Warsaw University
Faculty of Mathematics, Informatics and Mechanics

Maciej Makowski
Album no. 189188

An Implementation of a Tool for CASL Architectural Specification Analysis

Masters thesis in
COMPUTER SCIENCE

Thesis supervised by
prof. dr hab. Andrzej Tarlecki
Institute of Informatics

August 2004

Pracę przedkładam do oceny

Data Podpis autora pracy:

Praca jest gotowa do oceny przez recenzenta

Data Podpis kierującego pracą:

Abstract

The static analysis of architectural specifications in CASL and the way it has been implemented by the author in *Heterogeneous* CASL *Tool Set* (HETS) is described. All the stages of processing the specification are presented. The algorithm for checking amalgamability conditions in CASL institution is discussed in detail.

Keywords

CASL, architectural specifications, static analysis, amalgamability

Subject classification

D.2.1 Requirement/Specification
D.2.2 Design Tools and Techniques
F.3.1 Specifying and Verifying and Reasoning about Programs

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

This thesis describes the static analysis of CASL architecural specifications as it is implemented in HETS system. The current chapter serves as a brief introduction to CASL and an overview of HETS. However, reading a thorough description of CASL (e.g. [UM04]) is strongly recommended and may be necessary for understanding more advanced CASL features.

Chapter 2 describes the formal definition of CASL semantics and the way it may be used to verify static correctness of architectural specifications. The notion of *institution* is introduced and CASL-specific amalgamability checking problem is presented. The natural formulations of amalgamability condition and cell calculus rely on category theory. For the sake of clarity and brevity more advanced category-theoretical interpretations have been ommitted and intuitive, set-theoretical interpretations are provided instead. The assumption was that the ease of comprehension should take precedence over formalism; the descriptions are quite verbose in order to make understanding the concepts presented easier. For strictly formal definition of the cell calculus and cell condition please refer to [KHTSM01]. Anyway, familiarity with basic category theory may be necessary for understanding the concepts presented in this chapter.

This thesis is based on implementation work done by the author, which includes parsing of architectural specifications, static analysis of architectural specifications and amalgamation analysis for CASL. In chapter 3 the implementation details are described. Each stage of processing the specification is treated separately, with the data structures and functions specified. Whenever a non-trivial algorithm is applied, it is illustrated using pseudo-code. The knowledge of *Haskell* programming language might be helpful, but is not required.

Similar implementation (completed as a part of CATS system) has already been described by Bartek Klin in [Klin00]. The implementation described here uses updated static semantics (defined in [RM04]) and more recent results regarding amalgamability condition (presented in [KHTSM01]).

1.1. CASL

Algebraic specification is one of the most extensively developed approaches in the area of formal methods. In this approach the program is modelled as a *many-sorted algebra* with sets, values and functions over those sets. A great deal of research on the theory and practice of algebraic specifications has been done for past 25 years. However, the proliferation of different algebraic specification languages rendered creating a common library of examples, case studies etc. impossible and therefore has been a major obstacle to adopting these techniques in the industrial context. The need for a com-

mon framework resulted in 1995 in the initiation of *The Common Framework Initiative* (CoFI)[1]. As soon as in 1997 an initial design (with language summary, abstract syntax and formal semantics but lacking concrete syntax) of *Common Algebraic Specification Language* (CASL) was proposed. The version 1.0 of CASL was released in October 1998, version 1.0.1 followed in April 2001. The current version, 1.0.2, is documented in the CASL Reference Manual [RM04].

CASL is composed of four layers; from bottom to top these are:

- *basic specifications* — contain sorts, operations, predicates and axioms which constrain the behaviour of operations and predicates;
- *structured specifications* — are formed from basic specifications, references to named specifications and instantiations of generic specifications;
- *architectural specifications* — allow to specify architecture of the software to be implemented;
- *specification libraries* — group the specifications from previous layers.

1.1.1. Basic Specifications

A *basic specification* consists of *symbol declarations* and *axioms* and *constraints* that restrict the possible *interpretations* of these symbols. The declared symbols may represent sorts, operations or predicates interpreted in models in the standard way. A subsorting relation between sorts might be specified; if s is declared to be a subsort of t (written $s < t$) this is interpreted not as an inclusion of the carrier set of s into the carrier set of t but more generally, as an *embedding*, i.e. a 1-1 function from the carrier set of s to the carrier set of t. The subsort embeddings are required to commute: subsort declarations $s < t$, $t < u$ and $s < u$ specify the existence of embeddings α, β and γ (respectively); it is then required that $\beta \circ \alpha = \gamma$. Moreover, the embedding from s to s is required to be identity. These subsort embedding compatibility requirements play important role when the possibility of combining units is introduced (1.1.3).

spec STRICT_PARTIAL_ORDER =
  sort $Elem$
  pred $\_\_ < \_\_ : Elem \times Elem$
  $\forall x, y, z : Elem$
  - $\neg(x < x)$ %(strict)%
  - $x < y \Rightarrow \neg(y < x)$ %(asymmetric)%
  - $x < y \wedge y < z \Rightarrow x < z$ %(transitive)%

  %{ Note that there may exist x, y such that neither $x < y$ nor $y < x$. **}%**

end

The specification above declares one sort named $Elem$ and a predicate $<$[2], whose intended interpretation is specified with the axioms that follow the declarations. The *semantics* of a basic specification consists of:

- a *signature* Σ containing the defined symbols and
- a *class of Σ-models* containing those interpretations of the signature that satisfy the axioms.

[1] CoFI is pronounced like 'coffee'. See `http://www.cofi.info`.

[2] Not to be confused with the subsort embedding.

1.1.2. Structured Specifications

Structured specifications are formed from basic specifications, references to named specifications and instantiations of generic specifications using following constructs:

- *translation* — renames the symbols defined in a specification;
- *hiding* — removes symbols from a specification;
- *union* — composes the specifications identifying common symbols;
- *extension* — adds further symbols and axioms to a specification;
- *free extension* — adds further symbols and axioms and restricts the models of resulting specification to free models; if the specification extended is empty, then the models of free extension are just initial models;
- *generic specification* — defines specification parameters for which arguments have to be provided whenever such specification is referenced.

spec GENERIC_MONOID [**sort** $Elem$] =
  sort $Monoid$
  ops $inj : Elem \to Monoid$
    $1 : Monoid$
    $\_\_ * \_\_ : Monoid \times Monoid \to Monoid,\ assoc,\ unit\ 1$
  $\forall x, y : Elem \bullet inj(x) = inj(y) \Rightarrow x = y$
end

spec GENERIC_COMMUTATIVE_MONOID [**sort** $Elem$] =
  Generic_Monoid [**sort** $Elem$]
  then $\forall x, y : Monoid \bullet x * y = y * x$
end

GENERIC_MONOID is a generic specification parameterised by sort $Elem$. GENERIC_COMMUTATIVE_MONOID is a (generic) extension of GENERIC_MONOID that requires the monoid operation to be commutative. The semantics of structured specifications is of the same kind as that of basic specifications.

1.1.3. Architectural Specifications

Whilst structured specification constructs allow one to write specifications in modular fashion, they are merely syntactical operations and do not impose any structure on models. In order to specify the way in which software modules should be constructed and composed during the implementation, architectural specifications are introduced.

arch spec SYSTEM_1 =
  units Ord : STRICT_PARTIAL_ORDER;
    Col : COLOR;
    Cont : ELEM → CONTAINER[ELEM];
  result Cont[Ord] **and** Cont[Col **fit** $Elem \mapsto RGB$]
end

Assuming that we have previously defined specifications STRICT_PARTIAL_ORDER (with sort $Elem$), COLOR (with sort RGB) and a generic specification CONTAINER[**sort** $Elem$] the architecture SYSTEM_1 specified above requires its model to have defined:

- a module Ord satisfying (implementing) the specification STRICT_PARTIAL_ORDER,
- a module Col satisfying the specification COLOR,
- a parameterised module Cont that, given a module with sort $Elem$, produces a container module for that sort.

Finally the **result** section describes the way those modules are composed in order to produce the complete system: container modules for Ord and Col are constructed (the Cont parameterised module expects sort $Elem$, so a mapping has to be provided saying that RGB should be the sort of data stored in resulting containers) and are amalgamated to provide one module with the functionality of both containers.

The semantics of an architectural specification reflects its structure. The component units (modules) may be regarded as unit functions (functions without arguments yield self-conained units); the entire specification is a collection of those functions together with the result of their composition according to the description in **result**.

The point that requires special attention is handling amalgamation of two units into one. The semantics requires that whenever such amalgamation takes place the amalgamated units must share common components (i.e. the interpretations of symbols from the intersection of their signatures). Moreover, subsort embeddings must also be compatible (for instance, implicit embeddings, resulting from composition of embeddings from the amalgamated units, must be defined unambiguously) in order for the amalgamation to exist. The amalgamation conditions should be checked statically, as is the case e.g. with sharing conditions in Standard ML. Amalgamation condition checking will be the main problem dealt with in this thesis.

1.2. HETS

In order to practically apply CASL specification techniques to larger projects, tools supporting CASL are being developed. The CASL *Tool Set* (CATS) developed in Bremen[3] using Standard ML supports parsing, static analysis, mixfix analysis, encoding for theorem proving tools and LaTeX formatting of CASL specifications. However, the CASL logic is not always the most convenient formalism to specify system modules, sometimes it would be better to use a domain-specific logic. Hence a desired feature would be support for *heterogeneous* CASL — a superset of CASL that allows mixing different logics in the specifications.

