sect. 6. \hfill $\blacksquare$
4. Semantics of Casi. Refinements

Before we come to the semantics of refinements, we first need to discuss the semantics of both structured and architectural specifications. Note that Casi has model-theoretic semantics: a Casi 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 Casi.

4.1. Semantics of Casi. 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 $\vdash SP \triangleright \Sigma$, where $\Sigma$ is a signature, and then we sometimes write $\text{Sig}(SP)$ for $\Sigma$. Similarly, the class of models of $SP$ will be given by the rules of the model semantics, with judgements of the form $\vdash SP \Rightarrow M$, where $M$ is a class of $\Sigma$-models, and then we sometimes write $\text{Mod}(SP)$ for $M$. A structured specification $SP$ is consistent if its model class $\text{Mod}(SP)$ is non-empty.

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

  $\vdash (\Sigma, E) \triangleright \Sigma \quad \vdash (\Sigma, E) \Rightarrow \text{Mod}_E(E)$

- **Union**:

  $\vdash SP_1 \triangleright \Sigma_1 \quad \vdash SP_2 \triangleright \Sigma_2 \quad \vdash SP_1 \triangleright \Sigma_1 \cup \Sigma_2 \quad \vdash SP_2 \triangleright \Sigma_2 \Rightarrow \{M \in \text{Mod}(\Sigma_1 \cup \Sigma_2) \mid M\vert_{\Sigma_1\cup\Sigma_2} \in M_i, i = 1, 2\}$

- **Translation**:

  $\vdash SP \triangleright \Sigma \quad \sigma : \Sigma \rightarrow \Sigma' \quad \vdash SP \text{ with } \sigma \triangleright \Sigma' \quad \vdash SP \Rightarrow M \quad \vdash SP' \text{ with } \sigma' \triangleright \Sigma' \quad \vdash SP' \Rightarrow M' \quad \vdash SP' \text{ hide } \sigma' \triangleright \Sigma \quad \vdash SP' \text{ hide } \sigma' \Rightarrow M' \text{, where } M' \in M$

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 $USP = SP_1, \ldots, SP_n \rightarrow SP$ 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:

$\vdash SP_1 \triangleright \Sigma_1$

$\vdots$

$\vdash SP_n \triangleright \Sigma_n$

$\vdash SP \triangleright \Sigma$

$\Sigma$ extends $\Sigma_1 \cup \cdots \cup \Sigma_n$

$\vdash USP \triangleright \Sigma_1, \ldots, \Sigma_n \Rightarrow \Sigma$

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

19
The class of all $U\Sigma$-models is denoted $\text{Unit}(U\Sigma)$. The semantics of a unit specification $\text{USP} = \text{SP}_1, \ldots, \text{SP}_n \Rightarrow \text{SP}$ is given by the following rule:

\[
\begin{align*}
\vdash \text{USP} & \triangleright U\Sigma \\
\vdash \text{SP}_1 & \Rightarrow M_1 \\
\vdots \\
\vdash \text{SP}_n & \Rightarrow M_n \\
M_0 = M_1 \oplus \cdots \oplus M_n \\
\vdash \text{SP} & \Rightarrow M
\end{align*}
\]

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 $\text{USP}$ as defined in the model semantics by $\text{Unit}(\text{USP})$. A unit specification is consistent if $\text{Unit}(\text{USP})$ is non-empty.

### 4.3. Semantics of Architectural Specifications

![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 $A\Sigma = (C, U\Sigma)$ consists of a unit signature $U\Sigma$ for the result unit together with a static context $C$, which is a map assigning unit signatures to the names of component units.

Starting with the initial empty static context $C_0$, 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 $\Gamma, A\Sigma \triangleright \text{ASP} \Rightarrow A\Sigma$ where $A\Sigma$ is an architectural signature and $\Gamma$ 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. It classifies a class of architectural specifications $A\Sigma$ over the resulting 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 $\Gamma, \Gamma_m \triangleright A\Sigma \Rightarrow A\Sigma$ with $A\Sigma$ being an architectural model and $\Gamma_m$ being the global model environment, keeping track of the model class of the named structured specifications.

Figure 8 gives the basic static and model semantics for unit declarations. Unit declarations and definitions denote a set $C$ of unit environments. We write $C^0$ 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 $\text{ASP}$, we write $\vdash \text{ASP} \triangleright \Box$. An architectural specification $\text{ASP}$ 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 case $\Box$. If that is the case $\Box$, we have $\vdash \text{ASP} \triangleright \Box$.
The general case of units with multiple arguments, the signature is obtained with the following definition.

Definition 4.1. Let $F$ be a generic unit with the unit signature $\Sigma$. The static semantics produces the signature of the term $T$ as the union of all $\Sigma_i$, where $\Delta$ is the unit signature of $F$ stored in the context $C_r$. In the general case of units with multiple arguments, the signature is obtained with the following definition.

$$C \vdash T \Rightarrow \text{UE}_{\Sigma}$$

for each $E \in C$, there is a unique $M \in \text{Mod}(\Sigma(\Delta))$ such that

$$\text{UE}_{\Sigma} = \{ E \mapsto M \mid E \in C, M|_{\Sigma} = \text{UE}_{\Sigma}(E) \},$$

and returns as signature of $F[T]$ the selected pushout $\Sigma(\Delta)$ of the span $(\sigma, \Delta)$, where $\Delta$ is the unit signature of $F$ stored in the context $C_r$. In the general case of units with multiple arguments, the signature is obtained with the following definition.

Definition 4.1. Let $F$ be a generic unit with the unit signature $\Sigma_1, \ldots, \Sigma_n \rightarrow \Sigma$, and let $F[T_i \ fit \ \sigma_i] \ldots [T_n \ fit \ \sigma_n]$ be a unit term involving $F$. Let $\Sigma_i^A$ be the signature of the unit term $T_i$ for $i = 1, \ldots, n$. Let $\Sigma^A$ be the union of all $\Sigma_i^A$. Let $\Delta : \Sigma^F \rightarrow \Sigma$ be the inclusion of $\Sigma^F$ into $\Sigma$. Let $\sigma : \Sigma^F \rightarrow \Sigma^A$ be the union of the fitting morphisms $\sigma_i : \Sigma_i \rightarrow \Sigma_i^A$. Let $(\Sigma(\Delta), \sigma(\Delta), t_{\Sigma(\Delta)}^A, \Sigma^A(\Delta))$ be the selected pushout for $(\sigma, \Delta)$:
Then the signature of the term $F[T_1 \triangleright \sigma_1] \ldots [T_n \triangleright \sigma_n]$ 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 $\sigma$ the actual parameter fits the domain and $(\star)$ the models $UEv(E)$ and $E(F)\langle UEv(E) \rangle_C$ can be amalgamated to a $\Sigma_C$-model $M$, the result unit evaluator $UEv_R$ gives the amalgamation $M$ for each unit environment $E \in C$. The condition $(\star)$ will be discharged with the proof calculus for architectural specifications in Sect. 5.7.

Typically one would expect conditions of type $(\star)$ to be discarded statically. For this purpose, an extended static semantics was introduced in [29], 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 $\triangleright ASP \Rightarrow \Delta AM$ 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:

$$
R \Sigma := \langle U \Sigma, B \Sigma \rangle \mid \{UN_i \mapsto R \Sigma\}_{i \in J}
$$

$$
B \Sigma := U \Sigma \mid \{UN_i \mapsto B \Sigma\}_{i \in J}
$$

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

**Example 4.1.** The signature of $R1$ in Example 2.3 is the simple refinement signature $(\Sigma(Monoid), \Sigma(\text{Nat}))$. The signature of $R4$ in Example 2.4 is the branching refinement signature $(\Sigma(\text{NatWrnStr(c)}), \{N \mapsto \Sigma(\text{Nat}), F \mapsto \Sigma(\text{Nat}) \rightarrow \Sigma(\text{NatWrnStr(c)})\})$. The signature of $RComp$ in Example 2.5 is $\{N \mapsto (\Sigma(\text{Nat}), \Sigma(\text{NatBox}))\}$. 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 \text{RefComp}' = \{ N \to R2, F \to R' \}

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

\textbf{Definition 4.3.} Given refinement signatures \(R_1 \Sigma\) and \(R_2 \Sigma\), their composition \(R_1 \Sigma \circ R_2 \Sigma\) is defined inductively on the form of the first argument:

\begin{align*}
S;S;B \quad R_1 \Sigma = (U \Sigma, U \Sigma'); \text{ then } R_1 \Sigma \circ R_2 \Sigma \text{ is defined only if } R_2 \Sigma \text{ is a branching refinement signature of the form } (U \Sigma', B \Sigma''). \text{ Then } R_1 \Sigma \circ R_2 \Sigma = (U \Sigma, B \Sigma''). \\
B;C \quad R_1 \Sigma = (U \Sigma, BstC); \text{ then } R_1 \Sigma \circ R_2 \Sigma \text{ is defined only if } R_2 \Sigma \text{ is a component refinement signature that matches } BstC', \text{ i.e., } \text{dom}(R_2 \Sigma) \subseteq \text{dom}(BstC') \text{ and for each } UN \in \text{dom}(R_2 \Sigma),
\begin{itemize}
\item either } BstC'(UN) \text{ is a unit signature and then } R_2 \Sigma(UN) = (U \Sigma', B \Sigma'') \text{ with } U \Sigma' = BstC'(UN), \text{ or}
\item } BstC'(UN) \text{ is a branching static context and then } R_2 \Sigma(UN) \text{ matches } BstC'(UN),
\end{itemize}
\text{Then } R_1 \Sigma \circ R_2 \Sigma = (U \Sigma, BstC'(R_2 \Sigma)), \text{ where given any branching static context } BstC' \text{ and component refinement signature } R_2 \Sigma \text{ that matches } BstC', \text{ } BstC'[R_2 \Sigma] \text{ modifies } BstC' \text{ on each } UN \in \text{dom}(R_2 \Sigma) \text{ as follows:}
\begin{itemize}
\item if } BstC'(UN) \text{ is a unit signature then } BstC'[R_2 \Sigma](UN) = B \Sigma'' \text{ where } R_2 \Sigma(UN) = (BstC'(UN), B \Sigma''),
\item if } BstC'(UN) \text{ is a branching static context then } BstC'[R_2 \Sigma](UN) = BstC'(UN)[R_2 \Sigma(UN)].
\end{itemize}
\end{align*}

\(C;C \quad R_1 \Sigma\) is a component refinement signature: then \(R_1 \Sigma \circ R_2 \Sigma\) is defined only if \(R_2 \Sigma\) is a component refinement signature too, and moreover, for all \(UN \in \text{dom}(R_1 \Sigma) \cap \text{dom}(R_2 \Sigma), R_2 \Sigma(UN) = R_1 \Sigma(UN); R_2 \Sigma(UN)\) is defined. Then \(R_1 \Sigma \circ R_2 \Sigma\) is the component refinement signature with \(\text{dom}(R_1 \Sigma ; R_2 \Sigma) = \text{dom}(R_1 \Sigma) \cup \text{dom}(R_2 \Sigma) \text{ that on } UN \in \text{dom}(R_1 \Sigma) \setminus \text{dom}(R_2 \Sigma) \text{ coincides with } R_1 \Sigma, \text{ on } UN \in \text{dom}(R_2 \Sigma) \setminus \text{dom}(R_1 \Sigma) \text{ coincides with } R_2 \Sigma \text{ and on } UN \in \text{dom}(R_1 \Sigma) \cap \text{dom}(R_2 \Sigma)\) is by definition \(R_2 \Sigma(UN)\).

