Refinement trees: calculi, tools, and applications

Mihai Codescu and Till Mossakowski
{Mihai.Codescu, Till.Mossakowski}@dfki.de

DFKI GmbH Bremen

Abstract. We recall a language for refinement and branching of formal developments. We introduce 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. This technique has already been applied for showing the consistency of a first-order ontology that is too large to be tackled directly by model finders.

1 Introduction

Consider the task of providing an implementation (or finding a model) for a specification SP. The classical theory of refinement [2] provides the means for vertically 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 an implementation or a model description. For large specifications, this quickly becomes unmanageable. Horizontal decomposition allows the splitting of an implementation task into manageable subtasks [18] that can be implemented independently (by separate people). This leads to a refinement tree, with leaves being specifications that can be implemented directly.

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

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 theories like upper ontologies. Note that the latter is a challenge: recently a contest for showing (in)consistency of the ontology SUMO has been set up [16]. We propose using refinement trees as a way for managing this task; the leaves of the trees then are small theories whose consistency can be proved using an automated model finder.

We recall a language for expressing refinements and branching of formal developments from [13] in Sect. 2 using an example application from [3]. Section 3 formally introduces refinement trees, and section 4 recalls the semantics of refinements and their parts. Section 5 contains our main contribution, a calculus for checking correctness of refinement trees. Moreover, we implement the calculus in a tool that also integrates model finders and other tools. Section 6 provides a calculus for consistency of (possibly architectural) refinements, which already has been successfully applied for showing the consistency of the DOLCE [7] upper ontology. Section 7 concludes the paper.

2 The Casl Refinement Language: Syntax

The Common Algebraic Specification Language Casl [15] has been designed by the "Common Framework Initiative for Algebraic Specification and Development" 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 in orthogonal layers:

1. **basic specifications** provide means for writing specifications of the individual software modules. The underlying logic of Casl is multi-sorted first-order logic with equality, partial functions and induction principles for datatypes;
2. **structured specifications** allow to organize large specifications in a modular way;
3. **architectural specifications** [4] describe, in contrast to the previous layer, the structure of the *implementation* of a software system, given by a (possible structured) specification of the requirements.
4. **libraries of specifications** allow storage and retrieval of named specifications.

The orthogonality of layers means that the syntax and semantics of each layer are independent of those of the others. In particular, this allows to replace 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 the formalization of the notion of logical system as institutions [8] and defining the semantics of structured and architectural specifications in an institution-independent way. Casl is supported by the Heterogeneous Tool Set (Hets) [14], that collects parsing, static analysis and proof management tools for *heterogeneous* specifications. Heterogeneity is achieved by formalizing logics as institutions and adding them to the graph of logics that parameterizes Hets. Moreover, Hets interfaces various theorem provers, model finders or consistency checkers.

Example 1. We will illustrate the Casl architectural and refinement language with the help of an industrial case study: specification of a steam boiler control system for controlling the water level in a steam boiler. The problem has been formulated in [1] as a benchmark for specification languages; [3] gives a complete solution using CASL, including architectural design and refinement of components. However, the refinement steps were presented in an informal way and only the refinement language of [13] makes it possible to formally write down the refinement steps using CASL refinement specifications.

The specifications involved can be briefly explained as follows. VALUE specifies in a very abstract way a sort *Value* and some operations and predicates on values. This specification can be regarded 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. [1]

The initial design for the architecture of the system is recorded by the architectural specification below:

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

Here, the units P, S, A and C are all *generic units*, which denote partial functions taking compatible models of the argument specifications to models of the result specification in a persistent way (that is, the arguments are protected). The compatibility of arguments means that the models can be amalgamated to a model of the union of all the signatures. Moreover, the components are combined in the way prescribed in the result unit of ARCH_SBCS; notice that a model of VALUE is required to able to provide a model of the entire system. □

$ASP ::= S \mid$ **units** $UDD_1 \ldots UDD_n$ **result** UE
$UDD ::= UDEFN \mid UDECL$
$UDECL ::= USP$ **given** $UT_1, \ldots, UT_n$
$USP ::= SP \mid SP_1 \times \cdots \times SP_n \rightarrow SP$
$UDEFN ::= A = UE$
$UE ::= UT \mid \lambda\ A_1 : SP_1, \ldots,\ A_n : SP_n \bullet UT$
$UT ::= A \mid A\ [FIT_1] \ldots [FIT_n] \mid UT$ **and** $UT \mid UT$ **with** $\sigma : \Sigma \rightarrow \Sigma' \mid$
  UT **hide** $\sigma : \Sigma \rightarrow \Sigma' \mid$ **local** $UDEFN_1 \ldots UDEFN_n$ **within** UT
$FIT ::= UT \mid UT$ **fit** $\sigma : \Sigma \rightarrow \Sigma'$

Fig. 1. Syntax of the CASL architectural language.

Fig. 1 presents the syntax of the architectural language of CASL. Here, S stands for an architectural specification name, A for a component name, Σ and

[1] The complete specification of the SBCS example can be found at `https://svn-agbkb.informatik.uni-bremen.de/Hets-lib/trunk/UserManual/Sbcs.casl`

Σ' denote signatures and σ denotes a signature morphism. The architectural language can be shortly explained as follows: an architectural specification *ASP* consists of a list of unit declarations *UDECL* and unit definitions *UDEFN* (where declarations assign unit specifications *USP* to units and definitions assign unit expressions *UE* to units) and a result unit expression formed with the units declared/defined. Unit expressions are used to give definitions for generic units, while unit terms define non-generic units. When the result unit of an architectural specification is generic, the system is 'open', requiring some parameters to provide an implementation. We would like to point out a construction in the architectural language, namely units declared with an optional list of imported unit terms, specified using the **given** clause. This construction has been explained in literature as generic units instantiated once:

```
                                               units M : SP1;
units M : SP1;                                     N : arch spec {
      N : SP2 given M;     is equivalent to                units F : SP1 → SP2
      ...                                                  result F[M]};
                                                   ...
```

This equivalence has been made formal in [6] and thus allows us to treat the first syntactic construction as a "syntactic sugar" for the second, reducing thus the complexity of semantics and verification of architectural specifications. Notice that the name F of the generic unit is chosen arbitrarily and in order to be able to refine the unit N we must slightly adapt the original semantics rules for refinements from [13]. We will address this in more detail in Sections 4 and 5.