CATS has been designed with genericity in mind: static analysis of structured specifications has been implemented as a Standard ML functor parameterised over static analysis for arbitrary logic. Therefore extending the tool set to support new logic boils down to providing a tool for analysis of the specifications in-the-small (i.e. basic specifications) and instantiating the structured specification analysis with that tool. However, this approach presents serious limitations when heterogeneity is to be achieved: a separate instance of static analysis module would have to be created for each logic used in specification being analysed and it would not be possible to deal with these instances in an uniform way, especially when the number of required instances is not known in advance.

[3] See `http://www.informatik.uni-bremen.de/cofi/Tools/CATS.html`.

Therefore a decision has been made to move to Haskell, whose type system is more expressive; type system extensions provided by the Glasgow Haskell Compiler (GHC) are particularly useful for handling the heterogeneity in an elegant way. Present work in Bremen focuses on the *Heteregeneous Tool Set* (HETS)[4] [Moss01] — a set of tools implemented in Haskell that supports the analysis of heterogeneous CASL specifications.

1.2.1. Heterogeneous Specifications

In HETS the default logic for analysed library is determined based on the input file extension; e.g. `.casl` indicates that CASL logic should be used by default. It is possible to change the current logic within the specification using `logic` *logic_name* construct, e.g.

```
logic HasCASL
```

The specifications that follow are then treated as HASCASL specifications. The example below shows how heterogeneity can be exploited to specify a data type using CASL and process that involves this data type using CSP-CASL.

```
library Buffer

logic CASL

spec List =
  free type List[Elem] ::= nil | cons(Elem; List[Elem])
end

logic CspCASL

spec Buffer =
  data List
  channel read, write : Elem
  process read
  let Buf(l : List[Elem]) =
              read ? x -> Buf(cons(x, nil))
              [] if l = nil then STOP
                            else write ! last(l) -> Buf(rest(l))
  in Buf(nil)
end
```

1.2.2. Architecture of HETS

As shown in Figure 1.1 processing structured and architectural specifications in HETS is independent from processing the basic (in-the-small) specifications. In order to integrate new logic one has to provide a parser, static checker and a theorem prover for the desired logic. Thanks to orthogonal design of CASL, the machinery for structured and architectural specifications is generic and can be used with all the implemented logics.

[4] See `http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/`.

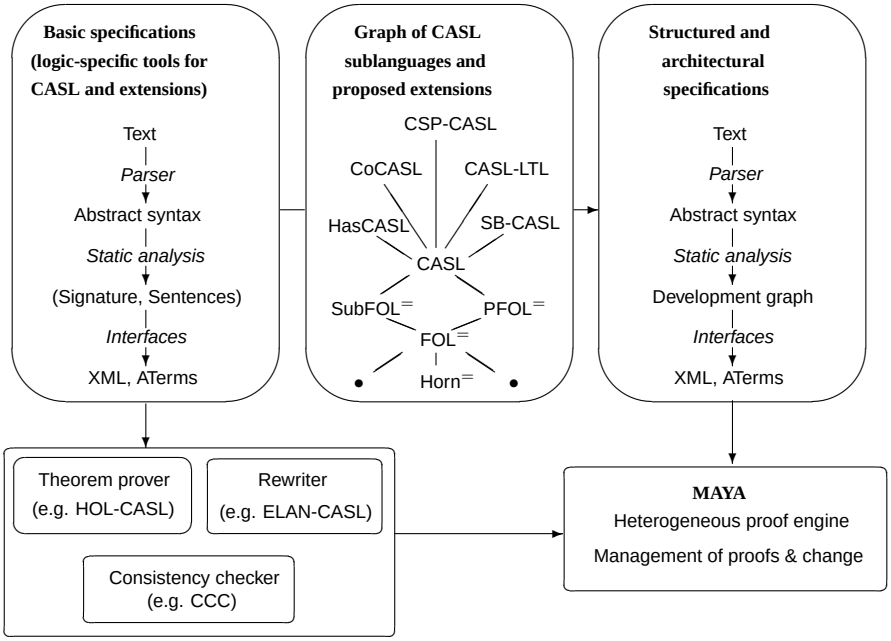

Figure 1.1: Architecture of the Heterogeneous Tool Set

Chapter 2

Static Analysis

2.1. Institution Independent Semantics

CASL is at the heart of a family of languages. Sublanguages might be obtained by imposing syntactic or semantic restrictions on the original language; the language might as well be extended to support various paradigms or applications. CASL design follows the principle of orthogonality of subsequent layers (basic, structured and architectural specifications and specification libraries), hence the sublanguages and superlanguages might be produced by restricting/extending the basic specification language only, while the other layers remain unchanged. The CASL *institution* defined by the semantics of basic specifications can be replaced by another institution for use with the higher-level specifications, particularly architectural specifications. This property is reflected by institution independent semantics style.

2.1.1. Institutions and Amalgamation

The formalisation of CASL specification semantics relies on the notion of *institution* [GB92]. An institution I is a tuple

$$(\mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models)$$

where $\mathbf{Sign}$ is a category of *signatures*, $\mathbf{Mod} : \mathbf{Sign}^{op} \to \mathbf{CAT}$ is a *model functor*, $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ is a functor that for a signature Σ gives a set of sentences over Σ and $\models$ is a family $\{\models_\Sigma\}_{\Sigma \in |\mathbf{Sign}|}$ of relations on $\mathbf{Mod}(\Sigma) \times \mathbf{Sen}(\Sigma)$, subject to so-called *satisfaction condition* (see [GB92]). $\mathbf{Mod}(\Sigma)$ is referred to as the category of Σ-*models* and for a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma_1) \to \mathbf{Mod}(\Sigma_2)$ is referred to as a *reduct functor*. $\mathbf{Mod}(\sigma)(M)$ is often written as $M|_\sigma$.

If $\mathbf{Mod}$ maps a cocone in $\mathbf{Sign}$ to a limit in $\mathbf{CAT}$, the cocone is said to be *amalgamable*. If $\mathbf{Mod}$ preserves (finite) limits, i.e. (finite) colimit cocones are amalgamable, the institution I is said to have the *(finite) amalgamation property*. The $SubPCFOL$ institution (later referred to as 'the CASL institution') that is the formalisation of the CASL underlying logic does not have the finite amalgamation property (see example in Sect. 2.2.3).

While the semantics of basic specifications is inherently institution-specific, the semantics of structured and architectural specifications is independent from the underlying institution. In the following sections the institution will be referred to implicitly when ensuring of amalgamability is required.

2.1.2. Simple and Extended Static Semantics

The CASL semantics presented in [RM04] is given in two steps. First, *static semantics* produces syntactic objects for well-formed phrases; then *model semantics* is applied to a phrase yielding a class of models of a particular language construct. While model semantics gives the ultimate meaning of phrases, it's the static semantics that forms the basis for automated static analysis.

For example, static and model semantics for unit translation

```
UNIT-TRANSLATION ::= unit-translation UNIT-TERM RENAMING
```

is defined as follows[1]:

Static semantics

$$\frac{\Gamma_s, C_s \vdash \texttt{UNIT-TERM} \triangleright \Sigma \quad \Sigma \vdash \texttt{RENAMING} \triangleright \sigma : \Sigma \to \Sigma'}{\Gamma_s, C_s \vdash \texttt{unit-translation UNIT-TERM RENAMING} \triangleright \Sigma'}$$

Model semantics

$$\frac{\begin{array}{c}\Gamma_s, C_s \vdash \texttt{UNIT-TERM} \triangleright \Sigma \\ \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow MEv \\ \Sigma \vdash \texttt{RENAMING} \triangleright \sigma : \Sigma \to \Sigma' \\ \text{for all } E \in C, \text{ there exists a unique } M' \in \mathbf{Mod}(\Sigma') \text{ with } M'|_\sigma = MEv(E)\end{array}}{\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{unit-translation UNIT-TERM RENAMING} \Rightarrow Mapping}$$

where $Mapping = \{E \mapsto M' | E \in C, M' \in \mathbf{Mod}(\Sigma'), M'|_\sigma = MEv(E)\}$

It should be noted that the static semantics given above does not discharge some conditions one would expect to be able to check statically. For instance the requirement that unique model exists boils down to ensuring model amalgamability, which is a static condition[2], since the classes of models considered are restricted only by signatures and signature morphisms. However, information about the dependencies between units needs to be preserved in order to perform a more in-depth analysis.

Sharing information is represented as a signature diagram — a dag (directed acyclic graph) whose nodes are labelled with signatures and edges are labelled with signature morphisms indicating how the source unit components are incorporated into the target unit. For a sample architectural specification

arch spec ARCH_SPEC_1 =
  units U : SPEC_1;
    F : SPEC_1 → SPEC_2;
    G : SPEC_1 → SPEC_3;
  result F[U] **and** G[U]
end

the signature diagram is shown in Figure 2.1. Please note that while actual diagram contains signatures and signature morphisms, the diagram in the figure is labelled with unit terms to indicate where the signatures and morphisms come from. F and G are represented as morphisms from SPEC_1 to SPEC_2 and from SPEC_1 to SPEC_3 respectively. They are instantiatied with U producing F[U] and G[U]. Finally the amalgamation F[U] **and** G[U] is represented by the bottom-right node.

[1] The semantics presented serves the purpose of demonstration only — it will later be compared to the extended semantics — so the exact understanding of all the symbols used is not required. For their definitions please refer to [RM04].

[2] If defined as in Sect. 2.1.1.