\textbf{Example 4.2.} With the specifications in Example 2.3 we can compose the signature \((\text{Sig}(\text{Monom}), \text{Sig}(\text{Nat}))\) of \(R_1\) with the signature \((\text{Sig}(\text{Nat}), \text{Sig}(\text{NatBin}))\) of \(R_2\) using the case \(S;S;B\) and we obtain \((\text{Sig}(\text{Monom}), \text{Sig}(\text{NatBin})).\) The branching refinement signature \((\text{Sig}(\text{NatWithSuc})), [N \mapsto \text{Sig}(\text{Nat}), F \mapsto \text{Sig}(\text{Nat} \mapsto \text{Sig}(\text{NatWithSuc})))\) of \text{ADDITION\_FIRST} can be composed with the component refinement signature \([N \mapsto (\text{Sig}(\text{Nat}), \text{Sig}(\text{NatBin}))\]) of \text{RCOMP} using the case \(B;C\) and the result is \((\text{Sig}(\text{NatWithSuc})), [N \mapsto \text{Sig}(\text{NatBin}), F \mapsto \text{Sig}(\text{Nat} \mapsto \text{Sig}(\text{NatWithSuc}))).\) For the case \(C;C\) we consider the following refinement:

\textbf{refinement} \text{CompRef1} = \{ A \to R4 \}

\textbf{refinement} \text{CompRef2} = \{ A \to [N \to R2] \}

\textbf{refinement} \text{RefComposition} = \text{CompRef1} \text{ then } \text{CompRef2}

where \text{CompRef1} states that a component named \(A\) of an architectural specification should be refined as described by \(R4\) and \text{CompRef2} states that the unit \(N\) of the component \(A\) should be refined as described by \(R2\). The signature of \text{CompRef1} is the component refinement signature \([A \mapsto (\text{Sig}(\text{NatWithSuc}), [N \mapsto \text{Sig}(\text{Nat}), F \mapsto \text{Sig}(\text{Nat} \mapsto \text{Sig}(\text{NatWithSuc})))\]) and the signature of \text{CompRef2} is \([A \mapsto [N \mapsto (\text{Sig}(\text{Nat}), \text{Sig}(\text{NatBin}))])\). The signature of \text{RefComposition} is then \([A \mapsto (\text{Sig}(\text{NatWithSuc}), [N \mapsto \text{Sig}(\text{NatBin}), F \mapsto \text{Sig}(\text{Nat} \mapsto \text{Sig}(\text{NatWithSuc})))].\)
The rules for static semantics of refinements are presented in Fig. 11. The general format of the judgements is 
\[ \Gamma \vdash \text{USP } \Rightarrow \Sigma \]
and the result of the analysis is a refinement signature \( \Sigma \). When we only want to state that the rules apply successfully and the result is not important, we will denote this \( \Gamma \vdash \text{USP } \Rightarrow \square \). We have a rule for each alternative form of refinement (see non-terminal \( \text{SPR} \) 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 \( \text{USP} \) and returns as result of the analysis of the unit specification regarded as a refinement specification (this is marked by \( \Gamma \vdash \text{USP qua SPR } \Rightarrow (U \Sigma, U \Sigma) \)) the pair \((U \Sigma, U \Sigma)\). In the second rule, if \( \sigma \) 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:

\[ \Gamma, \text{USP } \Rightarrow \Sigma \]
\[ \Gamma, \text{USP } \Rightarrow \Sigma, \text{UN } \notin \text{Dom}(RstC) \]
\[ \Gamma, \text{USP } \Rightarrow \Sigma, \text{UN } \notin \text{Dom}(RstC) \]

Note that here, as in the case of plain architectural static semantics, we build for each declared unit \( \text{UN} \) a refined-unit static context that maps \( \text{UN} \) to the refinement signature \( \Sigma \) of its specification \( \text{SPR} \). 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 \( \text{UN} \). 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 \( \pi_1 \) where required by the rest of plain architectural static semantics. For example, the rule for unit expressions uses judgements of the form \( \Gamma, \text{C } \Rightarrow \text{UE } \Rightarrow \Sigma \) and we can apply them in our new setting using \( \Gamma, \pi_1(RstC) \Rightarrow \text{UE } \Rightarrow \Sigma \). 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:

\[ \Gamma, \pi_1(RstC) \Rightarrow \text{UE } \Rightarrow \Sigma \]
\[ \Gamma, \text{UN } \notin \text{Dom}(RstC) \]
\[ \Gamma, \text{UN } \notin \text{Dom}(RstC) \]

Figure 11: Static semantics of CASL refinements
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 ∅ 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 π₁, 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 \( \kappa : \text{Mod}(\Sigma) \rightarrow \text{Mod}(\Sigma') \). An example of such a constructor is already provided by the model reduct along a signature morphism \( \sigma : \Sigma \rightarrow \Sigma' \), which takes any \( \Sigma' \)-model \( M' \) to its \( \Sigma \)-reduct \( M'_{\sigma} \).

**Definition 4.4.** Let \( SP, SP' \) be specifications such that \( \triangleright SP \triangleright \Sigma \) and \( \triangleright SP' \triangleright \Sigma' \) and let \( \kappa : \text{Mod}(\Sigma') \rightarrow \text{Mod}(\Sigma) \) be a constructor. We say that \( SP' \) is a constructor implementation of \( SP \) along \( \kappa \), denoted \( SP \xrightarrow{\kappa} SP' \), if for all \( M' \in \text{Mod}(SP') \), \( M' \in \text{dom}(\kappa) \) and \( \kappa(M') \in \text{Mod}(SP) \).

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

To capture branchings, we need \( n \)-ary constructors of the form \( \kappa : \text{Mod}(\Sigma_1) \times \ldots \times \text{Mod}(\Sigma_n) \rightarrow \text{Mod}(\Sigma) \), which are partial functions mapping compatible models over \( \Sigma_1, \ldots, \Sigma_n \) to \( \Sigma \)-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 \( \Sigma_i \)-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 \( \kappa \) 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 \( SP \xrightarrow{\kappa} SP' \), we denote by \( G \) the graph of \( \kappa \) restricted to \( \text{Mod}(SP) \), that is, pairs of the form \((M', \kappa(M'))\), where \( M' \in \text{Mod}(SP) \). We then take the inverse relation \( G^{-1} \) to obtain on the first component models over \( \text{Sig}(SP) \) and on the second component models over \( \text{Sig}(SP') \), 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 \( R \), is a class of \( R\Sigma \)-assignments, usually denoted \( R \), which can themselves be:

- S/B branching assignments: for \( R\Sigma = (US, BE_\Sigma) \), these are pairs \((U, BM')\), where (1) \( U \) is a unit over the unit signature \( US \) and (2) \( BM' \) is a branching model over the branching signature \( BE_\Sigma \), i.e. (2.1) a unit over \( BE' \) when \( BE' \) is a unit signature (in which case the branching assignment is a unit assignment), or (2.2) a branching environment \( BE' \) that fits \( BE_\Sigma \) when \( BE_\Sigma \) 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 \( R \) then \( U = U' \) (so that the corresponding constructor is a function, which in general is partial).

- C component assignments: for \( R\Sigma = \{U_i \mapsto R\Sigma_i\}_{i \in \mathcal{I}} \), these are (finite) maps \( \{U_i \mapsto U_i\}_{i \in \mathcal{I}} \) from unit names to assignments over the respective refinement signatures. When \( R\Sigma \) is a refined-unit static context (and so each \( R_i \), \( i \in \mathcal{I} \), is a branching assignment) we refer to \( R \) as \( \{U_i \mapsto (U_i, BM_i)\}_{i \in \mathcal{I}} \) as a refined-unit environment. Any such refined-unit environment can be naturally coerced to a unit environment \( \pi_1(RE) = \{U_i \mapsto U_i\}_{i \in \mathcal{I}} \) of the plain Casi semantics, as well as to a branching environment \( \pi_2(RE) = \{U_i \mapsto BM_i\}_{i \in \mathcal{I}} \).
Example 4.3. We illustrate the concept of refinement relation using the specifications in Example 2.3. The signature of R1 is \( \{ \text{Sig} \text{(Monoid)} \} \) and a refinement relation over this signature is \(((N, \sigma, N) \mid N \in \text{Mod}(\text{Nar}))\) where \( \sigma : \text{Sig}(\text{Monoid}) \rightarrow \text{Sig}(\text{Nar}) \) is the unique signature morphism taking the sort \( \text{Elem} \) to \( \text{Nat} \). Note that the class \(((M, N) \mid M \in \text{Mod}(\text{Monoid}), N \in \text{Mod}(\text{Nar}))\) is not a refinement relation, as it contains for each model \( N \in \text{Mod}(\text{Nar}) \) pairs \((M, N)\) and \((M', N)\) with \( M \) and \( M' \) being two different models of Monoid.

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

A component assignment over the signature \( [N \mapsto (\text{Sig}(\text{Nat}), \text{Sig}(\text{NatBin}))] \) of RComp is \([N \mapsto ((\text{BM}_1, \text{BM}) \mid \text{BM} \in \text{Mod}(\text{NatBin}))]\) where \( \sigma' : \text{Sig}(\text{Nat}) \rightarrow \text{Sig}(\text{NatBin}) \) is the signature morphism mapping the sort \( \text{Nat} \) to \( \text{Bin} \), 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 \( \text{BE} \) in the domain of \( R_1 \) and replace, for each unit \( UN \) in the domain of the second constructor \( R_2 \), the unit \( \text{BE}(UN) \) with a unit delivered by the second constructor, we get a new branching environment \( \text{BE}[R_2] \) which should still be in the domain of \( R_1 \). For each such unit name \( UN \) the domain of the composition is now determined by the domain of \( R_2(UN) \).

C;C Here, we require that for each unit name \( UN \) in the domains of \( R_1 \) and \( R_2 \), the image of \( R_2(UN) \) should be in the domain of \( R_1(UN) \). This might require several recursive applications of the definition, until \( R_k(UN) \) 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 \( R_1, R_2 \) over refinement signatures \( R_1, R_2 \), respectively, such that the composition \( R_2 = R_1 \cdot R_2 \) is defined, the composition \( R_1 ; R_2 \) is defined as a refinement relation over \( R_2 \) as follows:

\[
\text{S;S;S;B } R_1 = (U, \Sigma, U', \Sigma'), R_2 = (U', \Sigma', B, \Sigma'') \Rightarrow \text{then } R_1 ; R_2 \text{ is defined only if for all } (U', BM'') \in R_2 \text{ we have } (U, U') \in R_1 \text{ for some } U. \text{ Then } \]

\[
R_1 ; R_2 = \{(U, BM'') \mid (U, U') \in R_1, (U', BM'') \in R_2 \text{ for some } U'\}
\]

\[
\text{B;C } R_2 = (U, \Sigma, B, \Sigma) \text{ and } R_2 = (U', \Sigma, B', \Sigma') \text{ is a component refinement signature that matches } B; \text{ then } R_1 ; R_2 \text{ is defined only if for each } (U, \text{BE}) \in R_1 \text{ and each } R_2 \in R_2 \text{ there exists } U' \in \text{Unit}(\Sigma) \text{ such that } (U', \text{BE}[R_2]) \in R_1, \text{ where } \text{BE}[R_2] \text{ modifies BE on each } UN \in \text{dom}(R_2) \text{ as follows:}
\]

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

Then

\[
R_1 ; R_2 = \{(U', \text{BE}[R_2]) \mid (U, \text{BE}) \in R_1, R_2 \in R_2\}
\]

where \( U' \) is determined by the requirement that \( (U', \text{BE}[R_2]) \) should be in \( R_1 \) and \( \text{BE}[R_2] \) modifies \( \text{BE} \) on each \( UN \in \text{dom}(R_2) \) as follows:
• if \( BstC(\text{UN}) \) is a unit signature then \( BE(R_2(\text{UN})) = BE'' \) where \( R_2(\text{UN}) = (U'', BE'') \);
• if \( BstC'(\text{UN}) \) is a branching static context then we put \( BE(R_2(\text{UN})) = BE(UN)(R_2(\text{UN})) \).

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

- if \( R_2(UN) = (U', BM) \), there exists \( (U, U') \in R_1(UN) \), where \( R_1(UN) = \{ R(UN) | R \in R_1 \} \).
- if \( R_2(UN) \) is a component assignment,
  - if \( R_1(UN) = (U, BE) \) then there exists \( (U', BE[R_2(UN)]) \in R_1(UN) \), 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 \( R_2(UN) \) matches \( R_1(UN) \).

When defined, \( R_1 ; R_2 \) is the class of all assignments \( R_1(R_2) \) where \( R_1 \in R_1 \), \( R_2 \in R_2 \) and we define \( R_1(R_2) \) as the assignment with \( \text{dom}(R_1(R_2)) = \text{dom}(R_1) \cup \text{dom}(R_2) \) that on \( UN \in \text{dom}(R_1) \backslash \text{dom}(R_2) \) coincides with \( R_1 \), on \( UN \in \text{dom}(R_2) \backslash \text{dom}(R_1) \) coincides with \( R_2 \), and on \( UN \in \text{dom}(R_1) \cap \text{dom}(R_2) \)

- if \( R_1(UN) \) is a component refinement signature, and \( R_2(UN) = (U', BM'') \), then by the matching condition there must exist \( (U, U') \in R_1(UN) \). The uniqueness of \( U \) is ensured by the definition of assignments. Then \( R_1(R_2)(UN) = (U, BM'') \);
- if \( R_1(UN) \) is a branching refinement signature, and \( R_1(UN) = (U, BE) \), then we define \( R_1(R_2)(UN) = (U', BE[R_2(UN)]) \) 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 \( R_1(UN) \), and \( BE(R_2(UN)) \) is as defined in the second case of the main definition;
- if \( R_1(UN) \) is a component refinement signature, then \( R_1(R_2)(UN) = R_1(UN)(R_2(UN)) \).

\( \square \)

The model semantics of refinements is presented in Fig. 12. The judgements are now of the form \( \vdash \text{SPR} \Rightarrow \mathcal{R} \), where \( \text{SPR} \) 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 \( \text{RE} \) for the units of an architectural specification that we can coerce to a unit environment using the projection to the first component \( \pi_1 \). Then, the result unit expression is analysed using the rules of the plain architectural model semantics of the form \( \Gamma, \Gamma_m, C, \sigma \vdash UE \Rightarrow U \) in our new setting using \( \Gamma, \Gamma_m, \pi_1(\text{RstC}), \pi_1(\text{RE}) \vdash UE \Rightarrow U \), where \( \text{RstC} \) is obtained with the rules of the static semantics. Finally, the result of the analysis of the architectural specification is the class of all \( (U, \pi_2(\text{RE})) \).