In [13], the architectural layer has been complemented with a *refinement* language, which allows to formalize developments as refinement trees. The language provides syntactic constructs (Fig. 2) for expressing refinement between specifications, starting with the simplest form which denotes just model class inclusion between unit specifications: *USP* **refined via** σ **to** *USP'* is a correct refinement when $M|_\sigma \in Mod(USP)$ for any model M of *USP'* (here, $M|_\sigma$ denotes the reduct of M against the signature morphism σ). We denote this $USP \rightsquigarrow_\sigma USP'$ and when σ is identity or clear from the context we omit it. This grows in complexity to compositions of refinements (written as *USP* **refined to** *RSP*, where *RSP* is now an arbitrarily complex refinement or as *RSP* **then** *RSP'*), branching introduced by architectural design, with the specifications of the components being now refinement specifications and finally refinement of components of architectural specifications (written $\{UN_i$ **to** $RSP_i\}_{i\in\mathcal{J}}$, where UN_i stands for a unit name). Notice that refinement specifications subsume architectural specifications and we can therefore speak of a refinement level subsuming the architectural level in the CASL language. In Fig. 2 R stands for the name of a refinement specification.

Example 2. We can then write the initial refinement of the steam boiler system as

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

RSP := **refinement** R = RSP | USP | **arch spec** ASP | RSP **then** RSP |
SP **refined** [**via** σ] **to** RSP | $\{A_1$ **to** $RSP_1, \ldots, A_n$ **to** $RSP_n\}$
$UDECL$ ::= $A : RSP$ | $A : USP$ **given** $UT_1, \ldots, UT_n$

Fig. 2. Syntax of the CASL refinement language.

We further proceed with refining the individual units [2]. The specifications of C and S in ARCH_SBCS above do not require further architectural decomposition. The specification of S, recorded in the unit specification STATE_ABSTR, can be refined by providing an implementation of states as a record of all observable values. This is done in SBCS_STATE_IMPL, assuming an implementation of PRELIMINARY; we record this development in the unit specification UNIT_SBCS_STATE. The refinement of S is then written in STATE_REF.

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

For the units P and A, we proceed with designing their architecture. This is recorded in the architectural specifications ARCH_PRELIMINARY and ARCH_ANALYSIS (omitted here). We can now record the component refinement:

refinement REF_SBCS' = REF_SBCS **then**
        {P **to arch spec** ARCH_PRELIMINARY,
        S **to** STATEREF,
        A **to arch spec** ARCH_ANALYSIS}

Moreover, the components of ARCH_ANALYSIS are further refined:

refinement REF_SBCS" =
        REF_SBCS'
        then {A **to**
            {FD **to arch spec** ARCH_FAILURE_DETECTION,
            PR **to arch spec** ARCH_PREDICTION }}

□

3 Refinement Trees

We now give a formal definition of the concept of *refinement trees*. Refinement trees provide visualization means for the structure of the development and access points for the logical properties of architectural and refinement specifications.

[2] We tacitly correct some rather minor mistakes in the specifications in [13] that were revealed while testing the implementation of the static analysis rules of refinement in Hets.

They can be regarded as a counterpart of the development graphs [12] constructed for structured specifications. Notice that the proof calculus for refinements presented in Section 5 produces a development graph for proving the verification conditions, but the components of the system decomposition are quite difficult to observe, since e.g. the nodes introduced in applications of generic units and edges for dependencies between units in architectural specifications are present. The approach that we propose separates the diagram of dependencies from the tree-like representation of the development process.

While intuitively clear, refinement trees are built in a stepwise manner and must be *combined* in the way prescribed by the refinement specifications: composition of refinements gives rise to composition of refinement trees, and we need a mechanism for keeping track of the branches and nodes in the tree corresponding to units and connection points between trees. Moreover, in the case of component refinement, each component produces a (sub)tree.

Example 3. Fig. 3 presents the refinement tree of the specifications in Example 1. Single arrows denote components, while double arrows denote refinements.