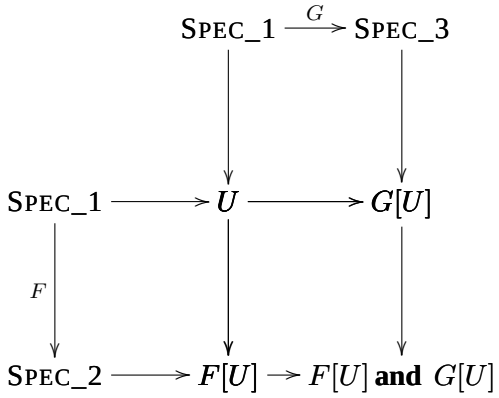

Figure 2.1: The signature diagram for ARCH_SPEC_1

The *extended static semantics* uses diagram nodes instead of plain signatures and uses the information about dependencies between units stored in the diagram to check amalgamability conditions (see Sect. 2.2.1 for a formal definition of 'ensuring amalgamability'). Interpreting diagram D as a functor from its *shape category* $\mathbf{I}$ to the category $\mathbf{Sign}$ of signatures the extended static semantics and corresponding model semantics for unit translation can be written as

Static semantics (extended)

$$\frac{\begin{array}{c}\Gamma_s, C_s \vdash \texttt{UNIT-TERM} \triangleright (p, D) \\ D(p) \vdash \texttt{RENAMING} \triangleright \sigma : D(p) \to \Sigma' \\ D \text{ ensures amalgamability along } (\Sigma', \langle \sigma : D(p) \to \Sigma' \rangle) \\ D' \text{ extends } D \text{ by new node } q \text{ and edge } e : p \to q \text{ with } D'(e) = \sigma\end{array}}{\Gamma_s, C_s \vdash \texttt{unit-translation UNIT-TERM RENAMING} \triangleright (q, D')}$$

Model semantics

$$\frac{\begin{array}{c}\Gamma_s, C_s \vdash \texttt{UNIT-TERM} \triangleright \Sigma \\ \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow MEv \\ \Sigma \vdash \texttt{RENAMING} \triangleright \sigma : \Sigma \to \Sigma'\end{array}}{\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{unit-translation UNIT-TERM RENAMING} \Rightarrow \mathit{Mapping}}$$

The extended static semantics is much more useful as far as tools and applications are concerned, as it is able to check a bigger class of specifications without resorting to theorem proving.

2.2. Amalgamability Analysis for CASL

The example presented in the previous section shows that the extended static semantics, being insititution-independent, simply states the requirement that the diagram must ensure amalgamability. This is a property dependent on the particular institution and appropriate machinery discharging this condition has to be provided for the logic used.

In this section the case of CASL institution will be analysed and a method of checking whether amalgamability is ensured for given diagram and a set of morphisms extending it will be presented. The notation introduced in the following paragraphs will also be used in Chapter 3 in the description of the implemented algorithms.

2.2.1. Signature Diagrams and Sinks

Let's denote the set of all the nodes in signature diagram D by $Nodes(D)$ and the set of all the edges in D by $Edges(D)$. We say that a family of models $\langle M_p \rangle_{p \in Nodes(D)}$ is *consistent with* D if

- for each $p \in Nodes(D)$ $M_p \in \mathbf{Mod}(D(p))$ and
- for each $e : p \to q \in Edges(D)$ we have $M_p = M_q|_{D(e)}$ — where $D(e)$ is the signature morphism with which edge e is labelled.

Consider diagram D where some nodes $p_1, p_2, ..., p_k$ labelled with signatures $\Sigma_1, \Sigma_2, ..., \Sigma_k$ ($D(p_i) = \Sigma_i$ for $i \in \{1, 2, ..., k\}$) are to be amalgamated. Let Δ stand for the signature of the amalgamation and $\tau_i : \Sigma_i \to \Delta$ denote respective signature morphisms. A *sink* τ is a pair consisting of the target signature Δ together with this family of morphisms: $(\Delta, \{\tau_1, \tau_2, ..., \tau_k\})$. We say that D *ensures amalgamability for* τ if for every family of models $\langle M_q \rangle_{q \in Nodes(D)}$ consistent with D there exists a unique model $M \in \mathbf{Mod}(\Delta)$ such that for all $i \in \{1, 2, ..., k\}$ $M|_{\tau_i} = M_{p_i}$.

The analysis of this condition is discussed in Sect. 2.2.2 – 2.2.5 and the implementation of algorithms that realise this analysis are presented in Chapter 3. The notation introduced above is used throughout the rest of this thesis.

2.2.2. Sharing Analysis

One obvious condition that must be met for the amalgamation to exist is that all the symbols (of sorts, operations or predicates) of the component units that are identified in the the signature of the amalgamation must share the same interpretation. This condition is easily decidable through the analysis of the signature diagram.

Let $Symbs(D)$ denote the disjoint union of all the symbols[3] from $\Sigma_1, \Sigma_2, ..., \Sigma_k$, that is a set of pairs (p_i, s) where s is one of the symbols of Σ_i. Let *morphism path* $\langle \sigma \rangle$ be a sequence of signature morphisms $\sigma_1, \sigma_2, ..., \sigma_m$ determined by a path $\langle e \rangle = e_1, e_2, ..., e_m$ in D where $\sigma_i = D(e_i)$. Note that $cod(\sigma_i) = dom(\sigma_{i+1})$ for $i \in \{1, 2..., m-1\}$. We write $dom\langle \sigma \rangle$ to denote the domain of the first morphism in the sequence and $cod\langle \sigma \rangle$ to denote the codomain of the last morphism in $\langle \sigma \rangle$. We also say that $\langle \sigma \rangle$ maps a symbol s from $dom\langle \sigma \rangle$ to symbol t from $cod\langle \sigma \rangle$ if the composition of morphisms in $\langle \sigma \rangle$ maps s to t. Let's define two equivalence relations on $Symbs(D)$, $\simeq$ and $\simeq_\tau$, in the following way:

- $\simeq$ is the least equivalence satisfying the following condition: $(p, s) \simeq (q, t)$ if there exists a node r and two paths in the diagram: $\langle e \rangle$ from r to p and $\langle f \rangle$ from r to q such that signature morphism path $\langle \sigma \rangle$ determined by $\langle e \rangle$ maps symbol u from $D(r)$ to s from $D(p)$ and morphism path $\langle \mu \rangle$ determined by $\langle f \rangle$ maps the same u to t from $D(q)$. Informally, this rule states that s and t have a common origin, thus denote essentially the same thing. The rule obviously is reflexive and symmetrical, so $\simeq$ is just the transitive closure of the relation given by this rule.
- $(p_i, s) \simeq_\tau (p_j, t)$ iff τ_i maps s to a symbol u in Δ and τ_j maps t to the same u. Informally, $(p_i, s) \simeq_\tau (p_j, t)$ means that s and t are identified in the amalgamation.

We may now formulate a condition regarding sharing, which is a necessary prerequisite for amalgamation:

[3] In case of operation and predicate symbols we consider the names together with their profiles; e.g. two operations named f but having different profiles are regarded as different symbols.

Theorem 1 *If the condition*

$$(*)\ \simeq_\tau \subseteq \simeq$$

holds then all the symbols that are identified in the amalgamation have common interpretations in models from model families consistent with D.

Proof: Assume that the sharing condition $(*)$ holds. From the definition of $\simeq_\tau$ it follows that for each pair of identified diagram symbols s, t from nodes p_i, p_j we have $(p_i, s) \simeq_\tau (p_j, t)$. Because of $(*)$ we also have $(p_i, s) \simeq (p_j, t)$. There are two possible ways in which these diagram symbols got to be in the $\simeq$ relation:

1. either s and t have common origin — in this case the theorem holds trivially, since the origin determines the interpretation of both s and t,
2. or s and t are equivalent w.r.t. $\simeq$ due to transitive closure of the rule that defines $\simeq$ — this is illustrated in Figure 2.2. Here symbol s from $D(p_i)$ has common origin with some symbol

Figure 2.2: Transitive closure case in Theorem 1

u_1 from $D(q_1)$, u_1 has common origin with some u_2 from $D(q_2)$ etc.; finally u_m from $D(q_m)$ and t have common origin. It follows by induction that s, all the u_i's and finally t must share common interpretation.

□

2.2.3. Cell Calculus

With CASL institution the aspect that turns out to be most complex and ultimately makes the amalgamability analysis undecidable is the presence of sort embeddings. Consider following architectural specification [KHTSM01]:

spec NUM_1 = **sorts** $Number$, $List[Digit]$, $List[Number]$
spec NUM_2 = **sorts** $Number < List[Digit]$, $List[Number]$
spec NUM_3 = **sorts** $Number < List[Number]$, $List[Digit] < List[Number]$

arch spec ARCH_SPEC_1 =
  units U : NUM_1;
    F : NUM_1 → NUM_2;
    G : NUM_1 → NUM_3;
  result F[U] **and** G[U]
end

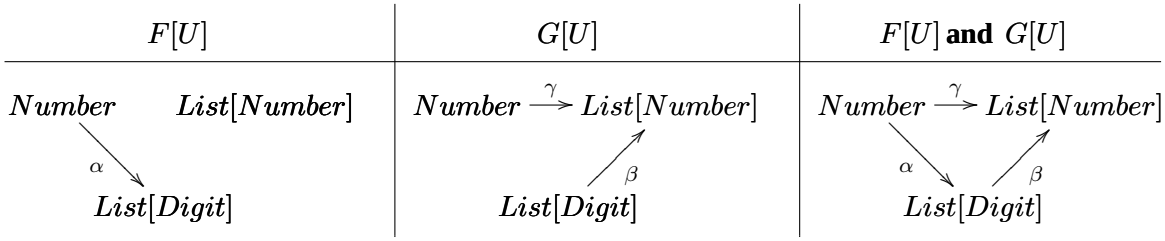

Figure 2.3: The sort embeddings in F[U], G[U] and F[U] **and** G[U]

Here the sharing condition for sorts is satisfied, since all the sorts come from the unit U. However, sort embeddings α in F[U] and β, γ in G[U] are defined independently, so there is no guarantee that in the amalgamation the embeddings $\beta \circ \alpha$ and γ will commute (see Figure 2.3). Indeed, under expected implementation[4] these injections do not commute. This relates to the lack of amalgamation property in the CASL institution: the pushout of the signature diagram (i.e. the union of F[U] and G[U] signatures) is not mapped by $\mathbf{Mod}$ to a pullback in $\mathbf{CAT}$.