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

**Definition 4.7.** A refinement \( \text{SPR} \) 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]

\begin{align*}
\text{unit spec } &\text{USP} = SP \rightarrow SP' \\
\text{unit spec } &\text{USP'} = SP_1 \rightarrow SP'_1 \\
\text{refinement } &\text{R} = \text{USP refined via } \tau \text{ to arch spec } \{
\text{units } F : \text{USP'} \\
\text{result lambda } X : SP \bullet F [X \text{ fit } \sigma]\}
\end{align*}

[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.
\[ \vdash \text{USP} \Rightarrow \mathcal{U} \]
\[ \vdash \text{USP qua SPR} \Rightarrow \{ (U, U) \mid U \in \mathcal{U} \} \]
\[ \vdash \text{USP} \Rightarrow \mathcal{U} \quad \text{\( \sigma : \Sigma \rightarrow \Sigma' \), \( \vdash \text{SPR} \Rightarrow \mathcal{R} \)} \]
\[ U'_1, \ldots, U'_n \in \mathcal{U}, \text{ for all } (U'_1, BM''_1) \in \mathcal{R} \]
\[ \mathcal{R}' = \{ (U'_1, BM''_1) \mid (U'_1, BM''_1) \in \mathcal{R} \} \]
\[ \vdash \text{USP refined via } \sigma \text{ to } \text{SPR} \Rightarrow \mathcal{R}' \]

\[ \vdash \text{ASP} \Rightarrow \mathcal{AM} \quad \vdash \text{arch spec } \text{ASP} \Rightarrow \{ (U, \pi_2(\text{RE})) \mid (U, \text{RE}) \in \mathcal{AM} \} \]

\[ \vdash \text{SPR}_1 \Rightarrow \mathcal{R}_1 \quad \ldots \quad \vdash \text{SPR}_n \Rightarrow \mathcal{R}_n \]
\[ \vdash \{ \text{UN}_1 \text{ to } \text{SPR}_1, \ldots, \text{UN}_n \text{ to } \text{SPR}_n \} \Rightarrow \{ \mathcal{R} \mid \text{dom}(\mathcal{R}) = \{ \text{UN}_1, \ldots, \text{UN}_n \}, \mathcal{R}(\text{UN}_1) \in \mathcal{R}_1, \ldots, \mathcal{R}(\text{UN}_n) \in \mathcal{R}_n \} \]
\[ \vdash \text{SPR}_1 \Rightarrow \mathcal{R}_1 \quad \vdash \text{SPR}_2 \Rightarrow \mathcal{R}_2 \quad \mathcal{R} = \mathcal{R}_1 ; \mathcal{R}_2 \]
\[ \vdash \text{SPR}_1 \text{ then } \text{SPR}_2 \Rightarrow \mathcal{R} \]

Figure 12: Model semantics of CASL refinements

where \( \tau \) is a specification morphism from \( SP' \) to the specification \( SP'_1 \oplus SP \) in the following diagram:

\[
\begin{array}{ccc}
SP_1 & \xrightarrow{\theta} & SP'_1 \\
\downarrow{\sigma} & & \downarrow{\sigma'} \\
SP & \xrightarrow{\alpha} & SP'_1 \oplus SP \\
\downarrow{\tau} & & \downarrow{\tau} \\
SP' & &
\end{array}
\]

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 \( \{ SP'_1 \text{ with } \sigma' \} \text{ and } \{ SP \text{ with } \theta' \} \).

\[ \square \]

5. Calcuhi 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.
within USP is inconsistent. For each unit specification result unit that can be built following the architectural specification ASP I architectural specification over an institution the box preceding the rules for each judgement). The main judgement is of the form ⊢ SP unit signature of ASP unit specification. It follows easily from a result in [6].

Let us consider the following specifications:

Example 5.1. Let us consider the following specifications:

\[
\begin{align*}
\text{spec} & \text{ Consts} = \\
& \text{ sort } s \\
& \text{ ops } a, b : s \\
\text{spec} & \text{ EqConsts } = \text{ Consts} \\
& \text{ then } \bullet a = b
\end{align*}
\]
The unit specification $\text{DiffConsts} \rightarrow \text{EqConsts}$ 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, $\text{ASP\_NAME}$ denotes the empty class of models. Hence, the right hand-side of the equivalence in Thm. 5.1 holds for any unit specification $\text{USP}$ over the signature of $\text{Consts}$. However, according to the rule for unit applications, $F[U]$ is correct only if for any family of models $M$ compatible with $\text{DiffConsts}$, $M_U$ is a model of $\text{DiffConsts}$. For any such family $M$ we have however by construction of the diagram of $\text{ASP\_NAME}$ that $M_U$ is a model of $\text{EqConsts}$, which cannot be at the same time a model of $\text{DiffConsts}$. Therefore the left hand-side of the equivalence is false for any unit specification.
Example 5.2. Let us consider the following architectural specification:

\[ S_{\text{ASP}}(UE) \]

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 \( S_{\text{ASP}}(UE) \) 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, \( S_{\text{ASP}}(UT \ with \ \sigma) = S_{\text{ASP}}(UT) \ with \ \sigma \). Similarly for hiding. A complication arises when determining the unit specification of a unit amalgamation or unit application. The naïve solution, to define \( S_{\text{ASP}}(UT_1 \ and \ UT_2) \) to be \( S_{\text{ASP}}(UT_1) \ and \ S_{\text{ASP}}(UT_2) \), 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 $S_{ASPNAMe}(UN) = (sort s; \emptyset)$. The naïve definition of $S_{ASPNAMe}(UT)$ 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 $BM(UN)$. 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:

$$
\begin{array}{c}
  \vdash t \\
  s \rightarrow u \\
  \downarrow \downarrow \\
  t, u \rightarrow s
\end{array}
$$

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 $\Gamma$ 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 $\Gamma$ corresponding to the node of $UT$.

Then we define $S_{amalg}(UT) = (\Sigma, \emptyset)$ hide $\eta_\Lambda$ where $\Lambda$ is the node of the unit term $UT$ in $D_{UT}$ and $\eta = (\Sigma_i | \forall x : \Sigma_i \in \text{dom}(D_{UT}))$ 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 $\Gamma_i$ assigning unit specifications to unit names. We denote the empty verification context by $\Gamma_\emptyset$.

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

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

- if $UE$ is a unit term $UT$, then $S_{\Gamma_i}(UT)$ is defined inductively:
  - if $UT$ is a unit name, then $S_{\Gamma_i}(UT) = SP$ where $\Gamma_i(UT) = SP$;
  - if $S_{\Gamma_i}(UT) = SP$, then $S_{\Gamma_i}(UT \text{ with } \sigma) = SP \text{ with } \sigma$;
  - if $S_{\Gamma_i}(UT) = SP$, then $S_{\Gamma_i}(UT \text{ hide } \sigma) = SP \text{ hide } \sigma$;
  - if $UT = UT_1$ and $UT_2$ and $S_{\Gamma_i}(UT_i) = SP_i$ for $i = 1, 2$ then $S_{\Gamma_i}(UT) = SP_1$ and $SP_2$ and $S_{\Gamma_i}(UE)$;
  - if $UT = F[UT_1 \text{ fit } \sigma_1, \ldots, UT_n \text{ fit } \sigma_n]$, where $\Gamma_i(F) = SP_1 \times \cdots \times SP_n \mapsto SP$ and for $i = 1, \ldots, n$, the verification condition $SP_i$, with $\sigma_i \mapsto S_{\Gamma_i}(UT_i)$ holds, then $S_{\Gamma_i}(UT) = [SP \text{ with } \sigma] \text{ and } S_{\Gamma_i}(UT_i) \text{ with } \tau_i$ and $\ldots$, and $S_{\Gamma_i}(UT_n)$ with $\tau_n$ and $S_{\Gamma_i}(UE)$, where the application is done as in the diagram below, with $\Sigma = \text{Sig}[SP_i]$, $\Sigma_0 = \text{Sig}[UT_i]$, $\Sigma_f = \bigcup_{i=1\ldots n} \Sigma_i$, $\Sigma_\delta = \text{Sig}[SP]$, $\Sigma_\delta_u = \bigcup_{i=1\ldots n} \Sigma_i^1$, $\Delta$ and $\tau$ and all $\tau_i$ are inclusions, $\sigma = \bigcup_{i=1\ldots n} \sigma_i$; $\Sigma_f \mapsto \Sigma_u$, and $(\Sigma_\delta, \tau, \sigma(\Delta))$ is the selected pushout of $(\Delta, \sigma)$ (see Def. 4.7).

\[\text{Definition 5.3.}\]

\[\text{Definition 5.2.}\]

\[\text{Definition 5.1.}\]
However, a model of $S$ is also a model of this specification. Consider the following architectural specification:

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

\[
\begin{align*}
&S_\Gamma, (\text{local UDEFN within UT}) = S_\Gamma (UT), \text{ where } \Gamma' \text{ extends } \Gamma, \text{ according to UDEFN in the obvious way.}
\end{align*}
\]

- If UE is a lambda expression $\lambda X : \text{SP. UT}$, then $S_\Gamma (UE) = \text{SP} \rightarrow S_\Gamma (UT)$, where $\Gamma'$ extends $\Gamma$ with $X : \text{SP}$.

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

If $\vdash UT \Sigma_U T$ and no non-generic unit is used more than once in $UT$, directly or indirectly via unit definitions, we have that $S_{\text{analyt}} (UT) = \Sigma_\text{UT} \text{hide id}_\Sigma_T$. This allows us to omit $S_{\text{analyt}} (UT)$ from $S_{\text{ASP.NAME}}(UT)$ 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 $\text{CASL}$ 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 $S_\Gamma (UT)$ as an approximation, since models of the unit term are also models of this specification.

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

\[
\begin{align*}
\text{arch spec ASP.NAME =}
\text{units}
\begin{align*}
A : \{\text{sort } s\}; \\
F : \{\text{sort } s\} \rightarrow \{\text{sort } s; \text{ op } a : s\}; \\
G : \{\text{sort } s\} \rightarrow \{\text{sort } s; \text{ op } a : s\}; \\
H : \{\text{sort } s; \text{ op } a : s\} \rightarrow \{\text{sort } s; \text{ ops } a, b : s\}; \\
B = F [A]; \\
C = G [A]
\end{align*}
\begin{align*}
\text{result } H [B] \text{ and } H [C] \text{ with } a \mapsto a', b \mapsto b'
\end{align*}
\end{align*}
\]

The specification $S_{\text{ASP.NAME}}$ of the result unit term of $\text{ASP.NAME}$ is $\{\text{sort } s; \text{ ops } a, b, a', b' : s\}$. However, if $M$ is a model of $S_{\text{ASP.NAME}}$ 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 $S_{\text{ASP.NAME}}$, which is in this case only an over-approximation of the model class of the result unit term of $\text{ASP.NAME}$.

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 $\vdash \text{ASP } \Sigma \vdash \text{USP}$, where $\text{USP}$ 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 $\text{USP}$ of a unit can itself be an architectural specification. The result of applying the calculus to $\text{USP}$ is a unit specification $\text{USP}'$, which can be either $\text{USP}$ if $\text{USP}$ was already a unit specification (last rule of the calculus) or the specification of the result unit of $\text{USP}$ if $\text{USP}$ is an architectural specification. We moreover use the context of outer declarations of definitions when verifying the architectural specification $\text{USP}'$. 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.
\[\begin{align*}
\Gamma^0 & \vdash \text{ASP} : :_c \text{USP} \\
\vdash \text{arch spec} \text{ASP} : :_c \text{USP} \\
\Gamma & \vdash \text{UDD}_1 : :_c \Gamma_1 \\
& \vdots \\
\Gamma_{n-1} & \vdash \text{UDD}_n : :_c \Gamma_n \\
\Gamma & \vdash \text{units} \ UDD_1 \ldots \ UDD_n \ \text{result} \ UE : :_c S_{\Gamma_1}(UE)
\end{align*}\]

\[\begin{align*}
\Gamma & \vdash \text{UDECL} : :_c \Gamma' \\
\Gamma & \vdash \text{UDEFN} : :_c \Gamma' \\
\Gamma & \vdash \text{USP} : :_c \text{USP}' \\
\Gamma & \vdash \text{UN} : \text{USP} : :_c \Gamma \cup \{\text{UN} \mapsto \text{USP}\} \\
\Gamma & \vdash \text{UN} = \text{UE} : :_c \Gamma \cup \{\text{UN} \mapsto S_{\Gamma}(UE)\}
\end{align*}\]

\[\Gamma \vdash \text{SP}_1 \times \ldots \times \text{SP}_n \rightarrow \text{SP} : :_c \text{SP}_1 \times \ldots \times \text{SP}_n \rightarrow \text{SP}\]

Figure 16: Proof calculus for CASL architectural specifications.

**Framework 5.1.** ASP is an architectural specification such that \(\vdash \text{ASP} \not\models \Box\) and no generic unit specification in ASP is inconsistent.

If an architectural specification ASP is of the form \(\text{units} \ UDD_1 \ldots \ UDD_n \ \text{result} \ UE\), we write \(\vdash \text{UDD} : :_c \Gamma\) if \(\vdash \text{UDD}_i : :_c \Gamma_{gen}\) for \(i = 1, \ldots, n\), \(\Gamma_{gen} = \bigcup_{i=1,\ldots,n} \Gamma_{gen}^i\) and \(\Gamma = \bigcup_{i=1,\ldots,n} \Gamma^i\). Similarly, we write \(\Gamma_0 \vdash \text{UDD}^+ : :_c \Gamma\) if \(\Gamma_0 \vdash \text{UDD}_i : :_c \Gamma_i\) 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 \(\vdash \text{ASP} : :_c \text{USP}\) implies \(\vdash \text{ASP} : :_c \text{USP}\).

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

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

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 \(\vdash \text{ASP} \not\models \Box\), 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.