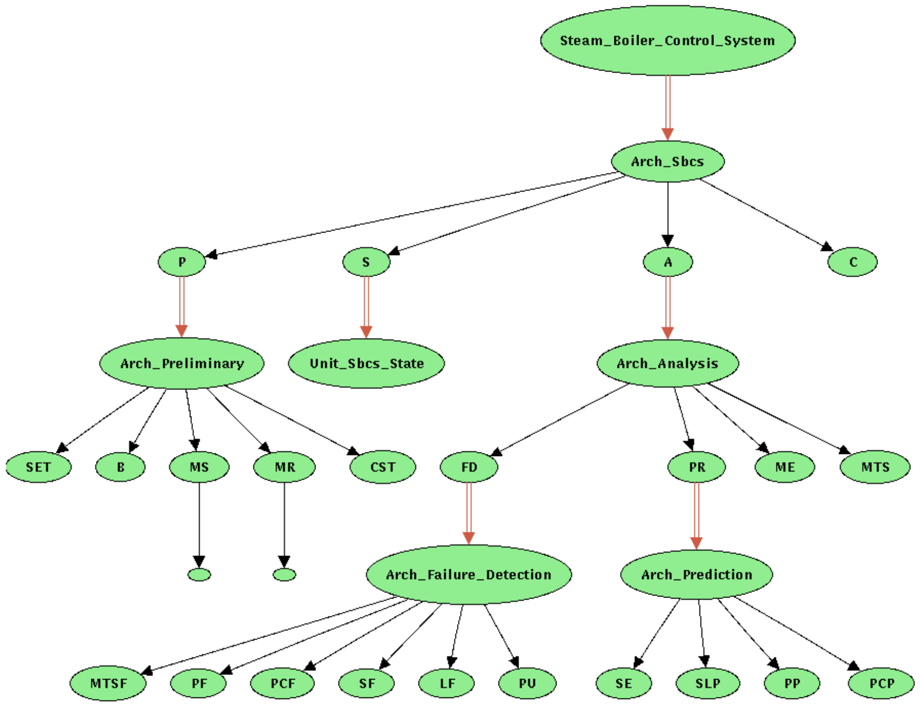

Fig. 3. The refinement tree of the steam boiler control system.

Notice that in the case of e.g. REF_SBCS”, we need to build the trees of the architectural specifications ARCH_FAILURE_DETECTION and ARCH_PREDICTION, store them as corresponding to the units FD and PR in a component refinement, obtaining thus a set of trees, then connect them to the corresponding components of the unit A, which must be correctly identified.

The example shows that refinement trees should consist of a collection of trees and that they can grow not only at the leaves, but also at the root, thus becoming subtrees. This leads us to the following definition.

Definition 4. *A refinement tree $\mathcal{RT}$ consists of a set of trees with nodes labelled with unit specifications and edges that can be either (i)* refinement links $n_1 \Rightarrow n_2$ *to denote refinement of specifications or (ii)* component links $n_1 \rightarrow n_2$*, where n_1, n_2 are nodes in $\mathcal{RT}$ to denote architectural decomposition.*

We need to define an auxiliary structure to keep track of the roots, leaves and nodes of the branching decompositions; this will make possible to compose refinement trees. We would like to stress that this is only done for book-keeping and it is not visible to the user at all. Let us define *refinement tree pointers* in a refinement tree $\mathcal{RT}$ as either (i) *simple refinement pointers* of form (n_1, n_2) with n_1, n_2 nodes in $\mathcal{RT}$, with the intuition that the first node is the root and the second node is the leaf of a chain[3], (ii) *branching refinement pointers* of form (n, f), where n is a node and f is a map assigning refinement tree pointers to unit names, or (iii) *component refinement pointers* which are maps assigning refinement tree pointers to unit names.

Let us give a series of notations and operations with refinement trees. We denote $\mathcal{RT}_\emptyset$ the empty tree. If $\mathcal{RT}$ is a refinement tree, $\mathcal{RT}[USP]$ is obtained by adding to it a new isolated node (also returned as result) labelled with *USP*. For $\mathcal{RT}_1, \ldots, \mathcal{RT}_k$ refinement trees, $\mathcal{RT}[n_1 \rightarrow \mathcal{RT}_1, \ldots, \mathcal{RT}_k]$ denotes the tree obtained by inserting component links from the node n_1 of $\mathcal{RT}$ to the roots of each of the argument trees. Moreover, for two refinement trees $\mathcal{RT}_1$ with pointer p_1 and $\mathcal{RT}_2$ with pointer p_2, we denote $(p, \mathcal{RT})$ the composition $\mathcal{RT}_1 \circ p_1, p_2 \mathcal{RT}_2$ defined as follows:

- if p_1 is a simple refinement pointer (n_1, n_2) and p_2 is a simple refinement pointer (m_1, m_2), $\mathcal{RT}$ is obtained by adding a refinement link from n_2 to the root of the subtree of m_1 in $\mathcal{RT}_2$. The pointer p is then (n_1, m_2). This corresponds to STATE_REF in Example 1, see branch for unit S in Fig. 3.
- if p_1 is a simple refinement pointer (n_1, n_2) and p_2 is a branching refinement pointer (m_1, f), $\mathcal{RT}$ is obtained by adding a refinement link from n_2 to m_1. The pointer p is (n_1, f). This corresponds to REF_SBCS in Example 1, see the initial refinement link in Fig. 3.
- if p_1 is a branching refinement pointer (n_1, f_1) and p_2 is a component refinement pointer f_2, $\mathcal{RT}$ is obtained by making for each X in $dom(f_2)$ the composition of the subtree pointed by $f_1(X)$ with the tree $f_2(X)$, which also returns a pointer p_X. The pointer p is $(n_1, f_1[f_2])$, where $f_1[f_2]$ updates the value of X in f_1 with the pointer p_X. This corresponds to REF_SBCS' in Example 1 and introduces the second level of refinement links in Fig. 3.

[3] Notice that the two nodes can coincide.

– if p_1 is a component refinement pointer f_1 and p_2 is a component refinement pointer p_2, then the refinement tree is obtained by making for each X in $dom(f_2)$ the composition of the subtree pointed by $f_1(X)$ with the tree $f_2(X)$, which also returns a pointer p_X. The pointer p is $f_1[f_2]$. We can obtain an example for this last case by writing the refinement of the components in a slightly different way:

```
refinement R = {
            P to arch spec ARCH_PRELIMINARY, S to STATEREF,
            A to arch spec ARCH_ANALYSIS }
            then {
            A to {FD to arch spec ARCH_FAILURE_DETECTION,
                     PR to arch spec ARCH_PREDICTION }}
```

and the corresponding refinement trees are obtained by removing the first three levels from the tree in Fig. 3.