It seems that similar sharing condition to that specified above for signature objects has to be checked: it must be ensured that the amalgamation does not assume equality between more embeddings than can be deduced from the diagram. However, as the example above shows, not only individual embeddings declared explicitly would have to be analysed, but also their compositions (*embedding paths*). That leads to undecidability, since there might exist embedding loops that generate infinite number of possible paths[5].

Finiteness aside, the problem might be formalised in a fashion similar to the sharing condition, bearing in mind that the rules according to which two embedding paths are considered to be equivalent are somewhat more complex than in case of $\simeq$ and $\simeq_\tau$ relations. An *embedding path* in diagram D is a sequence of embeddings $\alpha_1, \alpha_2, ...\alpha_m$ such that the domains and codomains of consecutive embeddings are equivalent w.r.t. supplied $\simeq$ relation (on the disjoint union of sort symbols from D). We define a *cell* to be a pair of embedding paths with equivalent beginning sorts and equivalent ending sorts; the paths in the cell are considered to be equivalent. We will need to show that the set of cells generated by the diagram is a superset of the set of cells induced by the amalgamation.

The considerations that follow assume that the only symbols in the signatures are sort symbols; operations and predicates are irrelevant for sort embedding analysis and would only clutter up the notation. First, consider the simple extension of $\simeq$ to all the sorts in the diagram: the least equivalence on the disjoint union of the sort symbols in D such that for all diagram edges $e : p \to q$ and sort s in $D(p)$, $(p, s) \simeq (q, D(e)(s))$. Let $Embs(D)$ be the disjoint union of all the sort embeddings defined in the signatures in D, i.e. a set of pairs (p, α) where α is an embedding in $D(p)$. The embedding paths mentioned above are formally defined as finite words over $Embs(D)$ in form $\langle (p_n, \alpha_n), ..., (p_1, \alpha_1) \rangle$ such that for $1 \leq i < n$ $(p_i, cod(\alpha_i)) \simeq (p_{i+1}, dom(\alpha_{i+1}))$. Let's denote the set of all such words

[4] $n : Number$ is treated as a list of digits and as a one-element long list of numbers, $ld : List[Digit]$ is regarded as a list of numbers of the same length as ld where each number is one-digit long.

[5] Infinite number of paths doesn't necessarily mean the problem presented is undecidable. However, the rules of cell calculus directly correspond to the axioms of monoids with left cancellation and the (undecidable) problem of deciding whether such a monoid defined by a finite set of equations is trivial can be reduced to the cell condition presented below (see [KHTSM01]).

by $Adm_{\simeq}$[6]. Note that the first morphism in a path (in the diagrammatic order) is the rightmost letter in the word. For a word $\omega = \langle (p_n, \alpha_n), ..., (p_1, \alpha_1) \rangle$ we write $dom(\omega)$ for $(p_1, dom(\alpha_1))$ and $cod(\omega)$ for $(p_n, cod(\alpha_n))$. Now, a *cell over* D may be defined as a pair (ω, υ) of words from $Adm_{\simeq}$ such that $dom(\omega) \simeq dom(\upsilon)$ and $cod(\omega) \simeq cod(\upsilon)$.

Let $\cong$ be the least relation on $Adm_{\simeq}$ satisfying following rules:

$$\textbf{(Refl)}\ \frac{\omega \in Adm_{\simeq}}{\omega \cong \omega} \qquad \textbf{(Symm)}\ \frac{\omega \cong \upsilon}{\upsilon \cong \omega} \qquad \textbf{(Trans)}\ \frac{\omega \cong \upsilon \qquad \upsilon \cong \psi}{\omega \cong \psi}$$

$$\textbf{(Diag)}\ \frac{l : p \to q \text{ is an edge in } D}{\langle (p, e) \rangle \cong \langle (q, D(l)(e)) \rangle} \qquad \textbf{(Comp)}\ \frac{ed \text{ is defined in } D(p)}{\langle (p, e), (p, d) \rangle \cong \langle (p, ed) \rangle}$$

$$\textbf{(Cong)}\ \frac{\omega \cong \upsilon \qquad \psi \cong \phi \qquad cod(\omega) \simeq dom(\psi)}{\psi\omega \cong \phi\upsilon} \qquad \textbf{(Lc)}\ \frac{\psi\omega \cong \psi\upsilon}{\omega \cong \upsilon}$$

Obviously (due to **(Refl)**, **(Symm)** and **(Trans)** rules) $\cong$ is an equivalence relation. $\cong$ can be thought of as a set of cells over D, the seven rules used to inductively define this set are therefore called the *cell calculus.*

In this way we have defined the relation that represents the expected equivalences between embedding paths. What still needs to be done is to represent the paths that are considered equivalent in the amalgamation, that is paths that have equivalent beginnings and ends in the amalgamation. Let $Adm_{\simeq_\tau}$ be the set of all the words defined in the same manner as $Adm_{\simeq}$ but where the domains and codomains of consecutive morphisms are equivalent w.r.t. $\simeq_\tau$ (and so $Adm_{\simeq_\tau}$ are words of embeddings in signatures $\Sigma_1, \Sigma_2, ..., \Sigma_k$). We may now define $\cong_\tau$ to be the least equivalence on $Adm_{\simeq_\tau}$ where for $\omega, \upsilon \in Adm_{\simeq_\tau}$ if $dom(\omega) \simeq_\tau dom(\upsilon)$ and $cod(\omega) \simeq_\tau cod(\upsilon)$ then $\omega \cong_\tau \upsilon$. The relation $\cong_\tau$ represents embedding paths that are identified in the amalgamation. Note that if the sharing condition $(*)$ holds then $Adm_{\simeq_\tau} \subseteq Adm_{\simeq}$.

Theorem 2 *In the above setting, if both the sharing condition* $(*)$ *and the cell condition:*

$$(**)\ \cong_\tau \subseteq \cong$$

hold then D *ensures amalgamability for* τ.

Proof: See the extended version of [KHTSM01]. □

2.2.4. Optimisations

The rules of cell calculus presented in Sect. 2.2.3 capture the formal aspect of amalgamability analysis; however, using them naively is inefficient as far as the implementation is concerned. In particular, the rules of extended static semantics often generate a number of diagram nodes with similar signatures and, as a consequence, the $\simeq$ relation has large equivalence classes. This leads to large $Adm_{\simeq}$ set, since embeddings with equivalent domains and codomains are equivalent and all combinations of

[6] $Adm_{\simeq}$ stands for $\simeq$-*admissible*; the words are constructed based on the $\simeq$ relation.

equivalent embeddings need to be used in order to obtain the set of all $\simeq$-admissible words. Large $Adm_{\sim}$ coupled with expensive algorithm for computing $\cong$ relation results in long execution time.

A way to remedy this is to observe that **(Diag)** rule of cell calculus corresponds to a trivial equivalence of embeddings. We may therefore compute an equivalence $\sim$ on $Embs(D)$ defined by this rule and factor $Embs(D)$ through $\sim$. Let $CanonicalEmbs(D)$ denote the set $Embs(D)/\sim$. For checking the cell condition it is now sufficient to consider the set of $\simeq$-admissible words over $CanonicalEmbs(D)$. The argument is that any equivalence proof $\omega \cong \upsilon$ in the cell calculus can be normalised to a proof $\omega \cong \phi \cong \psi \cong \upsilon$, where ϕ and ψ are $\simeq$-admissible words over $CanonicalEmbs(D)$. $\phi \cong \psi$ is proved using words over $CanonicalEmbs(D)$ only. However, this proof requires a slightly different set of rules; instead of **(Comp)** and **(Diag)** rules we use a combination of these:

$$\textbf{(CompDiag)}\frac{(p,e)\sim(p,e') \qquad (q,d)\sim(n,d') \qquad (m,f)\sim(n,e'd')}{\langle(p,e),(q,d)\rangle \cong \langle(m,f)\rangle}$$

Now the equivalences $\omega \cong \phi$ and $\psi \cong \upsilon$ are proved through obvious applications of **(Diag)** and **(Cong)** rules only. A further refinement is to close the $\sim$ relation w.r.t. congruence rule for single embeddings, namely:

$$\textbf{(Cong1)}\frac{(p,e)\sim(q,e') \qquad (p,d)\sim(q,d') \qquad (p,ed) \text{ and } (q,e'd') \text{ are in the domain of } \sim}{(p,ed)\sim(q,e'd')}$$

This reduces the set $CanonicalEmbs(D)$ even more, while the argument above remains valid.

The implementation described in Chapter 3 uses this optimisation, i.e. $\cong$ is computed over $CanonicalEmbs(D)$ and $\cong_\tau$ is translated to $CanonicalEmbs(D)$.

2.2.5. Checking the Cell Condition

The domains of $\cong$ and $\cong_\tau$ may be infinite and while we could find a finite subrelation $\cong'_\tau$ of $\cong_\tau$ such that if $\cong'_\tau \subseteq \cong$ then $\cong_\tau \subseteq \cong$, (it is sufficient to consider loopless words in the domain of $\cong_\tau$[7]) in general it is not possible to finitely represent $\cong$.

Anyway, attempts have been made in [KHTSM01] to distinguish special cases where the cell condition can be effectively analysed.