Let us first notice that in some cases, if ASP is an architectural specification with result unit expression UE, the model class of \(S_{\text{ASP}}(UE)\) 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 \(\vdash \text{ASP} : :_c \text{USP}\) then \(\text{ProjRes}(\text{Mod}(\text{ASP})) = \text{Mod}(\text{USP})\).

**Theorem 5.5.** Under the requirements of Framework 5.2 if \(\vdash \text{ASP} : :_c \text{USP}\) for some USP, then \(\vdash \text{ASP} : :_c \text{USP}'\) where \(\text{USP}' = S_{\text{ASP}}\) is the specification of the result unit expression UE of ASP and moreover \(\text{USP} \Rightarrow \text{USP}'\).
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 \( \vdash \text{ASP} \Rightarrow g \text{AM} \) for some \( \text{AM} \) such that for all \((U, BM) \in \text{AM}, U \in \text{Unit[USP]}\) for some USP then \( \vdash \text{ASP} :: \text{USP}' \), where \( \text{USP}' = S_{\text{ASP}} \) is the specification of the result unit expression \( \text{UE} \) of \( \text{ASP} \) and moreover \( \text{USP} \leadsto \text{USP}' \).

### 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:

```plaintext
arch spec ASP_NAME =
  units M_1 : SP_1
  ...
  M_n : SP_n;
  UN : SP \rightarrow SP' given M_1, \ldots, 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:

```plaintext
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 \rightarrow SP'
    result lambda X : SP \cdot 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 \( \text{UN} : \text{SP given UT} \) in a verification context \( \Gamma_v \), resulting from the analysis of previous declarations, can be written as \( \text{UN} : \text{arch spec [units F : S}_{\Gamma_v}((UT) \rightarrow SP; \text{result F[UT]})} \).

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 \( \text{UN} \) (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 \( M_1, \ldots, M_n \) visible in the anonymous architectural specification of \( \text{UN} \) 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 equivalent.

\(^{17}\)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 $(UΣ, BstC)$, 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 $\{ UN \rightarrow \{ F \mapsto R' \} \}$

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 $\{ UN \rightarrow R' \}$

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 $RΣ'$ of $R'$ into a component refinement signature mapping the name of the generated generic unit (in our case $F$) to $RΣ'$ 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:

- $RΣ_1 = (UΣ, BstC)$ with only one unit name $UN$ in the domain of $BstC$ and the composition $RΣ'$ of $BstC(UN)$ with $RΣ_2$ is defined. In this case, $RΣ_1 : RΣ_2 = (UΣ, BstC)[UN \mapsto RΣ']$.

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

- $RΣ_1 = (UΣ, BstC')$ and $BstC'$ has only $UN$ in the domain, and $RΣ' = BstC'(UN); RΣ_2$ is defined. In this case, $R_1 : R_2$ is defined as $R_1 ; [UN \mapsto R_2]$

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 $UN : arch spec \{ \text{units } UN : SP, \ldots \}$, 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 $RΣ$ be a refinement signature. A refinement specification $S$ over $RΣ$ is defined as follows:

A/B if $RΣ = (UΣ, BΣ)$, then $S$ takes the form $(USP, BSP)$, where $USP ⊃ UΣ$ and $BSP$ is a branching specification, which is in turn either a unit specification $USP'$ such that $USP' ⊃ UΣ$, or a map $SPM$ such that $\text{dom}(SPM) = \text{dom}(BstC)$ and $SPM(X)$ is a branching specification over $BstC(X)$, for each $X ∈ \text{dom}(BstC)$, when $BΣ = BstC$;

C if $RΣ = \{ UN_i \mapsto RΣ_i \}_{i ∈ J}$, then $S$ takes the form $\{ UN_i \mapsto S_i \}_{i ∈ J}$, where $S_i$ is a refinement specification over $RΣ_i$. 

36
The intuitive idea is that if a refinement specifies a constructor $\kappa: \text{Mod}(\Sigma) \rightarrow \text{Mod}(\Sigma')$, a refinement specification $(\text{USP}', \text{USP})$ consists of two unit specifications $\text{USP}$ and $\text{USP}'$ with $\vdash \text{USP} \supset \Sigma$ and $\vdash \text{USP}' \supset \Sigma'$ such that $\text{dom}(\kappa) = \text{Mod}(\text{USP})$ and $\text{cod}(\kappa) \subseteq \text{Mod}(\text{USP}')$. 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 $\text{SPM}_0$.

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

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

$S_1;S_2;B$ if $S_1 = (\text{USP}_1, \text{USP}_2)$, then $S_2 = (\text{USP}_3, \text{BSP})$ and $S_1;S_2 = (\text{USP}_1, \text{BSP})$, provided that $\text{USP}_2 \Rightarrow \text{USP}_3$.

$B:S$ if $S_1 = (\text{USP}_1, \text{SPM}_1)$ then $S_2$ must be of the form $(\text{USP}_1, S_1;S_2)$.

- if $\text{SPM}(A)$ is a unit specification $\text{USP}$, then $\text{SPM}(A)$ must be of the form $(\text{USP}', \text{BSP}')$. Then $\text{SPM}(A) = \text{BSP}'$, provided that $\text{USP} \Rightarrow \text{USP}'$.
- if $\text{SPM}(A) = \text{SPM}'$, then $\text{USP}(A)$ must be of the form $(\text{USP}'_1, S_1;S_2)$. Then $\text{SPM}(A) = \text{SPM}'(\text{USP}'_1) - \text{USP}'_1$.

$C:C$ if $S_1 = (\text{USP}_1, S_1;S_2)$, then $S_2$ is defined only if $S_2 = (\text{USP}_1, S_1;S_2)$. Then $S_1;S_2$ modifies the ill-defined union of $S_1$ and $S_2$ by putting $(S_1;S_2) = S_1(A);S_2(A)$ for $A \in \text{dom}(S_1) \cap \text{dom}(S_2)$.

The constructive proof calculus for architectural specifications is extended to the level of refinements as in Fig. 17. The following result states that if a statically well-formed refinement $\text{SPR}$ can be proven correct w.r.t. a refinement specification $S$ using the proof calculus for refinements, then $\text{SPR}$ has a denotation according to the model semantics and moreover the refinement relation thus obtained satisfies $S$. 

37
Theorem 5.7. [Soundness] Under the requirements of Framework 5.1, let \( \triangleright \) be a refinement such that \( \triangleright \). If \( \triangleright \), then \( \Rightarrow \) for some refinement relation \( \Rightarrow \) and moreover \( \Rightarrow \). □

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 \( \text{Monoid} \), \( \text{NatWithSuc} \), \( \text{Nat} \) and \( \text{NatBin} \), 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 \( \text{Monoid} \rightarrow \text{Nat} \). 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 \( (\text{Monoid}, \text{Nat}) \).

For R2, we use again the proof calculus rule for simple refinements and check that \( \text{Nat} \rightarrow \text{NatBin} \). The verification condition \( \text{Nat} \rightarrow \text{Nat} \) holds trivially and thus the composition of R1 and R2 is correct, with the resulting refinement specification \( (\text{Monoid}, \text{NatBin}) \).

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

\[
\text{arch spec AdditionFirst} = \\
\quad \text{units} \quad N : \text{Nat}; \\
\quad \text{M : arch spec} \{ \\
\quad \quad \text{units} \quad \Gamma : \text{Nat} \rightarrow \text{NatWithSuc} \\
\quad \quad \text{result} \quad \Gamma[N] \\
\quad \}\ \\
\quad \text{result} \quad M
\]

and the refinement specification of AdditionFirst is \( (\text{NatWithSuc}, [N \mapsto \text{Nat}, M \mapsto \{F \mapsto \text{Nat} \rightarrow \text{NatWithSuc}\}]) \).
Correctness of R4 is immediate because the condition NatWithSuc \(\rightarrow\) 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 \(\{N \mapsto (Nat, NatBin)\}\) 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 \(B(N) = Nat\); thus, the verification condition Nat \(\mapsto\) Nat again holds immediately. The refinement specification of R5 becomes (NatWithSuc, \(\{N \mapsto NatBin, M \mapsto \{F \mapsto Nat \mapsto NatWithSuc\}\}\)).

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

For Ref_Sbscs, the second rule in Fig. 17 must be applied (note that \(\sigma\) is the identity). This amounts to checking correctness of Arch_Sbscs and proving that

\[
\text{Sbscs}_\text{OPEN} \rightarrow S_{\text{Arch_Sbscs}}(UE)
\]

where \(UE = \lambda V : \text{VALUE} \cdot C[A[S[P[V]]]]\).

Since Arch_Sbscs contains only unit declarations, each unit is assigned its declared specification and \(S_{\text{Arch_Sbscs}}(UE)\) is obtained as follows:

- \(UE\) is a lambda expression, so \(S_{\text{Arch_Sbscs}}(UE) = \text{VALUE} \rightarrow \text{SP}\), where \(SP = S_{\text{Arch_Sbscs}}(C[A[S[P[V]]]]):\)

- since \(V : \text{VALUE} \rightarrow \text{PRELIMINARY}\), the verification condition for the unit application \(P[V]\) is \(\text{VALUE} \rightarrow \text{VALUE}\) which holds trivially. The specification of \(P[V]\) is \text{PRELIMINARY} (because \text{VALUE} is included in \text{PRELIMINARY});

- the other units are similar, and the last specification obtained is \text{Steam_Boiler_Control_System}.

We thus obtain \(S_{\text{Arch_Sbscs}}(UE) = \text{VALUE} \rightarrow \text{Steam_Boiler_Control_System}\), and since this is precisely Sbscs_OPEN, the refinement Ref_Sbscs is correct. The refinement specification of Ref_Sbscs is \((\text{Sbscs}_\text{OPEN}, \text{SPM})\) where \(\text{SPM}(P) = \text{VALUE} \rightarrow \text{PRELIMINARY}, \text{SPM}(S) = \text{PRELIMINARY} \rightarrow \text{Sbscs}_\text{STATE}, \text{SPM}(A) = \text{Sbscs}_\text{STATE} \rightarrow \text{Sbscs}_\text{ANALYSIS} \text{ and } \text{SPM}(C) = \text{Sbscs}_\text{ANALYSIS} \rightarrow \text{Steam_Boiler_Control_System}.

For Ref_Sbscs’, the third rule in Fig. 17 is applied. We have just checked correctness of Ref_Sbscs and obtained its refinement specification \((\text{Sbscs}_\text{OPEN}, \text{SPM})\). Therefore, we only have to check correctness of the component refinement following then and that the refinement specification obtained, which will be a map \(\text{SPM’}\), can be composed with \((\text{Sbscs}_\text{OPEN}, \text{SPM})\).

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 P, we must check correctness of State_Ref, which amounts to proving that models of \text{Sbscs}_\text{STATE_Impl} are indeed models of \text{Sbscs}_\text{STATE}. The refinement specification obtained is \((\text{State_Abstr}, \text{Unit_Sbscs}_\text{STATE})\). Note that the correctness proof is non-trivial in this case and needs to use a proof calculus for structured specifications, see Remark 5.1 at the end of Sect. 5.

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 \(S_{\text{Arch_Preliminary}}(MS) = \text{Messages_Sent}\) and \(S_{\text{Arch_Preliminary}}(MR) = \text{VALUE} \rightarrow \text{Messages_Received}\). The specification of the result unit expression (call it \(UE’\)) of Arch_Preliminary is of the form \(\text{VALUE} \rightarrow SP’\) 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 \(\text{SET}[\text{MS fit Elem} \rightarrow S_{\text{Message}}][V], \text{SET}[\text{MR}[V] \rightarrow R_{\text{Message}}][V]\) and \(\text{CST}[V]\) with the specification \(S_{\text{anal}}(\text{UT})\).

For the first term, the verification conditions are

\[
\text{Messages_Sent} \models \{\text{sort Elem} \rightarrow S_{\text{Message}}\}
\]

and \(\text{VALUE} \rightarrow \text{VALUE}\). Both hold immediately and the specification of the term is

\(\text{(Set[sort Elem] with Elem} \rightarrow S_{\text{Message}}\) and \text{Messages_Sent}\)

39
For the second term, we similarly obtain the specification

\[
\text{(Set[\text{sort Elem} \text{ with } \text{Elem} \rightarrow R\text{Message} \text{ and } \text{Messages}_{\text{received}}})
\]

For the third term, notice that \( V : \text{VALUE} \) and the condition \( \text{VALUE} \sim \rightarrow \text{VALUE} \) holds immediately. Because the specification \( \text{Srcs\_Constants} \) extends \( \text{VALUE} \), the specification obtained is just \( \text{Srcs\_ Constants} \).