The composition is undefined otherwise. Notice that the specification of a unit, needed to label the nodes of refinement trees, will be obtained in some cases with the proof calculus for refinement specifications that we present in Section 5 and therefore the refinement trees will be built with the proof calculus rules.

4 The CASL Refinement Language: Semantics

The semantics of refinement specifications is, as usual in CASL, model-theoretic: a specification denotes a signature (static semantics) and a class of models (model semantics). We briefly recall the semantics of architectural specifications (see [15] for details). Unit specifications describe self-contained units (models) or generic units, mapping compatible models of the arguments to models of the result specification, such that the arguments are preserved under reduct. Their static semantics is given by unit signatures, which are themselves either plain signatures Σ or generic unit signatures $\Sigma_1 \times \cdots \times \Sigma_n \to \Sigma$ where Σ extends the union of $\Sigma_1, \ldots, \Sigma_n$. Plain CASL static semantics of architectural specifications consist of a unit signature for the result unit and a mapping giving a unit signature for any unit component, while the model semantics consists of a class of architectural models, which themselves pair a model over the unit signature of the result with a mapping giving a model for each component.

For the static semantics of refinements, [13] introduces *refinement signatures*, $R\Sigma$, which take one of the following forms: (i) *unit refinement signatures* $(U\Sigma, U\Sigma')$ which consist of two unit signatures and correspond to simple refinements (ii)*branching refinement signatures* $(U\Sigma, B\Sigma')$ which consist of a unit signature $U\Sigma$ and a *branching signature* $B\Sigma'$, which is either a unit signature $U\Sigma'$ (in which case the branching refinement signature is a unit refinement signature) or a *branching static context* $BstC'$, which is in turn a (finite) map assigning branching signatures to unit names, and corresponds to architectural decompositions, and finally (iii) *component refinement signatures* which are (finite) maps $\{UN_i \mapsto R\Sigma_i\}_{i\in\mathcal{J}}$ from unit names to refinement signatures and give static semantics for refinements of components.

The rules for static and model semantics of refinements are introduced in [13] as well; due to space limitations, we do not repeat them here. The only modification that we bring handles the case of units with imports, as mentioned in Section 2. The refinement signature of a unit with imports is a branching refinement signature with the branching static context having only one entry; since the architectural specification that equivalently expresses units with imports is generated, the name of the generic unit is not available to the user. The convention that we propose is to adjust the definition of *composition of refinement signatures* given in [13].

Given refinement signatures $R\Sigma_1$ and $R\Sigma_2$, their *composition* $R\Sigma_1\,;\,R\Sigma_2$ is defined inductively on the form of the first argument. We extend the definition by making the composition $R\Sigma_1;\ R\Sigma_2$ also defined when $R\Sigma_1$ is of form $(U\Sigma, BstC)$ with only one unit name UN in the domain of $BstC$ and the composition $R\Sigma'$ of $BstC(UN)$ with $R\Sigma_2$ is defined . In this case, $R\Sigma_1;\ R\Sigma 2 = (U\Sigma, BstC[UN/R\Sigma'])$.

We will also make use of the *refinement relations* introduced in the paper cited above, which provide a notion of refinement model. Given a refinement signature $R\Sigma$, *refinement relations*, $\mathcal{R}$, are classes of *assignments*, R, which take the following forms:

- *unit assignments*, for $R\Sigma = (U\Sigma, U\Sigma')$, are pairs (U, U') of units over unit signatures $U\Sigma$ and $U\Sigma'$, respectively;
- *branching assignments*, for $R\Sigma = (U\Sigma, B\Sigma')$, are pairs (U, BM'), where U is a unit over the unit signature $U\Sigma$ and BM' is a *branching model* over the branching signature $B\Sigma'$, which is either a unit over $B\Sigma'$ when $B\Sigma'$ is a unit signature (in which case the branching assignment is a unit assignment), or a *branching environment* BE' that fits $B\Sigma'$ when $B\Sigma'$ is a branching static context. Branching environments are (finite) maps assigning branching models to unit names, with the obvious requirements to ensure compatibility with the branching signatures indicated in the corresponding branching static context.
- *component assignments*, for $R\Sigma = \{UN_i \mapsto R\Sigma_i\}_{i\in\mathcal{J}}$, are (finite) maps $\{UN_i \mapsto R_i\}_{i\in\mathcal{J}}$ from unit names to assignments over the respective refinement signatures. When $R\Sigma$ is a refined-unit static context (and so each R_i, $i \in \mathcal{J}$, is a branching assignment) we refer to $RE = \{UN_i \mapsto (U_i, BM_i)\}_{i\in\mathcal{J}}$ as a *refined-unit environment.* Any such refined-unit environment can be naturally coerced to a unit environment $\pi_1(RE) = \{UN_i \mapsto U_i\}_{i\in\mathcal{J}}$ of the plain CASL semantics, as well as to a branching environment $\pi_2(RE) = \{UN_i \mapsto BM_i\}_{i\in\mathcal{J}}$.

Again, the only change to the rules of model semantics is to adapt in the expected way the *composition* of refinement relations to accommodate refinements of unit with imports, in symmetry with the change of static semantics.

The static semantic rules are of form $\vdash SPR \rhd R\Sigma$ while the model semantic rules are of form $\vdash SPR \Rightarrow \mathcal{R}$; when we only want to state that SPR has a denotation w.r.t. the static or model semantics, the result is replaced with $\Box$.