1. When the set $Adm_{\sim}$ is finite (i.e. embeddings do not form non-trivial cycles) the relation $\cong$ is finite. Computing $\cong$ is PSPACE-hard, however, an observation that if the inclusion $\simeq_\tau \subseteq \simeq$ holds and the colimit of D (as a diagram in the category of left cancellable small categories[8]) is a thin category than the cell condition is satisfied leads to a polynomial time algorithm.

2. Consider a subrelation $\cong_0$ of $\cong$ that is the least equivalence satisfying all the rules of cell calculus except for **(Cong)** and **(Lc)**. Obviously $\cong_\tau \subseteq \cong_0$ implies the cell condition. Although $\cong_0$ has possibly infinite domain it has only finitely many non-reflexive elements and can therefore be finitely represented.

3. Let $\cong^R$ denote the relation $\cong$ restricted to loopless words. The domain of $\cong^R$ is finite, thus the relation itself is finite, so it is possible to devise a (PSPACE-hard) algorithm that checks the inclusion $\cong_\tau \subseteq \cong^R$.

[7] Note that the identity embeddings are essentially trivial w.r.t. $\cong$ and $\cong_\tau$, we can therefore ommit them in further considerations.

[8] We take the category $\mathbf{lcCat}$ of left cancellable small categories to be the category of signatures (we interpret preorders as thin categories), hence the diagram is a fuctor $D : \mathbf{I} \to \mathbf{lcCat}$.

These observations form the basis for an algorithm that is able to verify the cell condition for most of practically useful cases of architectural specifications. The least trivial part of the implementation presented in the next chapter is entirely based on these results.

Chapter 3

The Implementation

3.1. Module Overview

The source code of HETS is split into Haskell modules. The hierarchy of modules is reflected by the hierarchy of directories in which the source files are stored, i.e. the module `Static.ArchDiagram` corresponds to the file `Static/ArchDiagram.hs` in the HETS source code directory.

The modules that form the foundations of HETS are:

- `Logic.*` — provide data structures for logics and logic comorphisms and define Grothendieck logic — a heterogeneous logic over which the data structures and algorithms for specifications in-the-large are built.

- `Comorphisms.*` — define comorphisms between different logics implemented in HETS and assemble all these logics into the *logic graph.*

- `Syntax.*` — define the abstract syntax and parsers for CASL structured and architectural specifications and specification libraries.

- `Static.*` — contain the static analysis algorithms for specifictaions in-the-large.

In addtition there are logic-specific modules in `CASL.*`, `HasCASL.*`, `CspCASL.*` etc.; a separate directory is provided for each logic supported by HETS. A library of common data structures (maps, graphs etc.) is provided by modules in `Common.Lib.*`.

The modules implemented as a part of this thesis are:

- `Syntax.Parse_AS_Architecture` — parsing of architectural specifications,

- `Static.ArchDiagram` — definition of signature diagrams,

- `Static.AnalysisArchitecture` — institution independent static analysis of architectural specifications,

- `CASL.Amalgamability` — amalgamability analysis for CASL institution.

3.2. Parsing

Parsing in HETS is done using *monadic parser combinator* [HM96] library Parsec[1]. The abstract syntax elements for architectural specifications are defined in `Syntax.AS_Architecture`. Functions building the *abstract syntax tree* (AST) are defined in `Syntax.Parse_AS_Architecture`. They follow the abstract syntax defined in [RM04]; the grammar has only been slightly adjusted to eliminate left-hand-side recursion and disambiguate some constructs. There's one function for each non-terminal symbol of the grammar.

```
-- | Parse group unit term
-- @
-- GROUP-UNIT-TERM ::= UNIT-NAME
--                   | UNIT-NAME FIT-ARG-UNITS
--                   | { UNIT-TERM }
-- @
groupUnitTerm :: (AnyLogic, LogicGraph) -> AParser UNIT_TERM
groupUnitTerm l =
        -- unit name/application
    do name <- simpleId
       (args, pos) <- fitArgUnits l
       return (Syntax.AS_Architecture.Unit_appl name args pos)
    <|> -- unit term in brackets
    do lbr <- asKey "{"
       ut <- unitTerm l
       rbr <- asKey "}"
       return (Syntax.AS_Architecture.Group_unit_term ut
                                  (map tokPos [lbr, rbr]))
```

The function presented above parses a group unit term. The `<|>` symbol is a Parsec alternative that separates productions. The first two productions are captured in the first part of the alternative — $fitArgUnits$ may return an empty list and hence yield unit name. The code indeed looks very similar to the context-free grammar.

Special care needs to be taken when parsing unit specifications:

```
UNIT-SPEC ::= SPEC-NAME
            | UNIT-ARGS GROUP-SPEC
            | arch spec GROUP-ARCH-SPEC
            | closed UNIT-SPEC
```

The first two cases are indistinguishable at the parsing stage, since `UNIT-ARGS` might be empty and `GROUP-SPEC` might reduce to `SPEC-NAME`. Therefore in either of these cases the $unitSpec$ function parses a unit type (i.e. tries the second production). At the static analysis stage the correct production is regonised depending on the context and appropriate correction to the AST is made.

3.3. Static Analysis

The (institution independent) static analysis of architectural specifications is based on the extended static semantics defined in [RM04] (see Sect. 2.1.2).

[1] See `http://www.cs.uu.nl/~daan/parsec.html`.

3.3.1. Data Structures

Data structures used in the analysis are defined in `Static.ArchDiagram`. The *development graphs* defined in `Static.DevGraph` are used as well. Both development graphs and signature diagrams are represented using $Graph$ data type defined in `Common.Lib.Graph`.

The $DGraph$ type defined in `Static.DevGraph` is a graph whose nodes are labelled with signatures and edges are labelled with signature morphisms. It represents the dependencies between structured specifications: the leaves represent basic specifications and the inner nodes are structured specifications constructed from other specifications. The development graph also contains *theorem links* — edges to which proof obligations are attached. The $NodeSig$ type is a representation of signature that takes into account the development graph. $NodeSig$ contains both a node in development graph and a signature with which that node is labelled. The operations on $DGraph$ defined in `Static.DevGraph` are:

- $extendDGraph$ — extends given $DGraph$ with given morphism originating from given node.
- $extendDGraphRev$ — extends given $DGraph$ with given morphism whose codomain is in given node.
- $nodeSigUnion$ — extends given $DGraph$ with a node representing union of signatures from given set of nodes.

The $Diag$ type defined in `Static.ArchDiagram` is a graph whose nodes are labelled with $NodeSigs$ and edges are labelled with signature morphisms. The signature stored in a node is the signature of the unit that node represents. The operations on $Diag$ defined in `Static.ArchDiagram` are:

- $extendDiagram$ — extends given $Diag$ with a node containing given $NodeSig$ and an edge from given node to the new node. The edge is labelled with supplied signature morphism.
- $extendDiagramIncl$ — extends given $Diag$ with a node containing given $NodeSig$ and a set of edges from given set of nodes to the new node. The edges are labelled with signature inclusions.
- $extendDiagramWithMorphism$ — extends given $Diag$ with an edge from given node and a new node that is the target of the edge added. The edge is labelled with given signature morphism and the new node contains the codomain of this morphism.
- $extendDiagramWithMorphismRev$ — similar to $extendDiagramWithMorphism$, but the new node is at the begginning of the new edge and contains the domain of the morphism.
- $homogeniseDiagram$ — tries to coerce the logics of all the signatures and morphisms in given $Diag$ to given logic using coertion function $rcoerce$ from `Logic.Logic`.
- $homogeniseSink$ — tries to coerce the logics of all the morphisms in given list of signature morphisms to given logic using $rcoerce$ function from `Logic.Logic`.

3.3.2. Functions

Functions that perform the analysis are defined in `Static.AnalysisArchitecture`. There is essentially one function for each semantic rule. The only exported functions are ana_ARCH_SPEC and ana_UNIT_SPEC; given logic graph, global context (containing current development graph),

current logic and appropriate fragment of the abstract syntax tree these functions return architectural/unit signature, modified development graph and modified AST fragment.

The checking of amalgamability condition is realised by $assertAmalgamability$ and $homogeneousEnsuresAmalgamability$ functions. Given a diagram and a sink they homogenise the diagram and the sink by coercing the logic of each signature and each morphism to the logic of the target of the sink and then passing it to the $ensures\_amalgamability$ function for this logic. $ensures\_amalgamability$ is defined as a part of $StaticAnalysis$ type class, to which all the logics implemented in HETS belong.

3.4. Amalgamability Analysis for CASL

The amalgamability checking for CASL (specific for the CASL institution) is based on the algorithm presented in [KHTSM01]. All the code responsible for amalgamability analysis is in `CASL.Amalgamability`.

3.4.1. Data Structures

The data structures used extensively throughout the amalgamabilty checking algorithm are equivalence relations. Two types corresponding to two different representations of these have been defined.

- $EquivRel\ a$ is a list of equivalence classes ($[EquivClass\ a]$). $EquivClass\ a$ is a list of elements of type a. This is the representation commonly used in the algorithm, particularly when inclusion of one equivalence into another is to be checked.

- $EquivRelTagged\ a\ b$ is a list of pairs where the first element has type a and the second has type b ($[(a, b)]$). The first element of a pair is interpreted as an element of relation domain, the second is an equivalence class tag. Pairs with equal second elements represent elements of the same equivalence class. This representation is used while computing relations — it makes iterating through the elements of relation domain easy, since merging equivalence classes does not change the order of elements in the list.