Finally, for the unit \( A \), we need to check correctness of the architectural specification \( \text{Arch\_Analysis} \). 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 (\( \text{Srcs\_Open} \), \( \text{SPM} \)) and \( \text{SPM}' \) compose. Notice that the domain of \( \text{SPM}' \) is included in the domain of \( \text{SPM} \). For the unit \( S \), the verification condition is immediate and the corresponding specification of \( S \) is updated to \( \text{Unit\_Srcs\_State} \). For the unit \( A \), the verification condition holds by noticing that \( \text{PRELIMINARY} \) is equivalent to the specification \( \text{USP} \) obtained in the specification of the result unit of \( \text{Arch\_Preliminary} \). \( \text{SPM} \) is then updated in \( A \) to the map taking each of the units of \( \text{Arch\_PRELIMINARY} \) to its specification.

\[ \Box \]

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

\[
\text{spec } \text{Sig} = \text{sort } s
\]

\[
\text{ops } a, b : s
\]

\[
\text{spec } \text{EQ} = \text{Sig then } \{ \ast a = b \}
\]

\[
\text{refinement } \text{Incl} = \text{Sig refined to } \text{EQ}
\]

\[
\text{arch spec } \text{ASP\_NAME} = \text{units } M : \text{Sig result } M
\]

\[
\text{refinement } \text{ASP\_EQ} = \text{arch spec } \text{ASP\_NAME then } \{ M \to \text{Incl} \}
\]

\[
\text{refinement } \text{Ref\_EQ} = \text{EQ refined to } \text{ASP\_EQ}
\]

40
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 Ref_Eq, then the tree of ASP_Eq "grows" towards the root, via the refinement R ⊸ Eq.

We have that ⊢ ASP_NAME :::c (ΣG, [M ⊸ ΣG]) and ⊢ [M to Incl] :::c {M ⊸ (ΣG, Eq)}. Thus, ⊢ ASP_Eq :::c (ΣG, [M ⊸ Eq]) and the verification condition of Ref_Eq is Eq ⊸ ΣG, 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, SASP captures exactly (under some conditions) the models produced by the architectural specification. The idea is that rather than defining SASP 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 ✓ R ⇒ R where R = [(U)_x, U] U ∈ Unit(USP').

The same constructor as the one specified by R, namely Mod(σ): Unit(USP') → (Unit(USP') hide σ), can be equivalently specified by the architectural specification

\[
\text{arch spec ASP_NAME =} \quad \text{units UN : USP'} \quad \text{result UN hide σ}
\]

Given that ✓ ASP_NAME ⇒ [(U)_x, (UN ⊸ U)] U ∈ Unit(USP')], we get a one-to-one correspondence between the model class of R on one hand, consisting of all assignments (U)_x, U), with U ∈ Unit(USP'), and the model class of ASP_NAME on the other hand, consisting of all (U)_x, (UN ⊸ U)), with U ∈ Unit(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 VS:

\[
\forall VS := ASP \mid \{ UN \mapsto \forall VS \}_{i \in J}
\]

where ASP is an architectural specification, and we call enhanced verification specifications of the form \{ UN \mapsto \forall VS \}_{i \in J} enhanced verification maps. The idea is that if VS is the enhanced verification specification of a refinement SPR, VS specifies the constructor(s) induced by SPR. Moreover, we define a partial operation of composition between enhanced verification specifications, denoted \forall VS_1;\forall VS_2, as follows:

\[S;S:B \text{ if } \forall VS_1 = ASP_1 \text{ such that } ASP_1 \text{ has no branching and } \forall VS_2 \text{ is also an architectural specification } ASP_2, \text{ then } \forall VS_1;\forall VS_2 \text{ replaces the specification of the leaf in } ASP_1 \text{ with } ASP_2; \]

\[B:C \text{ if } \forall VS_1 = ASP_1 \text{ and } \forall VS_2 \text{ is an enhanced verification map, then } \forall VS_1;\forall VS_2 \text{ is defined iff for each } UN \in dom(\forall VS_2), UN \text{ is in } dom(\forall VS_1) \text{ such that:}
\]

---

16 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.
if the specification of $\text{UN}$ in $\text{ASP}_1$ is a unit specification $\text{USP}$, then $\text{VS}_2(\text{UN})$ must be an architectural specification $\text{ASP}_2$ and $\text{USP} \Rightarrow S_{\text{ASP}_2}$. Then we update the specification of $\text{UN}$ in $\text{ASP}_1$ to $\text{ASP}_2$;

- if the specification of $\text{UN}$ in $\text{ASP}_2$ is an architectural specification $\text{ASP}'$, then $\text{VS}_2(\text{UN})$ must be an enhanced verification map $\text{VS}'$ such that $\text{ASP};\text{VS}'$ is defined. Then we update the specification of $\text{UN}$ in $\text{ASP}$ to $\text{ASP};\text{VS}'$;

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

Models of enhanced verification specifications are obvious generalizations of models of architectural specifications: if an enhanced verification specification $\text{VS}$ is an architectural specification $\text{ASP}$, then the model of $\text{VS}$ is just the architectural model $\mathcal{A}\mathcal{M}$ such that $\vdash \text{ASP} \Rightarrow \mathcal{A}\mathcal{M}$, while if $\text{VS}$ is an enhanced verification map $\text{USP}_1 \Rightarrow S_{\text{ASP}_2}$ and $\mathcal{M}_i$ is a model for $\text{VS}_i$, for each $i \in J$, then $\text{USP}_1 \Rightarrow \mathcal{M}_i \in J$ is a model for $\text{VS}$. By a slight abuse of notation, we denote the models of enhanced verification specifications also by $\mathcal{A}\mathcal{M}$, as in the case of architectural specifications, and we write $\vdash \text{VS} \Rightarrow \mathcal{A}\mathcal{M}$ to denote that $\mathcal{A}\mathcal{M}$ is the model of $\text{VS}$.

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

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

**Theorem 5.9.** [Completeness of the enhanced calculus] Let $\text{SPR}$ be a refinement such that $\vdash \text{SPR} \Rightarrow \Box$ and the assumptions of Framework \[5.2\] hold for all architectural specifications appearing in $\text{SPR}$. If $\vdash \text{SPR} \Rightarrow \mathcal{R}$ then $\vdash \text{SPR} \vdash_r \text{VS}$ for some enhanced verification specification $\text{VS}$ and $\vdash \text{VS} \Rightarrow \mathcal{A}\mathcal{M}$ for some $\mathcal{A}\mathcal{M}$ such that $\mathcal{R}$ and $\mathcal{A}\mathcal{M}$ correspond up to unit names.
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 ⊢ SPR ⇒ R, then 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 ⊢ 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 SPR¹ then SPR², if SPR¹ contains a branching, it does not suffice to check consistency of SPR² (which must be a component refinement), because some component of SPR¹ outside the domain of SPR² might be inconsistent. On the other hand, if SPR¹ does not contain branchings and the composition SPR¹; SPR² 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 ⊢ SPR ≔c □ and ⊢ 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 \( \vdash SPR ::_c \Box \) and \( SPR \) is non-empty, then \( \vdash \text{cons}(SPR) \).

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 \( \text{http://hets.eu} \).

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 CSL 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 \( \text{http://www.omg.org/spec/DOL/} \) and \( \text{http://www.dol-omg.org} \).

\(^{20}\) 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 \( R = USP \text{ refined via } \sigma \text{ to } USP' \), 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 \( \sigma \). 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. \( \text{Ref}_{\text{SBCS}} \) 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 \( \text{Ref}_{\text{SBCS}} \), we first build the trees of the architectural specifications \( \text{Arch}_{\text{Failure Detection}} \) and \( \text{Arch}_{\text{Prediction}} \). These trees must then be connected with the refinement tree of \( \text{Ref}_{\text{SBCS}} \) 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 \( RT \) 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.
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 $RT$ as either

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

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.
Definition 6.2. Given two refinement trees presented in Fig. 22.

- The pointer of the specification MonoM is (MonoM, MonoM): the refinement tree consists of just one node, which is at the same time the root and the leaf.
- The pointer of R1 is (MonoM, Nat): the refinement tree consists of just one edge, with MonoM at the root and Nat at the leaf. Similarly the pointer of R2 is (Nat, NatBin).
- The pointer of AdditionFirst is (AdditionFirst, {F \mapsto (F,F), N \mapsto (N,N)}): the root of the tree corresponds to the result unit of AdditionFirst, 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, $\mathcal{RT}[\text{USP}]$ is obtained by adding to it a new isolated node $n$ labelled with USP. For refinement trees $\mathcal{RT}_1, \ldots, \mathcal{RT}_k$, $\mathcal{RT}[n_1 \mapsto \mathcal{RT}_1, \ldots, \mathcal{RT}_k]$ denotes the tree obtained by inserting component links from the node $n_i$ 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 $\mathcal{RT}_1 \circ p_1, p_2 \mathcal{RT}_2$, 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 $(n_1, n_2)$.

- **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 $\text{dom}(f_2)$ 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 $\text{dom}(f_2)$. 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 $\text{dom}(f_2)$ 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 $\text{dom}(f_2)$. 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.
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 \((\text{Monoid}, \text{Nat})\) and the pointer \( p_2 \) of R2 is \((\text{Nat}, \text{NatBin})\). The pointer of the result of their composition is \((\text{Monoid}, \text{NatBin})\).

The refinement tree of R4 is obtained by composing the tree of the unit specification \( \text{NatWrmSuc} \) with a single node and with pointer \( q_1 = (\text{NatWithSuc}, \text{NatWithSuc}) \) with the tree of the architectural specification \( \text{AnronovFirst} \) (case S:B). The pointer of the latter tree is \( q_2 = \text{AdditionFirst}, \{ F \mapsto (F, F), N \mapsto (N, N) \}\). After composition, the pointer \( q_3 \) is \((\text{NatWithSuc}, \{ F \mapsto (F, F), N \mapsto (N, N) \}\) (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 \( \text{ArchSbcs} \) consists of one node labeled S, and thus its pointer is \((S, S)\). This tree is composed with the refinement tree of \( \text{StateRef} \), which is \((\text{StateAbstr}, \text{UnitSbcsState})\), and the result is \((S, \text{UnitSbcsState})\).

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

**Refinement**

\[
\begin{align*}
\text{R} &= \{ \text{P to arch spec ArchPreliminary, S to StateRef, A to arch spec ArchAnalysis } \\
& \text{then } \{ \text{A to [FD to arch spec ArchFailureDetection, PR to arch spec ArchPrediction } \}}
\end{align*}
\]

and the resulting refinement tree is a forest with three trees: one for P, consisting of the subtree with the root in \( \text{ArchPreliminary} \) in Fig. 20, one for A, consisting of the subtree with the root in \( \text{ArchAnalysis} \) in Fig. 20 and one for S, consisting of the tree of \( \text{StateRef} \), 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
of the model finders available in the tool, e.g. Isabelle-refute [41], Darwin [42], or SPASS [43], 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
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. 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 $\kappa$ in a constructor implementation $\text{arch spec}$

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

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. 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 $\kappa$ in a constructor implementation $\text{arch spec}$

\[
\begin{align*}
\text{arch spec} & \quad \text{unit } \kappa : [ \text{SP'} \to \text{SP} ] \text{ refined to USP} \\
& \quad \text{result } \lambda X : \text{SP'} \bullet \kappa[X]
\end{align*}
\]

where USP is a monomorphic specification of $\kappa$ 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

\[
\text{SP } \xrightarrow{\kappa} \text{SP'} \xrightarrow{\kappa'} \text{SP''}
\]

\[24\] In case of non-termination, the (partial) function is not specified uniquely. Consider e.g. the specification $\forall x. f(x) = f(x)$. 50
is expressed as

\[
\text{refinement } R_5 = \\
\text{SP refined to arch spec} \\
\text{units } \kappa : \{ \text{SP' } \rightarrow \text{SP} \} \text{ refined to USP} \\
A' : \text{SP' refined to arch spec} \\
\text{units } \kappa' : \{ \text{SP'' } \rightarrow \text{SP'} \} \text{ refined to USP'} \\
A'' : \text{SP''} \\
\text{result } \kappa'[A''] \\
\]

where USP and USP' are monomorphic specifications of \( \kappa \) and \( \kappa' \), 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).


URL [http://www.cofi.info/old/Notes/L-9/](http://www.cofi.info/old/Notes/L-9/)


53
Appendix A. Proofs

Proof of Theorem 5.2
Assume that arch spec ASP = units UDO* result UE. Moreover, assume $\Gamma_0^{\text{gen}} \vdash \text{UDO}^* \vdash \Gamma_i$ and $\text{UDO}^* \vdash \Gamma_{\text{gen}}, \Gamma_i$. We prove that if $\text{USP} = S_{ASP}$ then $\Gamma_{\text{gen}}, \Gamma \vdash UE :: \text{USP}$.

- If $UE = UT$, it suffices to show that for any family of models $\mathcal{M}$ compatible with the diagram $\Gamma'$, $M_B \models S_{ASP}(UT)$, where $\Gamma_{\text{gen}}, \Gamma \vdash UT :: \Gamma'$, $B$. We will proceed by induction on the structure of $UT$. Notice that in the case of unit application and union, $M_B \models S_{\text{analy}}(B)$ using the amalgamation properties ensured by the successful run of the extended static semantics.

- If $UT = UN$, then by definition $S_{ASP}(UT) = SP$ where $UN : SP$ in ASP and by construction of $\Gamma'$, the node $UN$ is labeled with $SP$. This means that $M_{UN}$ must satisfy $SP = S_{ASP}(UT)$.

- If $UT = UT_1$ and $UT_2$, then by construction $B$ has two incoming edges from $A_1$ and $A_2$, where $\Gamma_{\text{gen}}, \Gamma \vdash UT_i :: \Gamma_i, A_i, i = 1, 2$. By the inductive hypothesis $M_{A_i} \models S_{ASP}(UT_i)$. It follows easily that $M_B \models S_{ASP}(UT_1)$ and $S_{ASP}(UT_2)$.