5 Proof Calculus for the Casl Refinement Language

The main motivation for formalizing the development process using Casl architectural specifications and refinements is that one can then *prove* that the process is correct. Verification of correctness for a refinement tree is presented as a *proof calculus*. In [15], Section IV:5, a proof calculus for verification of architectural specifications (in a slightly restricted variant) was introduced as an algorithm for checking whether the resulting units of an architectural specification satisfy a given unit specification. This is denoted $\vdash ASP :: USP$, where ASP is an architectural specification and USP is a unit specification. Since architectural specifications are now a particular case of refinements, we will extend this calculus to support the whole refinement language.

For space limitation reasons, we omit the complete presentation of the proof calculus of architectural specifications mentioned above and just explain the rough intuition. It can be regarded as having two components. The first one is a *constructive* component, building a graph Γ with nodes labeled with non-generic unit signatures and edges labeled with signature morphisms where additionally some of the nodes may be labeled with sets of specifications. Moreover, generic units declared in ASP are stored in a generic context Γ_{gen}. The second component is *deductive* and it uses the diagram built with the constructive component to check whether models of a unit expression satisfy a given unit specification SP. Fig. 4 presents one of the interesting rules of the calculus, namely those for unit expressions (only non-generic). What is particular about the rules for unit expressions is that here is where the two components of the calculus meet: the proof calculus rules for unit terms (of format $\Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A$) modify the context by adding the diagram of the unit term and also return the node of the unit term, and this node is further used for checking in Γ' whether the given specification actually holds or not. On the notations, R is an institution translation (formalised as so-called comorphism) that translates from the institution of interest to another with weak amalgamation property (i.e., given a diagram and a family of models compatible with it, they can be combined to a model of a cocone for the diagram) and the refinement condition verifies that for any family of models compatible with the diagram the model corresponding to the node A satisfies SP.

$$\frac{\begin{array}{c}\Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A \\ \text{for all } U \in \mathrm{dom}(\Gamma'), \text{ we have } U :_{\Sigma_U} \mathcal{SP}_U \text{ in } \Gamma' \\ \Sigma, \{\eta_U\}_{U \in \mathrm{dom}(\Gamma')} \text{ is a weakly amalgamable cocone over } R(\Gamma') \\ \eta_A(R(SP)) \rightsquigarrow_{\Sigma}^{J} \bigcup_{U \in \mathrm{dom}(\Gamma')} \eta_U(\overline{R}(\mathcal{SP}_U))\end{array}}{\Gamma_{gen}, \Gamma \vdash UT \text{ qua } UE :: SP}$$

Fig. 4. Architectural proof calculus rule for non-generic unit expressions.

As a first step, we modify the proof calculus such that it becomes fully constructive: *USP* is no longer provided, but rather obtained by defining the specification of each unit term inductively on its structure.

Definition 5. *Let ASP be an architectural specification and UE a unit expression. Then the specification of UE, denoted $\mathcal{S}_{ASP}(UE)$ is defined as follows:*

- *if UE is a unit term UT, $\mathcal{S}_{ASP}(UT)$ is defined inductively:*
  - *if UT is a unit name, then $\mathcal{S}_{ASP}(UT) = SP$ where $UT : SP$ is the declaration of UT in ASP;*
  - *if $UT = F[A_1 \textbf{ fit } \sigma_1] \dots [A_n \textbf{ fit } \sigma_n]$, where $\mathcal{S}_{ASP}(F) = SP_1 \times \dots \times SP_n \rightarrow SP$ and for any $i = 1, \dots, n$, $\mathcal{S}_{ASP}(A_i) \textbf{ hide } \sigma_i \models SP_i$, then $\mathcal{S}_{ASP}(UT) = \{SP \textbf{ with } \sigma\} \textbf{ and } \mathcal{S}_{ASP}(A_1) \textbf{ hide } \sigma_1 \textbf{ and} \dots \textbf{and } \mathcal{S}_{ASP}(A_n) \textbf{ hide } \sigma_n$, where $\sigma = \cup_{i=1,\dots,n}\sigma_i$;*
  - *if $\mathcal{S}_{ASP}(A_i) = SP_i$ then $\mathcal{S}_{ASP}(A_1 \textbf{ and } \dots \textbf{ and } A_n) = SP_1 \textbf{ and } \dots \textbf{ and } SP_n$;*
  - *if $\mathcal{S}_{ASP}(A) = SP$, then $\mathcal{S}_{ASP}(A \textbf{ with } \sigma) = SP \textbf{ with } \sigma$;*
  - *if $\mathcal{S}_{ASP}(A) = SP$, then $\mathcal{S}_{ASP}(A \textbf{ hide } \sigma) = SP \textbf{ hide } \sigma$;*
  - *$\mathcal{S}_{ASP}(\textbf{local } UDEFN \textbf{ within } UT) = \mathcal{S}_{ASP}(UT)$, where UDEFN is used to obtain the specification of the locally defined units.*
- *if UE is a lambda expression $\lambda A : SP \,.\, UT$, then $\mathcal{S}_{ASP}(UE) = SP \rightarrow \mathcal{S}_{ASP}(UT)$.*

The specification of a unit term also gets employed in expressing declarations of units with imports as generic units: a declaration $UN : SP$ **given UT** can be written as $UN :$ **arch spec**$\{$**units** $F : \mathcal{S}_{ASP}(UT) \rightarrow SP$; **result F[UT]**$\}$. As noticed in [9], it is not always possible to give a precise axiomatization of the class of models produced by the unit expression. However, we will use $\mathcal{S}_{ASP}(UE)$ as an approximation, since models of the unit expression are also models of this specification.

The proof calculus is then turned into a fully constructive variant by removing verification conditions from the rules for unit expressions and returning the specification of the result unit expression of the architectural specification instead of having one as a input parameter of the calculus. Let us denote the constructive version of the architectural proof calculus (built using $\mathcal{S}_{ASP}(UE)$) by $\vdash ASP ::_c USP$.