$CASLDiag$ is defined as a diagram (graph) whose nodes are labelled with CASL signatures ($CASLSign$) and edges are labelled with CASL signature morphisms ($CASLMor$). Types have been defined as well for elements of disjoint unions of sorts, operations, predicates and sort embeddings from the diagram; these are $DiagSort$, $DiagOp$, $DiagPred$ and $DiagEmb$ respectively. $DiagEmbWord$ is a list of $DiagEmbs$.

3.4.2. Auxiliary Functions

The operation commonly performed throughout the algorithm is computing equivalence relations. It is usually done according to Algorithm 1.

Algorithm 1: Computing an equivalence relation

Data : • a set E of elements
• a set R of rules
Result : the least equivalence on E satisfying all the rules in R represented as $EquivRel$

1 $rel \leftarrow$ the least equivalence on E represented as $EquivRelTagged$
foreach $mergeRule \in R$ **do**
2 merge the equivalence classes in rel according to $mergeRule$
3 convert rel from $EquivRelTagged$ to $EquivRel$
return rel

Computing the equivalence in line **1** is straightforward: based on set E (represented as a list) we need to create a list containing for each element $e \in E$ a pair (e, e). This list would represent a relation where each element is in a different equivalence class, hence is in relation only with itself (see 3.4.1 for explanation how an equivalence relation is represented using $EquivRelTagged$). The conversion in line **3** is performed by $taggedValsToEquivClasses$ function; the opposite conversion is also possible and is done by $equivClassesToTaggedVals$. Each element of R is a function that given two elements $e_1, e_2 \in E$ returns $True$ if e_1 and e_2 should be in relation according to the rule it represents and $False$ otherwise. Merging of equivalence classes from line **2** is performed by $mergeEquivClassesBy$ function (described in Algorithm 2). Note, that $mergeRule$ passed to Algorithm 2 does not depend on the current state of the relation whose closure is being generated, hence it is possible to compute the closure in one pass, without resorting to fixpoint computation.

Algorithm 2: $mergeEquivClassesBy$ function

Data : • a function $mergeRule : E \to E \to Bool$
• a relation rel represented as $EquivRelTagged\ E\ T$
Result : the least equivalence rel' satisfying $mergeRule$ such that $rel \subseteq rel'$

foreach $(e_1, t_1) \in rel$ **do**
  foreach $(e_2, t_2) \in rel$ **do**
    if $mergeRule\ e_1\ e_2$ **then**
      $rel \leftarrow mergeEquivClasses\ rel\ t_1\ t_2$
return rel

Algorithm 3: $mergeEquivClasses$ function

Data : • a relation rel represented as $EquivRelTagged\ E\ T$
• a tag $t_1 : T$
• a tag $t_2 : T$

Result : the relation rel with equivalence classes represented by t_1 and t_2 merged

foreach $(e, t) \in rel$ **do**
  if $t = t_2$ **then** replace (e, t) by (e, t_1) in rel

return rel

Another operation performed often is checking whether an inclusion between two relations holds. This is done by $subRelation$ function described in Algorithm 4.

Algorithm 4: $subRelation$ function

Data : • an equivalence sub represented as $EquivRel$
• an equivalence sup represented as $EquivRel$

Result : $Just(e_1, e_2)$ where $(e_1, e_2) \in sub$ and $(e_1, e_2) \notin sup$ or $Nothing$ if $sub \subseteq sup$

foreach *equivalence class* $eqcl_{sub}$ *from* sub **do**
  $e_1 \leftarrow$ the first element of $eqcl_{sub}$
  if *there exists an equivalence class in* sup *containing* e_1 **then**
    $eqcl_{sup} \leftarrow$ the equivalence class from sup containing e_1
  else
    $eqcl_{sup} \leftarrow$ an empty list
  foreach e_2 *from* $eqcl_{sub}$ **do**
    if $e_2 \notin eqcl_{sup}$ **then return** $Just(e_1, e_2)$

return $Nothing$

3.4.3. Functions

The main function in the module is $ensuresAmalgamability$. It performs the amalgamability check based on the algorithm outlined in [KHTSM01] and is a realisation of the analysis presented in Sect. 2.2; for the meaning of symbols used in Algorithm 5 please refer to that section. Note that $\simeq$ (and, correspondingly, $\simeq_\tau$) have been split to three relations: $\simeq$ for sorts, $\simeq^{Op}$ for operations and $\simeq^{Pred}$ for predicates. Subsort embeddings are dealt with using $\cong$ and $\cong_\tau$ relations.

Algorithm 5: $ensuresAmalgamability$ function

Data : • diagram D
• sink τ — a list of pairs (p_i, τ_i), where p_i is a node in D and $\tau_i : D(p_i) \to \Sigma$ is a signature morphism; Σ is common for all the morphisms in the sink

Result : Yes, No or $DontKnow$, saying whether D ensures amalgamability for τ

```
1  compute ≃ for D and ≃_τ for D, τ
   if ≃_τ ⋢ ≃ then return No
   else
2      compute ≃^Op for D and ≃_τ^Op for D, τ
       if ≃_τ^Op ⋢ ≃^Op then return No
       else
3          compute ≃^Pred for D and ≃_τ^Pred for D, τ
           if ≃_τ^Pred ⋢ ≃^Pred then return No
           else
4              compute ≅_τ for τ, ≃_τ
5              ≅_τ^0 ←- ≅_τ restricted to non-reflexive elements
6              compute ≅_0 for D, ≃
               if ≅_τ^0 ⊆ ≅_0 then return Yes
               else
                   compute Embs(D) for D
7                  compute ∼ for D, Embs(D)
8                  ≅_τ^C ←- ≅_τ translated to canonical embeddings using ∼
9                  if Adm_∼ is finite then
10                     compute Adm_∼ for Embs(D), ≃
11                     if colimit of D is thin (for ≃, Embs(D), ≅_0) then return Yes
                       else
12                         compute CanonicalEmbs(D) for ∼
13                         compute Adm_∼^C for CanonicalEmbs(D), ≃
14                         compute ≅ for Adm_∼^C, ≃, ∼
                           if ≅_τ^C ⊆ ≅ then return Yes
                           else return No
                   else
15                     compute ≅^R for D, ≃, ∼
                       if ≅_τ^C ⊆ ≅^R then return Yes
                       else return DontKnow
```

First (line **1** of Algorithm 5), equivalences $\simeq$ and $\simeq_\tau$ need to be computed. This is performed by $simeq$ and $simeq\_tau$ respectively (Algorithms 6 and 7). Computations of $\cdot^{Op}$ and $\cdot^{Pred}$ variants (lines **2** and **3** in Algorithm 5) of $\simeq$ and $\simeq_\tau$ are done in exactly the same way.

Algorithm 6: $simeq$ function

Data : diagram D
Result : the $\simeq$ relation represented as $EquivRel\ DiagSort$

$Sorts(D) \leftarrow$ the set of all the sorts in D
$rel \leftarrow$ the least equivalence on $Sorts(D)$ represented as $EquivRelTagged$
$mergeCond \leftarrow \lambda s.\lambda s'$. "there exists morphism in D that maps s to s' or s' to s"
$rel \leftarrow mergeEquivClassesBy\ mergeCond\ rel$
$rel \leftarrow taggedValsToEquivClasses\ rel$
return rel

Algorithm 7: $simeq\_tau$ function

Data : • the diagram D
• the sink τ — a list of pairs (p_i, τ_i), where p_i is a node in D and $\tau_i : D(p_i) \to \Sigma$ is a signature morphism; Σ is common for all the morphisms in the sink
Result : the $\simeq_\tau$ relation represented as $EquivRel\ DiagSort$

$rel \leftarrow$ an empty list
foreach $(p_i, \tau_i) \in \tau$ **do**
  foreach *sort* s *from* $D(p_i)$ **do**
    $t \leftarrow \tau_i(s)$
    add a pair $((p_i, s), t))$ to rel
$rel \leftarrow taggedValsToEquivClasses\ rel$
return rel

The next point after checking the sharing condition is computing $\cong_\tau$, $\cong^0_\tau$ and $\cong_0$ (lines **4-6**). $\cong_\tau$ is computed by $cong\_tau$ function (Algorithm 8).

Algorithm 8: $cong\_tau$ function

Data : • the diagram D
• the sink τ — a list of pairs $(p_i, \tau_i : D(p_i) \to \sigma)$
• the $\simeq_\tau$ relation represented as $EquivRel\ DiagSort$
Result : the $\cong_\tau$ relation represented as $EquivRel\ DiagEmbWord$

$Embs(\tau) \leftarrow$ the set of all pairs (p, e) such that p is a node in one of the pairs from τ and e is an embedding in $D(p)$
$W \leftarrow loopless Words\ Embs(\tau)\ \simeq_\tau$
$rel \leftarrow$ the least equivalence on W
$mergeCond \leftarrow \lambda\omega.\lambda\upsilon.dom(\omega) \simeq_\tau dom(\upsilon)$ and $cod(\omega) \simeq_\tau cod(\upsilon)$
$rel \leftarrow mergeEquivClassesBy\ mergeCond\ rel$
$rel \leftarrow taggedValsToEquivClasses\ rel$
return rel

In order to obtain $\cong^0_\tau$ it's sufficient to filter out these equivalence classes from $\cong_\tau$ that have just one element. Generating $\cong_0$ boils down to building the least equivalence on one-letter words (i.e. single embeddings), merging equivalence classes according to **(Diag)** rule and finally closing the relation w.r.t. **(Comp)** rule by generating all the two-letter words and inserting them into appropriate equivalence classes. This is illustrated in Algorithm 9. The $diagRule$ in line **1** is a (simple to construct) functional representation of the **(Diag)** rule.