- If $UT = \text{fit} \sigma$, then, using the notation of the rule, $B$ has two incoming edges from $A_r$ and $A_n$ by construction. $A_r$ is labeled with $SP_r$, and $M_{A_r} \models S_{ASP}(UT')$ by inductive hypothesis. It is easy to check that $M_B \models S_{ASP}(UT)$.

- If $UT = UT' \text{ hide} \sigma$, then, using the notation of the rule, $B$ has an outgoing edge to $A$ by construction and $M_A \models S_{ASP}(UT')$, which implies $M_B \models S_{ASP}(UT)$.

- If $UT = \text{local uDecl within UT'}$, then we must show that $M_B \models S_{ASP}(UT)$ which by definition is $S_{\text{UE}}(UT')$ where $\Gamma_i$ 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 $M_B \models S_{\text{UE}}(UT')$.

- If $UE = AX : SP . UT$, we have $M_B \models S_{ASP}(UT)$ where $\Gamma_{\text{gen}}, \Gamma \vdash UT :: \Gamma', B$ and since $SP$ is obviously equivalent with itself, we have that $\Gamma_{\text{gen}}, \Gamma \vdash UE :: SP \rightarrow S_{ASP}(UT)$.

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 $\vdash ASP \triangleright \Box$ and $\vdash ASP :: USP$.

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 $S_{ASP}(oT)$, 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 $M_{oT'} := M_{1|σ_1}$, where $σ_1$ is the inclusion of the signature of $T_1$ into that of $T$. From $M_T ∈ Mod(S_{ASP}(T))$, we easily get $M_{oT'} ∈ Mod(S_{ASP}(T))$;

• if $T$ is $T_1$ with $σ$, let $M_{oT'} := M_{1|σ}$. Again, from $M_T ∈ Mod(S_{ASP}(T))$, we easily get $M_{oT'} ∈ Mod(S_{ASP}(T))$;

• if $T$ is $T_1$ hide $σ$, $S_{ASP}(T)$ is $S_{ASP}(T_1)$ hide $σ$. Hence, by $M_T ∈ Mod(S_{ASP}(T))$, there is some (not necessarily unique) $M_1 ∈ Mod(S_{ASP}(T_1))$ with $M_1|σ = M_T$. Put $M_{oT'} := M_1$;

• if $T = F[T_1 \text{ fit } σ_1] \ldots [T_n \text{ fit } σ_n]$, then we know that $S_{ASP}(T) = \{ SP \text{ with } σ \} \text{ and } S_{ASP}(UT_i) \text{ with } i = 1; \ldots; n \text{ and } S_{ASP}(UT_n) \text{ with } i, i' \text{ and } S_{ASP}(UT_n)$ with $i, i'$ and $S_{ASP}(UT_n)$ with $ι$. Hence, for $i = 1, \ldots, n$, we define $M_{oT'} := M_{1|ι'}$. We also define the action of $F$ on the diagrams thus defined: $F_M(M_{oT'}|ι, \ldots, M_{oT'}|ι') := M_{oT'}$. (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 $F_M$, defining $F$ with $F_{FM} := F_T$.

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 $M_T$ 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 $UT$, and $UT_j$ respectively, for some $i, j = 1, \ldots, n$ or a unit amalgamation $UT_1 \ post \ and \ UT_2$ with $oT_1, oT_2$ subterms of $UT_1$ and $UT_2$ respectively, for some $i, j = 1, \ldots, n$. In both cases, by Def. 4.3, we have that $M_{oT'} \models S_{alt}(UT')$. 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:

![Diagram](image)

where the nodes labeled with $σ_1(Δ)$ correspond to the two occurrences $oT_1$ and $oT_2$, $Δ : σ_f ∈ Δ_1$ 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 $σ_1(Δ), σ(Δ), i$ is the selected pushout of the span (cf. Def. 4.4). Then the symbols of $Δ$ that are not in $Δ_T$ do not have a common origin in the diagram, and therefore their image along $σ_1(Δ)$ gives symbols that do not necessarily 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 $ASP \models \text{ units } UDD^+ \text{ result } UE$. Moreover, assume $+ UDD^+ :: Γ_{gen}, Γ$. 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 $Γ_{gen}, Γ ⊢ UE' :: UDD'$ for some $USP'$, then $Γ_{gen}, Γ ⊢ UDD^+ :: Γ_{gen}, Γ, + UE' :: UDD' :: UDD' :: USP' where USP' = S_{UT_1(UE')}$ and $USP' \models UDD'$. 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 \( \Gamma_{\text{gen}} \vdash UN :: USP \) for some \( USP \). By definition of \( S_{\text{ASP}}(UN) \), we have that \( \Gamma_i \vdash UN :: USP' \). We have to show that \( USP \Rightarrow USP' \). Let \( ASP' \) be the architectural specification obtained from \( ASP \) by replacing its result unit expression with \( UN \). Since \( \vdash ASP' :: USP' \), by Thm. 5.4 we have \( \text{ProjRes}(\text{Mod}(ASP')) = \text{Mod}(USP') \). By \( \vdash ASP' :: USP \) and soundness of \( \vdash :: :: \) we have \( \text{ProjRes}(\text{Mod}(ASP')) \subseteq \text{Mod}(USP) \). Hence, \( \text{Mod}(USP') \subseteq \text{Mod}(USP) \).

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 \( F : SP \Rightarrow SP' \) in \( \Gamma_{\text{gen}} \). Since \( \Gamma_{\text{gen}} \vdash UE :: USP \) for some \( USP \), we can easily show that \( \Gamma_{\text{gen}} \vdash UE' :: SP \) (this follows directly from the verification condition of the proof calculus \( \vdash :: :: \) for unit applications). By the inductive hypothesis, we get that \( \Gamma_i \vdash UE' :: S_{\text{ASP}}(UE') \) and moreover \( SP \Rightarrow S_{\text{ASP}}(UE) \), which is precisely the verification condition of the proof calculus \( \vdash :: :: \) for \( F[UE] \). This means that \( \Gamma_i \vdash UE :: S_{\text{ASP}}(UE) \), where \( S_{\text{ASP}}(UE) \) is as given in Def. 5.3. We need to show that \( USP \Rightarrow S_{\text{ASP}}(UE) \), and this is done with the same argument as for the case of unit names.

---

\textbf{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 \( \vdash SPR \Rightarrow R \) where \( R = \{(U,U)|U \in \text{Unit}(USP)\} \). With the proof calculus rule we have \( \vdash SPR ::c (USP,USP) \) and it is obvious that \( R \models (USP,USP) \).

If \( SPR = USP \) refined via \( b \) to \( SPR' \) with \( \vdash SPR :: S \), then according to the proof calculus rule we have that \( \vdash SPR' ::c (USP',BSP), USP' \Rightarrow USP \) and \( S = (USP,BSP) \). Then \( \vdash SPR \Rightarrow R \) where \( R = \{(U_i,r_i,BM)|U_i,BM \in R \} \) and the refinement condition \( USP \Rightarrow USP' \) ensures that \( R \models S \).

If \( SPR = \text{arch spec} ASP \), then according to the proof calculus rule we have that for any unit declaration of the form \( UN_i : SPR_i \) in \( \text{asp} \), \( \vdash SPR_i ::c (USP_i,BSP_i) \). 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 \( \text{UE} \) denote the result unit of \( ASP \). Then we define \( S_{\text{PM}}(UN) \) for any unit \( UN \) of \( ASP \) as \( S_{\text{PM}}(UN) = BSP_i \) if \( UN \) is a declared unit \( UN_i \) of \( ASP \), and \( S_{\text{PM}}(UN) = S_{\text{UE}} \) if \( UN = UE \) is a unit definition of \( ASP \), where \( \text{UE} \) is the verification environment of \( ASP' \). This allows us to define \( S = \{S_{\text{PM}}(UN)\} \). By the inductive hypothesis we have \( \vdash SPR_i \Rightarrow R_i \) and \( R_i \models (USP_i,BSP_i) \) for each \( i \).

Then \( \vdash SPR \Rightarrow R \) where \( R = \{(U_i,r_i, BM_i)|U_i, BM \in R \} \) and \( \text{UE} \) combines the units in \( \pi_i(RE) \) according to \( UE \) and \( R \models S \) by Thm. 5.3 where \( \text{AM} = R \) and \( \vdash ASP ::c S_{\text{UE}} \) by the rules of the proof calculus (Fig. 15).

If \( SPR = \{UN_i \to SPR_i\}_{i \in J} \) with \( \vdash SPR :: S \), then according to the proof calculus rule we have that \( S = \{UN_i \to S_i\}_{i \in J} \) and \( \vdash SPR_i ::c S_i \) for \( i \in J \). By the inductive hypothesis we get that \( \vdash SPR_i \Rightarrow R_i \) and \( R_i \models S_i \). Then \( \vdash SPR \Rightarrow R \) where \( R = \{(U_i,r_i(RE_i))|U_i, R_i \in R_i \} \) and notice that \( R \models S \) by definition.

Finally, if \( SPR = SPR_i \text{ then } SPR_2 \), 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 \( \vdash SPR_i \Rightarrow R_i \) and \( R_i \models S_i \) for \( i = 1,2 \). It follows (by case analysis, sketched below) that \( R = R_1;R_2 \text{ is defined and } R \models S \).

Let us assume that \( \vdash SPR_1 \triangleright (U_i,B_i) \); the other cases are similar. Then \( S_1 = \{USP_i,SPM_i\}, \vdash SPR_2 \triangleright \{UN_i \to R_{\Sigma} \}_{i \in J} \) with \( UN_i \in \text{dom}(B_i) \) for \( i \in J \) and \( S_2 = \{UN_i \to S_i\}_{i \in J} \). Let \( R_2 = \{UN_i \to R_i\}_{i \in J} \subseteq R_2 \) and assume there exist \( R_1 = \{U_i, BE \} \subseteq R_1 \). If \( R_1 \) were empty, the result follows by noticing that \( R = R_1;R_2 \text{ is empty as well} \) and that \( R \models S_1;S_2 \); and by distinguishing the subcases when an inconsistent specification in \( S_{\text{PM}} \) 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, that \( \text{PM} \) corresponds to a leaf in the branching tree. This means that \( \text{dom}(R_2) = \{UN\} \) and moreover \( \text{Bst}(UN) = U_{\Sigma} \). Then \( S_{\text{PM}}(UN) = USP \) and \( S_2(UN) = \{USP_i,BSP_i\} \). Since \( R_2 \models S_2 \) we have that \( R_2(U_i) = \{(U_i',BM')\} \) such that \( U' \in \text{Unit}(USP) \). Since \( S_1;S_2 \text{ is defined} \), the associated verification condition
we have $\vdash R$ up to unit names, so do $\vdash \text{units}$. Since the choice of models for $\text{ASP}$ each $= i$, we get that to the model class of $\text{units}$. The other cases are similar but tedious and we omit them.

Proof of Theorem 5.8 By induction on the structure of $\text{SPR}$.

If $\text{SPR} = \text{ASP}$, then $\vdash \text{ASP} :: \text{units}$ where $\text{arch spec ASP} \Rightarrow \text{SPR} \Rightarrow \text{USP}$ for any $\text{arch spec ASP} = \text{arch spec ASP'}$ and $\text{arch spec ASP} = \text{arch spec ASP''}$, where $\text{arch spec ASP''} = \text{arch spec ASP' result}$ (UN hide $\sigma$), as in the rule for simple refinements. By the induction hypothesis we get $\vdash \text{SPR} \Rightarrow \text{R'}$ and $\text{R'}$ and $\text{AM'}$ correspond up to unit names, where $\vdash \text{ASP} \Rightarrow \text{AM}$, where $\text{AM} = \{(U, (U \mapsto U)) | U \in \text{Unit(USP)}\}$. It is obvious that $\text{AM}$ and $\text{R}$ correspond up to unit names.

If $\text{SPR} = \text{arch spec ASP}$, then by the hypothesis we get $\vdash \text{SPR} \Rightarrow \text{UN}$, 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 $\text{R}$ was arbitrary, we get this for all possible refined-unit static contexts, and thus for all models of $\text{ASP}$ and $\text{ASP'}$. Then the models of $\text{ASP}$ and $\text{ASP''}$ correspond up to unit names.

Proof of Theorem 5.9 By induction on the structure of $\text{SPR}$.

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

If $\text{SPR} = \text{USP refined via } \sigma$ to $\text{SPR'}$ and we know that $\vdash \text{SPR} \Rightarrow \text{R}$, then from the rules of the model semantics we get that $\vdash \text{SPR} \Rightarrow \text{R'}$ and for every $(U, BM) \in \text{R'}$, $U \in \text{Unit(USP)}$. By the inductive hypothesis we have that $\vdash \text{SPR'} :: \text{ASP}$ and there is a one-to-one correspondence between $\text{R'}$ and $\text{AM'}$, where $\vdash \text{ASP} \Rightarrow \text{AM'}$. We need to prove that $\vdash \text{ASP} \Rightarrow \text{SPR}$.
and we know that \((V, BM)\) corresponds up to unit names with an assignment of \(\mathcal{R}'\), which gives us that \(V|_{\nu} \models USP\). Thus \(\vdash SPR \models ASP\). 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 \(\mathcal{A}\) 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 \(\vdash SPR_i \models ASP\) and model class of ASP\(_i\) correspond up to unit names with \(\mathcal{R}_i\). It follows that \(\vdash SPR \models ASP\) and the model of ASP\(_i\) corresponds up to unit names with \(\mathcal{R}\) with the same argument in the soundness proof.