When the architectural language is restricted by omitting the unit imports , the constructive and the deductive versions of the proof calculus are related by the following straightforward result.

Proposition 6. *If unit imports are omitted, $\vdash ASP \rhd \Box$ and $\vdash ASP ::_c USP$ then $\vdash ASP :: USP$.*

Moreover, in some cases, the obtained unit specification exactly captures the models of the architectural specification, if the latter are projected with the possible semantics for the result unit term. Let therefore $ProjRes$ take any model of an architectural specification to the interpretation of its result unit term in this model.

Theorem 7. *Let ASP be a consistent architectural specification without unit imports and unit definitions, where each parametric unit is applied only once. If* $\vdash ASP \rhd \Box$ *and* $\vdash ASP ::_c USP$ *then* $ProjRes(\mathbf{Mod}(ASP)) = \mathbf{Mod}(USP)$.

While the proof calculus of architectural specifications checks that result units satisfy a given unit specification, we introduce a counterpart for the specification that refinements should satisfy, tailored according to the three kinds of refinements. This allows to express the proof calculus rule for compositions of refinements in a more concise manner.

Definition 8. *Let $R\Sigma$ be a refinement signature. A* verification specification *S over $R\Sigma$ is defined as follows:*

- *if $R\Sigma = (U\Sigma_1, U\Sigma_2)$, $S = (USP_1, USP_2)$ such that $\vdash USP_i \rhd U\Sigma_i$, for i=1,2;*
- *if $R\Sigma = (U\Sigma, B\Sigma)$, $S = (USP, BSP)$, where $\vdash USP \rhd U\Sigma$ and BSP is a branching specification, which is in turn either a unit specification USP' such that $\vdash USP' \rhd U\Sigma'$, when $B\Sigma = U\Sigma'$ or a map SPM such that and $SPM(X)$ is a verification specification over $BstC(X)$, for any $X \in dom(BstC)$, when $B\Sigma = BstC$;*
- *if $R\Sigma = \{UN_i \mapsto R\Sigma_i\}_{i \in \mathcal{J}}$, then $S = \{UN_i \mapsto S_i\}_{i \in \mathcal{J}}$, where S_i is a verification specification over $R\Sigma_i$.*

Again, the proof calculus rules rely on a composition operation on verification specifications. The composition $S_1;\ S_2$ is defined inductively as follows:

- if $S_1 = (USP_1, USP_2)$, then $S_1;\ S_2$ is defined only when $S_2 = (USP_3, SPM)$ and moreover $\vdash USP_2 \rightsquigarrow USP_3$. Then $S_1;\ S_2 = (USP_1, SPM)$.
- if $S_1 = (USP_1, SPM_1)$ then S_2 must be of form SPM_2. We define $S_1;\ S_2 = (USP_1, SPM_1[SPM_2])$, where $SPM_1[SPM_2](A) = SPM_1(A)$, if $A \notin dom(SPM_2)$ and $SPM_1[SPM_2](A) = SPM_1(A);\ SPM_2(A)$ otherwise.
- if $S_1 = SPM_1$, then $S_1;\ S_2$ is defined only if $S_2 = SPM_2$. Then $S_1;\ S_2$ modifies the ill-defined union of SPM_1 and SPM_2 by putting $(S_1;\ S_2)(A) = S_1(A);\ S_2(A)$ for any $A \in dom(S_1) \cap dom(S_2)$.

The proof calculus for architectural specifications is complemented at the level of refinement specifications as in Fig. 5. The judgments of the proof calculus for refinements are then of form $\vdash SPR :: S, \mathcal{RT}, p$, where SPR is a refinement specification, S is a verification specification, $\mathcal{RT}$ is a refinement tree and p is a refinement tree pointer. By a slight abuse of notation, when we are not interested in the refinement tree and the pointer, we omit them as results.

Notice that the proof calculus for architectural specifications of [15] only takes into account the case when the specification of a unit is a unit specification. In the context of refinements, the specification of a unit can be an arbitrary refinement specification, and the proof calculus of architectural specifications must be therefore adapted. This means that for a unit declaration $UN_i : SPR_i$, we define $\mathcal{S}_{ASP}(UN_i) = USP_i$ if $\vdash SPR_i :: (USP_i, SPM_i)$. Moreover, we can set $SPM(UN_i) = SPM_i$ in the verification specification of ASP. Finally, in the case of named refinement specifications, we need to store their verification specifications at the library level and retrieve them by name, in the usual way.

$$\frac{(n,\mathcal{RT}) = \mathcal{RT}_{\emptyset}[\mathit{USP}]}{\vdash \mathit{USP} :: (\mathit{USP},\mathit{USP}),\mathcal{RT},(n,n)} \qquad \frac{\begin{array}{c}\vdash \mathit{USP} :: (\mathit{USP},\mathit{USP}),\mathcal{RT}_1,p_1\\ \vdash \mathit{SPR} :: (\mathit{USP}',\mathit{BSP}),\mathcal{RT}_2,p_2\\ (\mathcal{RT},p) = \mathcal{RT}_1 \circ_{p_1,p_2} \mathcal{RT}_2\\ \vdash \mathit{USP} \leadsto \mathit{USP}'\end{array}}{\vdash \mathit{USP}\ \textbf{refined to}\ \mathit{SPR} :: (\mathit{USP}',\mathit{BSP}),\mathcal{RT},p}$$