Algorithm 9: $cong\_0$ function

Data : • the diagram D
• the $\simeq$ relation represented as $EquivRel\ DiagSort$
Result : the $\cong_0$ relation represented as $EquivRel\ DiagEmbWord$

$Embs(D) \leftarrow$ the set of all sort embeddings in D
$W \leftarrow$ the set of all one-letter words over $Embs(D)$
$rel \leftarrow$ the least equivalence on W
1 $rel \leftarrow mergeEquivClassesBy\ diagRule\ rel$
$rel \leftarrow taggedValsToEquivClasses\ rel$
$W_2 \leftarrow$ the set of all $\simeq$-admissible two-letter words over $Embs(D)$
foreach $\omega \in W_2$ **do**
  foreach $eqcl \in rel$ **do**
    $\upsilon \leftarrow$ the first word in $eqcl$
    if $dom(\omega) = dom(\upsilon)$ **and** $cod(\omega) = cod(\upsilon)$ **then**
      add ω to $eqcl$
      break
return rel;

Computing $\sim$ (line **7** in Algorithm 5) boils down to applying the **(Diag)** rule of cell calculus to $Embs(D)$ and closing the resulting relation w.r.t **(Cong1)** rule (see Sect. 2.2.4). This is done in function sim presented in Algorithm 10. $congruenceClosure$ function from line **1** is described in Algorithm 11.

Algorithm 10: sim function

Data : • a diagram D
• a set $Embs$ of sort embeddings represented as a list $[DiagEmb]$

Result : the relation $\sim$ represented as $EquivRel\ DiagEmb$

$rel \leftarrow$ the least equivalence on $Embs$
$rel \leftarrow mergeEquivClassesBy\ diagRule\ rel$
$check \leftarrow \lambda(p,e).\lambda(q,d).$"$p = q$ and $Embs$ contains a pair (p, ed)"
$op \leftarrow \lambda(p,e).\lambda(q,d).(p, ed)$
repeat
1 | $rel \leftarrow congruenceClosure\ check\ op\ rel$
until *fixpoint on* rel *is reached*
$rel \leftarrow taggedValsToEquivClasses\ rel$
return rel

Algorithm 11: $congruenceClosure$ function

Data : • $check$: $E \to E \to Bool$ — a function that checks rule prerequisite
• op: $E \to E \to E$ — a function combining two elements
• a relation rel represented as $EquivRelTagged\ E\ T$

Result : a relation rel' obtained from rel by applying the **(Cong)** rule once to each combination of elements in rel

foreach *pair* $(\omega, t_\omega) \in rel$ **do**
  foreach *pair* $(\upsilon, t_\upsilon) \in rel$ *such that* $t_\omega = t_\upsilon$ **do**
    foreach *pair* $(\psi, t_\psi) \in rel$ *such that (*$check\ \omega\ \psi$*)* **do**
      foreach *pair* $(\phi, t_\phi) \in rel$ *such that* $t_\psi = t_\phi$ *and (*$check\ \upsilon\ \phi$*)* **do**
        if *a pair* $(op\ \omega\ \psi, t) \in rel$ **then** $t_{\omega\psi} \leftarrow t$
        else $t_{\omega\psi} \leftarrow Nothing$
        if *a pair* $(op\ \upsilon\ \phi, t) \in rel$ **then** $t_{\upsilon\phi} \leftarrow t$
        else $t_{\upsilon\phi} \leftarrow Nothing$
        if $t_{\omega\psi} \neq Nothing$ **and** $t_{\upsilon\phi} \neq Nothing$ **then**
          $rel \leftarrow mergeEquivClasses\ rel\ t_{\omega\psi}\ t_{\upsilon\phi}$
return rel

We take the first element of each equivalence class of $\sim$ to be the canonical embedding that represents the whole equivalence class. The translation from line **8** in Algorithm 5 is therefore straightforward. The check if $Adm_{\simeq}$ is finite (line **9** in Algorithm 5) and the computation of this set (line **10**) are actually performed together by *finiteAdm_simeq* function (Algorithm 12). ϵ denotes an empty word (i.e. empty list). $Nothing$ doesn't mean non-termination — it is a proper return value.

Algorithm 12: *finiteAdm_simeq* function

Data : • a set $Embs$ of sort embeddings represented as a list $[DiagEmb]$
• a relation $\simeq$ that defines admissibility represented as $EquivRel\ DiagSort$

Result : $Just\ Adm_{\simeq}$ if $Adm_{\simeq}$ is finite; $Nothing$ otherwise

1 **return** $embWords\ Embs\ (\simeq)\ \epsilon$

Algorithm 13: $embWords$ function

Data : • a set $Embs$ of sort embeddings represented as a list $[DiagEmb]$
• a relation $\simeq$ that defines admissibility represented as $EquivRel\ DiagSort$
• a word $\omega : DiagEmbWord$

Result : $Just\ W$ (where W is the set of all the $\simeq$-admissible words over $Embs$ ending with ω) if W is finite; $Nothing$ otherwise

if $\omega = \epsilon$ **then** $W \leftarrow$ an empty list
else $W \leftarrow$ a singleton of ω
foreach $e \in Embs$ **do**
  if $e\omega$ *is* $\simeq$*-admissible* **then**
    if e *occurs in* ω **then return** $Nothing$
    else
      $W'_? \leftarrow embWords\ Embs\ (\simeq)\ e\omega$
      switch $W'_?$ **do**
        case $Nothing$
          return $Nothing$
        case $Just\ W'$
          $W \leftarrow W \cup W'$
return W

The colimit thinness check from line **11** in Algorithm 5 is probably the most complex piece of code in the module. It is illustrated in Algorithm 14.

Algorithm 14: $colimitIsThin$ function

Data : • a relation $\simeq$ represented as $EquivRel\ DiagSort$
• a set $Embs$ of sort embeddings (from diagram D) represented as list $[DiagEmb]$
• a relation $\cong_0$ represented as $EquivRel\ DiagEmbWord$

Result : $True$ if the colimit of D is thin, $False$ otherwise

$Sorts_C \leftarrow$ the list of first elements of each equivalence class in $\simeq$
$\simeq \leftarrow equivClassesToTaggedVals\ \simeq$
foreach $s \in Sorts_C$ **do** $ordMap(s) \leftarrow \emptyset$
foreach $e \in Embs$ **do** $ordMap(dom_C(e)) \leftarrow ordMap(dom_C(e)) \cup \{cod_C(e)\}$
foreach *pair* (s_1, s_2) *of sorts from* $Sorts_C$ **do**
  $S_{\geq}(s_1, s_2) \leftarrow$ the set of colimit sorts t such that $t \geq s_1$ and $t \geq s_2$
foreach *pair* (s_1, s_2) *of sorts from* $Sorts_C$ **do**
  $rel \leftarrow$ the least equivalence on $S_{\geq}(s_1, s_2)$
  $mergeCond \leftarrow \lambda s'.\lambda s''.S_{\geq}(s', s'') \neq \emptyset$
  $rel \leftarrow mergeEquivClassesBy\ mergeCond\ rel$
  $B(s_1, s_2) \leftarrow taggedValsToEquivClasses\ rel$
while $ordMap$ *is not empty* **do**
  $s \leftarrow$ a sort $t \in Sorts_C$ such that $ordMap(t) = \emptyset$
  $ordMap \leftarrow ordMap$ with the mapping for s removed
  foreach $t \in Sorts_C$ *such that* $ordMap(t)$ *is defined* **do**
    $ordMap(t) \leftarrow ordMap(t) \setminus \{s\}$
  $Embs_s \leftarrow$ all $e \in Embs$ such that $dom_C(e) = s$
  $C \leftarrow$ an empty map
  foreach *pair* (d, e) *of embeddings from* $Embs_s$ **do** $C(d, e) \leftarrow$ an empty list
  foreach *pair* (d, e) *of embeddings from* $Embs_s$ *such that* $d \cong_0 e$ **do**
    $C(d, e) \leftarrow B(cod_C(d), cod_C(e))$
  foreach *pair* (d, e) *of embeddings from* $Embs_s$ *such that* $dom(d) = dom(e)$ **do**
    if *there exists* $f \in Embs$ *such that* $dom(f) = cod(d)$ *and* $cod(f) = cod(e)$ **then**
      $\beta \leftarrow$ the equivalence class from $B(cod_C(d), cod_C(e))$ that contains $cod_C(e)$
      add β to $C(d, e)$ and to $C(e, d)$
  repeat
    foreach *tuple* (e_1, e_2, e_3) *of embeddings from* $Embs_s$ **do**
      foreach *tuple* $(\beta_{12}, \beta_{23}, \beta_{13})$ *where* $\beta_{12} \in C(e_1, e_2)$, $\beta_{23} \in C(e_2, e_3)$, $\beta_{13} \in B(e_1, e_3)$ **do**
        if $\beta_{12} \cap \beta_{23} \cap \beta_{13} \neq \emptyset$ **then**
          add β_{13} to $C(e_1, e_3)$ and to $C(e_3, e_1)$ if it's not already there
  until *fixpoint of* C *is reached*
  foreach *pair* (d, e) *of embeddings from* $Embs_s$ **do**
    if $B(cod_C(d), cod_C(e)) \not\subseteq C(d, e)$ **then return** $False$
return $True$

The set $Sorts_C$ in line **1** is the set of colimit sorts; a colimit sort is an equivalence class of $\simeq$, therefore an element from each class is chosen to represent the colimit sort. For embedding e, let $dom_C(e)$ and $cod_C(e)$ denote the colimit sorts — i.e. the equivalence classes of $\simeq$ — to which (respectively) $dom(e)$ and $cod(e)$ belong. In line **2** the relation $\simeq$ is converted to $EquivRelTagged$ representation; each sort is tagged with the first element of its equivalence class in $\simeq$. Note that the sorts used for tags are the same sorts that represent colimit sorts in $Sorts_C$. The loop in line **3** constructs $ordMap$ — a map that represents partial order on $Sorts_C$ by mapping s to the sorts immediately larger, i.e. such sorts t that there exists an embedding mapping sort from equivalence class of s to a sort from equivalence class of t. We use $\geq$ symbol to denote the order imposed by $ordMap$. The map $S_{\geq}$ constructed in line **4** maps each pair (s_1, s_2) of colimit sorts to the set of colimit sorts that are larger or equal than both s_1 and s_2. The map B created in line **5** maps each pair (s_1, s_2) of colimit sorts to an equivalence on $S_{\geq}(s_1, s_2)$.