If \(SPR = \{Un_i to SPR_i\}_{i \in J}\) 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 \(\vdash \mathcal{R}_i \models \mathcal{VS}_i\) and with the proof calculus rule we get \(\vdash \mathcal{R} \models \mathcal{VS} = \{Un_i to \mathcal{VS}_i\}_{i \in J}\). 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 \(i \in J\).

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 \(\vdash SPR \models \mathcal{VS}\) 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 \(Un\) is architectural reduces to this case) and \(\mathcal{VS}_2(\mathcal{USP})\). Let \(Un \models USP\) and then by Thm. \ref{thm:archspec} we know that there is a \(BM\) such that \((U, BM)\) is an architectural model of \(USP\). 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 \(BM'(\mathcal{USP}) = U\). 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 \(VS\) but also a unit environment for \(\mathcal{VS}_1\) to verify.\(\mathcal{VS}\) 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 \(\mathcal{R}_2\).

\(\boxtimes\)

Proof of Theorem \ref{thm:archspec}

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 \(\sigma\) to \(SPR'\), by definition, \(\vdash cons(\mathcal{SPR})\) holds if \(\vdash cons(\mathcal{SPR}')\) holds. By the induction hypothesis, the refinement relation \(\mathcal{R}'\) of \(SPR'\) is non-empty. Since \(\vdash SPR \models \mathcal{R}'\), it follows that \(USP \sim \mathcal{USP}'\), where \(\vdash SPR \models \mathcal{USP}'\). This means that \(Un|_{\nu} \in Unit(USP)\) for any \((U, BM)\) in \(\mathcal{R}'\). It follows that the refinement relation \(\mathcal{R} = \{(Un|_{\nu}, BM) | (U, BM) \in \mathcal{R}'\}\) of \(SPR\) is non-empty.

If \(SPR = \{Un_i to SPR_i\}_{i \in J}\), by the hypothesis we get that \(\vdash cons(\mathcal{SPR})\) and \(\vdash SPR_i \models \mathcal{VS}_i\) for each \(i \in J\). By the induction hypothesis, for each \(i \in J\), 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. \ref{fig:archspec} 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 \(\vdash cons(\mathcal{SPR}_1)\) and \(\vdash SPR_1 \models \mathcal{R}_1\), 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 \(\vdash SPR \models \mathcal{R}_1\) and \(SPR_1\) and \(SPR_2\) are consistent, using the soundness of the proof calculus we have that \(R_1; R_2\) is defined and \(\vdash SPR \Rightarrow \mathcal{R}_1; \mathcal{R}_2\). Since the composition \(R_1; R_2\) is defined, there must be a model \(R' = (U', BE[R_2]) \in \mathcal{R}_1\), where \(R_1 = (U, BE)\) and the definition of \(BE[R_2]\) has been given in the second main case of Def. \ref{def:archspec}. Using the definition of the composition, we get that \((U', BE[R_2])\) (as defined in the second main case of Def. \ref{def:archspec}) 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 \(\vdash cons(\mathcal{SPR}_2)\) and \(\vdash SPR_2 \models \mathcal{R}_2\), 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 \(\vdash SPR_1 \supset \{(U_2, U_2')\}\) and \(\vdash SPR_2 \supset \{(U_2', BE)\}\). Then \(\vdash SPR \models \mathcal{R}_1\) and \(\vdash SPR \models \mathcal{R}_2\), where USP is USP and moreover USP is USP. This means that if \((U, BM)\) is an assignment in \(\mathcal{R}_2\), \(U\) is not only a USB-model but also a USP-model and since \(\mathcal{R}_1 \supset \{(USP, USP')\}\), the constructor \(\kappa\) associated with \(SPR_1\) is defined on \(U\). We get thus the assignment \((\kappa(U), BM)\) in \(\mathcal{R}_1; \mathcal{R}_2\).
If \( \text{SPR} = \text{arch spec ASP} \), by the induction hypothesis, \( \text{SPR}_i \) has a model, for all \( UN_i : \text{SPR}_i \) in \( \text{ASP} \). This gives us a unit environment for \( \text{ASP} \) by projecting each of the models of \( \text{SPR}_i \) to the first component and we can combine the units in the way described by the result unit of \( \text{ASP} \) to get a model of \( \text{ASP} \) because \( \text{ASP} \) is statically correct.

\[ \square \]

**Proof of Theorem 5.11**

By structural induction on \( \text{SPR} \).

If \( \text{SPR} = \text{USP} \), the result follows from the completeness of the calculus for structured specifications and conservativity of extensions of structured specifications.

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

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

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

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

\[ \square \]
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
  op __+_ : Value × Value → Value, assoc, comm, unit 0;
  op __-_ : Value × Value → Value;
  op __* : Value × Value → Value, assoc, comm, unit 1;
  op __/2, __^2 : Value → Value;
  pred __<_ , __<=_ : 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 ::= 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 x Set[R_Message];
 PU_OK : State x Set[R_Message] x PhysicalUnit;
 DangerousWaterLevel : State x Set[R_Message]]
given Sbcs_STATE_1 =
local

% % Auxiliary predicates to structure the specification of next_mode.
preds Everything_OK, AskedToStop, SystemStillControllable,
   Emergency : State x 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)
  ∧ ¬ SystemStillControllable(s, msgs)
  ⇒ next_mode(s, msgs) = Degraded

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

\[ \Rightarrow \text{next\_status}(s, \text{msgs}, \text{pu}) = \text{FailureWithoutAck} \]

- \( \text{status}(s, \text{pu}) = \text{FailureWithAck} \land \text{REPAIRED}(\text{pu}) \text{ eps } \text{msgs} \Rightarrow \text{next\_status}(s, \text{msgs}, \text{pu}) = \text{OK} \)

- \( \text{status}(s, \text{pu}) = \text{FailureWithAck} \land \sim \text{REPAIRED}(\text{pu}) \text{ eps } \text{msgs} \Rightarrow \text{next\_status}(s, \text{msgs}, \text{pu}) = \text{FailureWithAck} \)

end

spec \text{Message\_Transmission\_System\_Failure} = \text{Sbc\_State\_2}
then local

%%% Static analysis:

\[ \text{pred is\_static\_OK : Set[R\_Message]} \]
\( \forall \text{msgs : Set[R\_Message]} \]
- \( \text{msgs\_is\_static\_OK} \)
  \( \Leftrightarrow \sim \text{junk} \text{ eps } \text{msgs} \land \exists! \text{ v : Value} \bullet \text{LEVEL}(\text{v}) \text{ eps } \text{msgs} \)
  \( \land (\exists! \text{ v : Value} \bullet \text{STEAM}(\text{v}) \text{ eps } \text{msgs}) \)
  \( \land (\forall \text{ pn : PumpNumber} \)
    \( \bullet \exists! \text{ ps : PumpState} \)
    \( \bullet \text{PUMP\_STATE}(\text{pn, ps}) \text{ eps } \text{msgs} \)
  \( \land (\forall \text{ pn : PumpNumber} \)
    \( \bullet \exists! \text{ pcs : PumpControllerState} \)
    \( \bullet \text{PUMP\_CONTROLLER\_STATE}(\text{pn, pcs}) \text{ eps } \text{msgs} \)
  \( \land \forall \text{ pu : PhysicalUnit} \)
- \( \sim \text{FAILURE\_ACKNOWLEDGEMENT}(\text{pu}) \text{ eps } \text{msgs} \land \text{REPAIRED}(\text{pu}) \text{ eps } \text{msgs} \)

%%% Dynamic analysis:

\[ \text{pred is\_NOT\_dynamic\_OK\_for\_} \]
\( \text{ : Set[R\_Message]} \times \text{State} \)
\( \forall \text{s : State; msgs : Set[R\_Message]} \)
- \( \text{msgs\_is\_NOT\_dynamic\_OK\_for\_s} \)
  \( \Leftrightarrow \sim \text{mode}(\text{s}) = \text{Initialization} \)
  \( \land (\text{STEAM\_BOILER\_WAITING} \text{ eps } \text{msgs} \)
    \( \lor \text{PHYSICAL\_UNITS\_READY} \text{ eps } \text{msgs}) \)
  \( \lor (\exists \text{ pu : PhysicalUnit} \)
    \( \bullet \text{FAILURE\_ACKNOWLEDGEMENT}(\text{pu}) \text{ eps } \text{msgs} \)
    \( \land (\text{status}(\text{s, pu}) = \text{OK} \)
      \( \lor \text{status}(\text{s, pu}) = \text{FailureWithAck}) \)
  \( \lor (\exists \text{ pu : PhysicalUnit} \)
    \( \bullet \text{REPAIRED}(\text{pu}) \text{ eps } \text{msgs} \)
    \( \land (\text{status}(\text{s, pu}) = \text{OK} \)
      \( \lor \text{status}(\text{s, pu}) = \text{FailureWithoutAck}) \)

within

\[ \text{pred Transmission\_OK : State \times Set[R\_Message]} \]
\( \forall \text{s : State; msgs : Set[R\_Message]} \)
- \( \text{Transmission\_OK}(\text{s, msgs}) \)
  \( \Leftrightarrow \text{msgs\_is\_static\_OK} \land \sim \text{msgs\_is\_NOT\_dynamic\_OK\_for\_s} \)

end

spec \text{Sbc\_State\_3} = \text{Sbc\_State\_2}
then free type

63
ExtendedPumpState ::= sort PumpState | Unknown_PS

op PS_predicted : State \times PumpNumber \rightarrow ExtendedPumpState

\%
\| status(s, Pump(pn)) = OK \leftrightarrow \\
\| not (PS_predicted(s, pn) = Unknown_PS) \% \\
end

spec PUMP_FAILURE =

SBCS_STATE_3

then pred Pump_OK : State \times Set[R_Message] \times PumpNumber

\forall s : State; msgs : Set[R_Message]; pn : PumpNumber

\bullet Pump_OK(s, msgs, pn)

\Leftrightarrow PS_predicted(s, pn) = Unknown_PS \lor PUMP_STATE (pn, PS_predicted (s, pn) as PumpState) \; \text{eps} \; msgs \\
end

spec SBCS_STATE_4 =

SBCS_STATE_3

then free type

ExtendedPumpControllerState ::= sort PumpControllerState | SoonFlow | Unknown_PCS

op PCS_predicted : State \times PumpNumber \rightarrow 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 \times Set[R_Message] \times PumpNumber

\forall s : State; msgs : Set[R_Message]; pn : PumpNumber

\bullet Pump_Controller_OK(s, msgs, pn)

\Leftrightarrow PCS_predicted(s, pn) = Unknown_PCS \\
\lor PCS_predicted(s, pn) = SoonFlow \\
\lor PUMP_CONTROLLER_STATE \\
\; (pn, PCS_predicted (s, pn) as PumpControllerState) \\
\; \text{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 \rightarrow 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 \times Set[R_Message]

\forall s : State; msgs : Set[R_Message]

\bullet Steam_OK(s, msgs) \\
64
\[ \forall v: \text{Value} \]
\[ \Rightarrow \text{low}(\text{steam} \text{predicted}(s)) \leq v \]
\[ \land v \leq \text{high}(\text{steam} \text{predicted}(s)) \]

end

spec \text{LEVEL\_FAILURE} =
\text{SCS\_STATE_5}

then pred \ Level\_OK : \text{State} \times \text{Set}[\text{R\_Message}] \\
\forall s: \text{State}; \text{msgs}: \text{Set}[\text{R\_Message}] \\
\bullet \text{Level}\_OK(s, \text{msgs}) \\
\Rightarrow \forall v: \text{Value} \\
\bullet \text{LEVEL}(v) \text{eps msgs} \\
\Rightarrow \text{low}(\text{level} \text{predicted}(s)) \leq v \\
\land v \leq \text{high}(\text{level} \text{predicted}(s)) 

end

spec \text{FAILURE\_DETECTION} =
[\text{MESSAGE\_TRANSMISSION\_SYSTEM\_FAILURE} \\
\text{and} \ \text{PUMP\_FAILURE} \\
\text{and} \ \text{PUMP\_CONTROLLER\_FAILURE} \\
\text{and} \ \text{STEAM\_FAILURE} \\
\text{and} \ \text{LEVEL\_FAILURE} \\
\text{then pred} \ PU\_OK : \text{State} \times \text{Set}[\text{R\_Message}] \times \text{PhysicalUnit} \\
\forall s: \text{State}; \text{msgs}: \text{Set}[\text{R\_Message}]; pn: \text{PumpNumber} \\
\bullet \text{PU}\_OK(s, \text{msgs}, \text{Pump}(pn)) \\
\Rightarrow \text{Pump}\_OK(s, \text{msgs}, pn) \\
\bullet \text{PU}\_OK(s, \text{msgs}, \text{PumpController}(pn)) \\
\Rightarrow \text{Pump\_Controller\_OK}(s, \text{msgs}, pn) \\
\bullet \text{PU}\_OK(s, \text{msgs}, \text{SteamOutput}) \\
\Rightarrow \text{Steam\_OK}(s, \text{msgs}) \\
\bullet \text{PU}\_OK(s, \text{msgs}, \text{WaterLevel}) \Rightarrow \text{Level\_OK}(s, \text{msgs}) \\
] \\
\text{hide pred} \text{Pump\_OK, Pump\_Controller\_OK, Steam\_OK, Level\_OK} 

end

spec \text{STEAM\_AND\_LEVEL\_PREDICTION} =
\text{FAILURE\_DETECTION} \\
\text{and} \ \text{Set[} \text{sort PumpNumber} \ \text{fit Elem} \Rightarrow \text{PumpNumber}] \\
\text{then local} \\
\text{ops} \ received\_steam : \text{State} \times \text{Set}[\text{R\_Message}] \rightarrow \text{Value}; \\
\text{adjusted\_steam} : \text{State} \times \text{Set}[\text{R\_Message}] \rightarrow \text{Valpair}; \\
\text{received\_level} : \text{State} \times \text{Set}[\text{R\_Message}] \rightarrow \text{Value}; \\
\text{adjusted\_level} : \text{State} \times \text{Set}[\text{R\_Message}] \rightarrow \text{Valpair}; \\
\text{broken\_pumps} : \text{State} \times \text{Set}[\text{R\_Message}] \rightarrow \text{Set}[\text{PumpNumber}]; \\
\text{reliable\_pumps} : \text{State} \times \text{Set}[\text{R\_Message}] \times \text{PumpState} \rightarrow \text{Set}[\text{PumpNumber}] \\
\% \% \text{Axioms for STEAM:} \\
\forall s: \text{State}; \text{msgs}: \text{Set}[\text{R\_Message}]; pn: \text{PumpNumber}; \text{ps}: \text{PumpState} \\
\bullet \text{Transmission}\_OK(s, \text{msgs}) \Rightarrow \text{STEAM}(\text{received\_steam}(s, \text{msgs})) \text{eps msgs} \\
\bullet \text{adjusted\_steam}(s, \text{msgs}) = \text{pair}(\text{received\_steam}(s, \text{msgs}), \text{received\_steam}(s, \text{msgs}))
when Transmission_OK(s, msgs) \& PU_OK(s, msgs, SteamOutput)
else steam_predicted(s)