$$\frac{\begin{array}{c}\vdash \mathit{ASP} ::_c \mathit{USP}\\ \vdash \mathit{SPR}_i :: (\mathit{USP}_i,\mathit{BSP}_i),\mathcal{RT}_i,p_i\\ \text{for any } UN_i : \mathit{SPR}_i \text{ in } \mathit{ASP}\\ \mathit{SPM}(UN_i) = \mathit{BSP}_i\\ (n,\mathcal{RT}') = \mathcal{RT}_{\emptyset}[\mathit{USP}]\\ \mathcal{RT} = \mathcal{RT}'[n \to \mathcal{RT}_1,\dots,\mathcal{RT}_k]\\ p = \{UN_i \mapsto p_i\}_{i=1,\dots,k}\end{array}}{\vdash \mathit{ASP} :: (\mathit{USP},\mathit{SPM}),\mathcal{RT},p} \qquad \frac{\begin{array}{c}\vdash \mathit{SPR}_i :: S_i,\mathcal{RT}_i,p_i\\ \mathcal{RT} = \cup\mathcal{RT}_i\\ p = (n,\{UN_i \to p_i\})\end{array}}{\vdash \{UN_i\ \textbf{to}\ \mathit{SPR}_i\}_{i\in\mathcal{J}} :: \{UN_i \to S_i\}_{i\in\mathcal{J}},\mathcal{RT},p}$$

$$\frac{\begin{array}{c}\vdash \mathit{SPR}_1 :: S_1,\mathcal{RT}_1,p_1\\ \vdash \mathit{SPR}_2 :: S_2,\mathcal{RT}_2,p_2\\ S = S_1;\ S_2\\ (p,\mathcal{RT}) = \mathcal{RT}_1 \circ_{p_1,p_2} \mathcal{RT}_2\end{array}}{\vdash \mathit{SPR}_1\ \textbf{then}\ \mathit{SPR}_2 :: S,\mathcal{RT},p}$$

Fig. 5. Proof calculus for CASL refinements.

Definition 9. *Let $R\Sigma$ be a refinement signature, S a verification specification of $R\Sigma$ and $\mathcal{R}$ a refinement relation over $R\Sigma$. We define the satisfaction of a verification specification by a refinement relation, denoted $\mathcal{R} \models S$, inductively as follows:*

- *if $R\Sigma = (U\Sigma, U\Sigma')$, then $S = (\mathit{USP}, \mathit{USP}')$ and $\mathcal{R} = \{(u,u')|u \in Unit(U\Sigma), u' \in Unit(U\Sigma')\}$. Then $\mathcal{R} \models S$ iff $u \in Unit(\mathit{USP})$ and $u' \in Unit(\mathit{USP}')$ for any $(u,u') \in \mathcal{R}$;*
- *if $R\Sigma = (U\Sigma, B\Sigma)$, then $S = (\mathit{USP}, \mathit{SPM})$ and $\mathcal{R} = \{(u,bm)|u \in Unit(U\Sigma)$, bm is a branching model over $B\Sigma\}$. Then $\mathcal{R} \models S$ iff for any $(u,bm) \in R$, $u \in Unit(\mathit{USP})$ and for any $A \in dom(\mathit{SPM})$ we have that $bm(A) \models S(A)$. (Notice that SPM and bm have the same domain);*
- *if $R\Sigma = \mathit{SPM}$, then $S = \{UN_i \to S_i\}_{i\in\mathcal{J}}$ and $\mathcal{R} = \{UN_i \to \mathcal{R}_i\}_{i\in\mathcal{J}}$. Then $\mathcal{R} \models S$ iff $\mathcal{R}_i \models S_i$ for any i.*

The following lemma can be proven by induction and making a case distinction on refinement signatures.

Lemma 10. *Let $R\Sigma_1, R\Sigma_2$ be two refinement signatures such that their composition is defined. Let S_i be a verification specification over $R\Sigma_i$ and $\mathcal{R}_i$ be a refinement relation over $R\Sigma_i$ such that $\mathcal{R}_i \models S_i$, for $i = 1,2$ such that $\mathcal{R}_1;\ \mathcal{R}_2$ and $S_1;\ S_2$ are defined. Then $\mathcal{R}_1;\ \mathcal{R}_2 \models S_1;\ S_2$.*

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

Theorem 11 (Soundness).
Let SPR be a refinement specification such that $\vdash SPR \rhd \Box$. If $\vdash SPR :: S$, then there is $\mathcal{R}$ such that $\vdash SPR \Rightarrow \mathcal{R}$ and $\mathcal{R} \models S$.

Because we approximate the specification of the result unit of architectural specifications, completeness is much more difficult to obtain and is therefore postponed to future work.

6 Checking Consistency of Refinement Specifications

We introduce a calculus for checking whether a refinement specification is consistent, i.e. it has a refinement model. In [10], we have successfully applied this calculus to verify the consistency of the upper ontology Dolce. Indeed, 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 an architectural refinement. This has the advantage of giving a modular model for Dolce, i.e. one that can be changed at various local places (= leaves of the refinement tree) without affecting the possibility to assemble (via the semantics of architectural specifications) a global model of Dolce.