%% Axioms for LEVEL:
• Transmission_OK(s, msgs)
  \Rightarrow 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)
  \Leftrightarrow \neg PU_OK(s, msgs, PumpController(pn))
• pn eps reliable_pumps (s, msgs, ps)
  \Leftrightarrow \neg pn eps broken_pumps (s, msgs)
  \& PUMP_STATE (pn, ps) eps msgs

within
ops next_steampredicted
  : State \times Set[R_Message] \rightarrow Valpair;
chosen_pumps : State \times Set[R_Message] \times PumpState \rightarrow
  Set[PumpNumber];
minimal_pumped_water, maximal_pumped_water : State \times Set[R_Message] \rightarrow Value;
next_level_predicted : State \times Set[R_Message] \rightarrow Valpair

pred DangerousWaterLevel : State \times Set[R_Message]

%% Axioms for STEAM:
\forall s : State; msgs : Set[R_Message]; pn : PumpNumber
• low(next_steampredicted(s, msgs))
  = max(0, low(adjusted_steampredicted(s, msgs)) - (U2 * dt))
• high(next_steampredicted(s, msgs))
  = min(W, high(adjusted_steampredicted(s, msgs)) + (U1 * dt))

%% Axioms for PUMPS:
• pn eps chosen_pumps (s, msgs, Open)
  \Rightarrow pn eps reliable_pumps (s, msgs, Closed)
• pn eps chosen_pumps (s, msgs, Closed)
  \Rightarrow pn eps reliable_pumps (s, msgs, Open)
• minimal_pumped_water(s, msgs)
  = (dt * P) \ast \#
  (reliable_pumps (s, msgs, Open) - chosen_pumps (s, msgs, Closed))
• maximal_pumped_water(s, msgs)
  = (dt * P) \ast \#
  (((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)))
\[(dt \text{ square} \ast U1 \text{ half}) +
(dt \ast \text{ high} (\text{ adjusted_steam}(s, \text{ msgs})))\]

- high\((\text{next_level}_\text{predicted}(s, \text{ msgs}))\)
  = min
  \((C, \text{maximal_pumped_water}(s, \text{ msgs}))\)
- \((dt \text{ square} \ast U2 \text{ half}) +
(dt \ast \text{ low} (\text{ adjusted_steam}(s, \text{ msgs})))\)

- \(\text{DangerousWaterLevel}(s, \text{ msgs})\)
  \(\Leftrightarrow\) \(\text{low}(\text{next_level}_\text{predicted}(s, \text{ msgs})) \leq M1\)
  \(\lor M2 \leq \text{high}(\text{next_level}_\text{predicted}(s, \text{ msgs}))\)

\(\text{hide ops minimal_pumped_water, maximal_pumped_water}\)

\[\begin{align*}
\text{spec Pump\_State\_Prediction} = \\
\text{Status\_Evolution[Failure\_Detection]} \\
\text{and Steam\_And\_Level\_Prediction}
\end{align*}\]

\[\begin{align*}
\text{then op next\_PS\_predicted} \\
: \text{State} \times \text{Set}\[\text{R\_Message}\] \times \text{PumpNumber} \rightarrow \\
\text{ExtendedPumpState}
\forall s : \text{State}; \text{msgs} : \text{Set}\[\text{R\_Message}\]; \text{pn} : \text{PumpNumber}
\bullet \text{next\_PS\_predicted}(s, \text{msgs}, \text{pn})
\quad = \text{Unknown\_PS}
\quad \text{when } \neg \text{next\_status}(s, \text{msgs}, \text{Pump}(\text{pn})) = \text{OK}
\quad \text{else Open}
\quad \text{when } (\text{PUMP\_STATE}(\text{pn}, \text{Open}) \text{ eps } \text{msgs}
\quad \land \neg \text{pn eps chosen\_pumps}(s, \text{msgs}, \text{Closed}))
\quad \lor \text{pn eps chosen\_pumps}(s, \text{msgs}, \text{Open})
\quad \text{else Closed}
\end{align*}\]

\[\begin{align*}
\text{spec Pump\_Controller\_State\_Prediction} = \\
\text{Status\_Evolution[Failure\_Detection]} \\
\text{and Steam\_And\_Level\_Prediction}
\end{align*}\]

\[\begin{align*}
\text{then op next\_PCS\_predicted} \\
: \text{State} \times \text{Set}\[\text{R\_Message}\] \times \text{PumpNumber} \rightarrow \\
\text{ExtendedPumpControllerState}
\forall s : \text{State}; \text{msgs} : \text{Set}\[\text{R\_Message}\]; \text{pn} : \text{PumpNumber}
\bullet \text{next\_PCS\_predicted}(s, \text{msgs}, \text{pn})
\quad = \text{Unknown\_PCS}
\quad \text{when } \neg \text{next\_status}(s, \text{msgs}, \text{PumpController}(\text{pn})) = \text{OK}
\quad \land \text{next\_status}(s, \text{msgs}, \text{Pump}(\text{pn})) = \text{OK}
\quad \text{else Flow}
\quad \text{when } (\text{PUMP\_CONTROLLER\_STATE}(\text{pn}, \text{Flow}) \text{ eps } \text{msgs}
\quad \lor (\text{PUMP\_CONTROLLER\_STATE}\n\quad (\text{pn}, \text{NoFlow}) \text{ eps } \text{msgs}
\quad \land \text{PCS\_predicted}(s, \text{pn}) = \text{SoonFlow}))
\quad \land \neg \text{pn eps chosen\_pumps}(s, \text{msgs}, \text{Closed})
\end{align*}\]
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
    ∀ 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 \( P : \text{Value} \rightarrow \text{Preliminary} \);
\( S : \text{Preliminary} \rightarrow \text{Sbcs\_State} \);
\( A : \text{Sbcs\_State} \rightarrow \text{Sbcs\_Analysis} \);
\( C : \text{Sbcs\_Analysis} \rightarrow \text{Steam\_Boiler\_Control\_System} \)

result lambda \( V : \text{Value} \bullet C [A [S [P [V]]]] \)
end

spec \( \text{Elem} = \)
sort \( \text{Elem} \)
end

arch spec \( \text{Arch\_Preliminary} = \)
units \( \text{SET} : \text{Elem} \times \text{NAT} \rightarrow \text{Set}[\text{Elem}] \);
\( B : \text{Basics} \);
\( \text{MS} : \text{Messages\_Sent} \text{ given } B \);
\( \text{MR} : \text{Value} \rightarrow \text{Messages\_Received} \text{ given } B \);
\( \text{CST} : \text{Value} \rightarrow \text{Sbcs\_Constants} \)
result lambda \( V : \text{Value} \bullet \)
\( \cdot \text{SET} [\text{MS} \text{ fit Elem} \mapsto S\_\text{Message}] [V] \text{ and } \text{SET} [\text{MR} [V] \text{ fit Elem} \mapsto R\_\text{Message}] [V] \text{ and } \text{CST} [V] \)
end

spec \( \text{Sbcs\_State\_Impl} = \)
\( \text{Preliminary} \)
then free type \( \text{Status} ::= \text{OK} \mid \text{Failure\_Without\_Ack} \mid \text{Failure\_With\_Ack} \)
free type ExtendedPumpState ::= sort PumpState \mid Unknown\_PS
free type ExtendedPumpControllerState ::= sort PumpControllerState \mid Soon\_Flow \mid Unknown\_PCS
free type Valpair ::= pair(low : Value; high : Value)
then \( \text{TotalMap}[\text{Basics fit } S \mapsto \text{PhysicalUnit}][\text{sort Status}] \)
and \( \text{TotalMap} \)
\( \text{[Basics fit } S \mapsto \text{PumpNumber}][\text{sort ExtendedPumpState}] \)
and \( \text{TotalMap} \)
\( \text{[Basics fit } S \mapsto \text{PumpNumber}] \)
\( \text{[sort ExtendedPumpControllerState]} \)
then free type \( \text{State} ::= \text{mk\_state(mode : Mode; \)}
\( \text{numSTOP : NAT; \)}
\( \text{status : TotalMap[PhysicalUnit, Status]; \)}
\( \text{PS\_predicted : TotalMap[PumpNumber, ExtendedPumpState]; \)}
\( \text{PCS\_predicted : TotalMap [PumpNumber, ExtendedPumpControllerState]; \)}
\( \text{steam\_predicted, level\_predicted : Valpair} \)
ops \( \text{status}(s : \text{State}; pu : \text{PhysicalUnit}) : \text{Status} = \text{lookup}(pu, \text{status}(s)); \)
\( \text{PS\_predicted}(s : \text{State}; pn : \text{PumpNumber}) : \text{ExtendedPumpState} = \text{lookup}(pn, \text{PS\_predicted}(s)); \)
\( \text{PCS\_predicted}(s : \text{State}; pn : \text{PumpNumber}) : \text{ExtendedPumpControllerState} = \text{lookup}(pn, \text{PCS\_predicted}(s)) \)
end

unit spec \( \text{Unit\_Sbcs\_State} = \)
\( \text{Preliminary} \rightarrow \text{Sbcs\_State\_Impl} \)
end

arch spec \( \text{Arch\_Analysis} = \)
units \( \text{FD : Sbcs\_State} \rightarrow \text{Failure\_Detection} \);
\( \text{PR : Failure\_Detection} \rightarrow \text{PU\_Prediction} \);
ME : PU_p predicates \rightarrow Mode_Evolution[PU_p predicates];
MTS : Mode_Evolution[PU_p predicates] \rightarrow Sbcs_Analysis
result lambda S : Sbcs_State \& MTS [ME [PR [FD [S]]]]
end

arch spec Arch_Failure_Detection =
  units MTSF : Sbcs_State \rightarrow Message_Transmission_System_Failure;
  PF : Sbcs_State \rightarrow Pump_Failure;
  PCF : Sbcs_State \rightarrow Pump_Controller_Failure;
  LF : Sbcs_State \rightarrow Level_Failure;
  PU : Message_Transmission_System_Failure \times Pump_Failure \times Pump_Controller_Failure \times
      Steam_Failure \times Level_Failure \rightarrow 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 \rightarrow Status_Evolution[Failure_Detection];
  SLP : Failure_Detection \rightarrow Steam_And_Level_Prediction;
  PP : Status_Evolution[Failure_Detection] \times Steam_And_Level_Prediction \rightarrow Pump_State_Prediction;
  PCP : Status_Evolution[Failure_Detection] \times Steam_And_Level_Prediction \rightarrow 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 \rightarrow Sbcs_System_Impl
end

unit spec Sbcs_Open =
  Value \rightarrow Steam_Boiler_Control_System
end

refinement Ref_Sbcs =
  Sbcs_Open refined to arch spec Arch_Sbcs
end

unit spec StateAbstr = Preliminary \rightarrow Sbcs_State
end

refinement StateRef =
  StateAbstr refined to Unit_Sbcs_State
end
refinement $\text{Ref}_{\text{SBCS}}' = \text{Ref}_{\text{SBCS}}$
then $\{ P \text{ to arch spec } \text{Arch}_\text{Preliminary}, S \text{ to State}_{\text{REF}}, A \text{ to arch spec } \text{Arch}_\text{Analysis} \}$
end

refinement $\text{Ref}_{\text{SBCS}}'' = \text{Ref}_{\text{SBCS}}'$
then $\{ A \text{ to}
\begin{align*}
&\text{FD \text{ to arch spec } } \text{Arch}_\text{Failure Detection},
&\text{PR \text{ to arch spec } } \text{Arch}_\text{Prediction} 
\end{align*}
\}$
end