Intuitively, a refinement is consistent if its target is, and an architectural specification is consistent if all its unit specifications are. This makes it clear that our calculus eventually (for checking consistency of the leaves of the refinement tree) needs to be based on a calculus for the consistency of unit specifications, which we denote $\vdash cons(USP)$ and is given by the rules in Fig. 6. Checking consistency of non-parametric unit specification amounts to checking consistency of structured specifications; a calculus for this has been introduced in [17] (this is in turn based on some institution-specific calculus for consistency of basic specifications). Checking consistency of parametric unit specification amounts to checking conservativity of extensions of structured specifications; for the case of first-order logic and Casl basic specifications, a calculus has been developed in [11].

$$\frac{\vdash cons(USP)}{\vdash cons(USP\ qua\ \texttt{SPEC-REF})} \qquad \frac{\vdash cons(SPR)}{\vdash cons(USP\ \textbf{refined to}\ SPR)}$$

$$\frac{\vdash cons(SPR)\ for\ any\ UN : SPR\ in\ ASP}{\vdash cons(ASP)} \qquad \frac{\vdash cons(SPR_i)}{\vdash cons(\{U_i\ \textbf{to}\ SPR_i\}_{i \in \mathcal{J}})}$$

$$\frac{\begin{array}{c}\vdash cons(SPR_1)\\ \vdash cons(SPR_2)\end{array}}{\vdash cons(SPR_1\ \textbf{then}\ SPR_2)}$$

Fig. 6. Consistency calculus for refinement specifications.

Note that in the case of compositions, if SPR_1 contains a branching, it does not suffice for SPR_2 (which must be a component refinement) to be consistent, because some component of SPR_1 outside the domain of SPR_2 might be inconsistent.

Proposition 12 (Soundness).
If $\vdash SPR \rhd \Box$ *and* $\vdash cons(SPR)$*, there is a refinement relation* $\mathcal{R}$ *such that* $\vdash SPR \Rightarrow \mathcal{R}$*.*

Completeness holds only by restricting the language again to a variant without imports.

Proposition 13 (Completeness).
If unit imports are omitted, $\vdash SPR \rhd \Box$ *and* $\vdash SPR \Rightarrow \mathcal{R}$*, then* $\vdash cons(SPR)$*.*

7 Conclusions

We have recalled the language for refinements in CASL and we provided a sound proof calculus for it. Thus we can formalize development process for software systems and prove their correctness. Moreover, we have introduced refinements trees in theory, and also practically implemented them in the Heterogeneous Tool Set Hets, such that browsing through and inspection of complex formal developments becomes possible. An implementation of the proof calculus is currently in progress; the refinement part is already implemented. Note that the proof calculus for architectural specifications of [15] was given for a restricted version of the language; it can be extended to the whole language in a way substantially simplified by the transformation of units with imports into generic units. We also have introduced and implemented a sound and complete calculus for consistency of refinements and architectural specification, which already has been applied for proving the consistency of the upper ontology Dolce in a modular way.

Future work includes extending the language to support *behavioral refinement*. Often, a specification does not satisfy the requirements literally, but only up to some observational equivalence. The standard example is the implementation of stacks as arrays with pointer that may differ (inessentially w.r.t. the behavior) on the entries beyond the pointer position. This has been discussed in the case of CASL in [5]. Another useful addition would be amalgamability checks for other logics in the Hets' logic graph, making thus possible to have architectural specifications in that logic.

Acknowledgement. This work has been supported by the German Research Council (DFG) under grant Mo-971/2 "Logic Atlas and Integration (LATIN)".

References

1. Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995)*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996.

2. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner. *Algebraic Foundations of Systems Specification.* Springer, 1999.
3. Michel Bidoit and Peter D. Mosses. CASL *User Manual.* LNCS 2900 (IFIP Series). Springer, 2004.
4. Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273, 2002.
5. Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Observational interpretation of CASL specifications. *Mathematical Structures in Computer Science*, 18(2):325–371, 2008.
6. Mihai Codescu. Lambda Expressions in CASL Architectural Specifications. In Till Mossakowski and Hans-Jörg Kreowski, editors, *Recent Trends in Algebraic Development Techniques, 20th International Workshop, WADT 2010*, Lecture Notes in Computer Science. Springer, 2011.
7. A. Gangemi, N. Guarino, C. Masolo, A. Oltramari, and L. Schneider. Sweetening ontologies with DOLCE. In A. Gómez-Pérez and V. R. Benjamins, editors, *EKAW*, volume 2473 of *LNCS*, pages 166–181. Springer, 2002.
8. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992. Predecessor in: LNCS 164, 221–256, 1984.
9. Piotr Hoffman. *Architectural Specifications and Their Verification.* PhD thesis, Warsaw University, 2005.
10. Oliver Kutz and Till Mossakowski. A modular consistency proof for Dolce. In *25th conference on Artificial Intelligence, AAAI-11*, 2011. To appear.
11. Mingyi Liu. Konsistenz-Check von CASL-Spezifikationen. Master's thesis, University of Bremen, 2008.
12. T. Mossakowski, S. Autexier, and D. Hutter. Development graphs – proof management for structured specifications. *Journal of Logic and Algebraic Programming*, 67(1-2):114–145, 2006.
13. T. Mossakowski, D. Sannella, and A. Tarlecki. A simple refinement language for CASL. In Jose Luiz Fiadeiro, editor, *WADT 2004*, volume 3423 of *LNCS*, pages 162–185. Springer; Berlin, 2005.
14. Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Orna Grumberg and Michael Huth, editors, *TACAS 2007*, volume 4424 of *LNCS*, pages 519–522. Springer-Verlag Heidelberg, 2007.
15. Peter D. Mosses(Ed.). CASL *Reference Manual.* LNCS 2960 (IFIP Series). Springer, 2004.
16. Adam Pease. The SUMO challenges. `http://www.cs.miami.edu/~tptp/SUMOChallenge/`.
17. Markus Roggenbach and Lutz Schröder. Towards trustworthy specifications I: Consistency checks. In Maura Cerioli and Gianna Reggio, editors, *Recent Trends in Algebraic Specification Techniques, 15th International Workshop, WADT 2001*, volume 2267 of *Lecture Notes in Computer Science*. Springer; Berlin; http://www.springer.de, 2001.
18. D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica*, 25:233–281, 1988.