The colimit thinness check should answer the following question: are all the cells over D derivable in the cell calculus? We call a cell (ω, υ) an *s-cell* if the equivalence class of $dom(\omega)$ and $dom(\upsilon)$ is s. The algorithm relies on the observaion that for any s-cell c all the derivations of c involve only such t-cells for which $t \geq s$. The loop in line **6** is the implementation of an induction on the colimit sorts: assuming that for all $t > s$ all t-cells belong to $\cong$, check that all s-cells belong to $\cong$. The set $Embs_s$ computed in line **7** contains all the sort embeddings whose domains are in s. The map C (computed in the subsequent lines) maps a pair (d, e) of embeddings from $Embs_s$ to a set of cells where first path starts with d and the second starts with e. Roughly, C can be thought of as a set of cells derivable in the cell calculus, while B represents all the cells in D; therefore the check in line **8** answers the question of colimit thinness. For a thorough description of this algorithm and a proof of its correctness please see the extended version of [KHTSM01].

As mentioned above, $CanonicalEmbs(D)$ (line **12** in Algorithm 5) is just a list consisting of the first elements from each equivalence class of $\sim$. $Adm^C_{\simeq}$ in line **13** is then computed using Algorithm 12 — we ensured that $Adm_{\simeq}$ is finite, so $Adm^C_{\simeq}$ must be finite as well. Generating $\cong$ (line **14**) is slightly more complicated than in case of previous relations, since the rules of cell calculus (presented in Sect. 2.2.3) and the rule **(CompDiag)** (Sect. 2.2.4) need to be represented somehow in order to use them in our framework for constructing equivalences. Rules **(Refl)**, **(Symm)** and **(Trans)** hold automatically in the construction presented. Rule **(CompDiag)** is simple, since it does not require refering to existing relation; hence closures w.r.t. those rules can be computed in one pass using $mergeEquivClassesBy$. Closures for **(Cong)** and **(Lc)** are represented by functions $congruenceClosure$ and $leftCancellableClosure$ respectively, illustrated in Algorithms 11 and 15. Note, that despite of function names, in order to obtain actual closure one needs to subsequently apply $congruenceClosure$ and $leftCancellableClosure$ until a fixpoint is reached.

Algorithm 15: $leftCancellableClosure$ function

Data : a relation rel represented as $EquivRelTagged\ DiagEmbWord\ DiagEmbWord$

Result : a relation rel' obtained from rel by applying the **(Lc)** rule once to each common prefix of each pair of words in rel

foreach *pair* $(\omega, t_\omega) \in rel$ **do**
  foreach *pair* $(\upsilon, t_\upsilon) \in rel$ *such that* $t_\upsilon = t_\omega$ **do**
    repeat
      $e_\omega \leftarrow$ the first letter of ω
      $\omega \leftarrow \omega$ with first letter removed
      $e_\upsilon \leftarrow$ the first letter of υ
      $\upsilon \leftarrow \upsilon$ with first letter removed
      if $e_\omega = e_\upsilon$ **and** ω *is not empty* **and** υ *is not empty* **then**
        $t_\omega \leftarrow t$ where $(\omega, t) \in rel$
        $t_\upsilon \leftarrow t$ where $(\upsilon, t) \in rel$
        $rel \leftarrow mergeEquivClasses\ rel\ t_\omega\ t_\upsilon$
    until $e_\omega \neq e_\upsilon$ **or** ω *is empty* **or** υ *is empty*
return rel

Now the $cong$ function used to compute $\cong$ may be defined as shown in Algorithm 16. The $\cong^R$ from line **11** is computed using the same function — only the set of words passed to it is restricted to loopless words over canonical embeddings.

Algorithm 16: $cong$ function

Data : • a set W of embedding words, represented as list $[DiagEmbWord]$
• a relation $\simeq$ represented as $EquivRel\ DiagSort$
• a relation $\sim$ represented as $EquivRel\ DiagEmb$ (for $compDiagRule$)

Result : the least equivalence on W satisfying all the rules of cell calculus w.r.t. $\simeq$

$rel \leftarrow$ the least equivalence on W
$rel \leftarrow mergeEquivClassesBy\ compDiagRule\ rel$
$check \leftarrow \lambda\omega.\lambda\upsilon.cod(\omega) \simeq dom(\upsilon)$
$op \leftarrow \lambda\omega.\lambda\upsilon.\upsilon\omega$
repeat
  $rel \leftarrow congruenceClosure\ check\ op\ rel$
  $rel \leftarrow leftCancellableClosure\ rel$
until *fixpoint on* rel *is reached*
$rel \leftarrow taggedValsToEquivClasses\ rel$
return rel

Chapter 4

Summary

Parsing and static analysis of architectural specifications and amalgamability analysis for CASL has been implemented in HETS and presented in this thesis. While parsing and institution-independent static analysis exactly follow the formal definitions presented in [RM04], it is the amalgamability analysis that turns out most challenging.

Although the algorithms implemented perform in a satisfactory manner for most of real-world specifications, their efficiency turns out to be low in certain cases and optimisations should be pursued. Even though in general the problems are PSPACE hard, it should be possible to achieve significant performance improvements. In particular $congruenceClosure$ and $colimitIsThin$ functions suffer from high time-complexity. A variation of Nelson-Oppen congruence closure algorithm [NO80] might be considered in the first case.

The comparison of architecture specfication analysis support in CATS and HETS revealed that while CATS delivers more consistent time performance it sometimes fails to detect lack of amalgamability. E.g. although the sample specification A.3 is incorrect — because of the problem with embeddings described in 2.2.3 — CATS does not issue any warning or error. Also the error messages given by HETS are more meaningful, as they specify which elements of signatures do not ensure amalgamability.

Further work on the architectural specification support in HETS might include generating proof obligations when the amalgamability conditions cannot be verified statically. This would also require implementing a verification calculus for architectural specifications [Hoff01], which is a complex area with several open research problems.

Acknowledgments I would like to express my gratitude to Andrzej Tarlecki, who has introduced me to the subject of CASL and architectural specifications and has supervised this thesis. I would also like to thank Till Mossakowski and Piotr Hoffman for their assistance while implementing the analysis of architectural specifications in HETS.

Appendix A

Sample Specifications

A.1. Failing Sharing Check

spec SPEC_1 = **sort** s
spec SPEC_2 = **sorts** s, t

arch spec ARCH_SPEC_1 =
  units U : SPEC_1;
    F : SPEC_1 $\rightarrow$ SPEC_2;
    G : SPEC_1 $\rightarrow$ SPEC_2;
  result F[U] **and** G[U]
end

In the result of specification ARCH_SPEC_1 the sorts t from F[U] and G[U] are identified although they might have different interpretations.

A.2. Using (Lc) Rule

spec SPEC_1 = **sorts** $s < u$; $t < u$
spec SPEC_2 = **sorts** $s < t$; $s < u$; $t < u$

arch spec ARCH_SPEC_1 =
  units U : SPEC_1;
    F : SPEC_1 $\rightarrow$ SPEC_2;
    G : SPEC_1 $\rightarrow$ SPEC_2;
  result F[U] **and** G[U]
end

Here the **(Lc)** rule of cell calculus is required to ensure that $s < t$ in F[U] and $s < t$ in G[U] are equal.

A.3. Failing Cell Condition

spec SPEC_1 = **sorts** s, t, u
spec SPEC_2 = **sorts** $s < t$, u

spec SPEC_3 = **sorts** $s < u$, $t < u$

arch spec ARCH_SPEC_1 =
units U : SPEC_1;
F : SPEC_1 $\rightarrow$ SPEC_2;
G : SPEC_1 $\rightarrow$ SPEC_3;
result F[U] **and** G[U]
end

This example (with more meaningful sort names) has been discussed in 2.2.3. The embedding paths $s < u$ and $s < t < u$ in the amalgamation are not guaranteed to commute.

Appendix B

CD-ROM Contents

The attached CD-ROM contains the source code of HETS. This is the version that was obtained from CVS at the moment of finishing this thesis.

B.1. Directories

- `HetCATS` – the source code of HETS,
- `uni` – the source code of UniForm Workbench that contains libraries required by HETS.

B.2. Compiling HETS

Current version of GHC[1] is required in order to compile HETS. HETS is known to compile on Linux and Solaris.

1. When in directory containing `uni` and `HetCATS` directories, enter the following commands:

   ```
   cd uni
   ./configure --enable-Het
   make boot
   make packages
   cd ../HetCATS
   ```

2. Edit the `Makefile` in `HetCATS` directory: update the variable `LINUX_IMPORTS` to match the `imports` path of your GHC installation (usually `/usr/lib/ghc-*/imports`).

3. Enter `make` in order to build HETS.

B.3. Running HETS

In order to run HETS enter the directory containing the executable file `hets` and enter:

[1] See `http://www.haskell.org/ghc`

```
./hets
```

This will display a quick reference of HETS command-line commands. The command flag `--casl-amalg` gives the user control over CASL amalgamability checking algorithms. Its syntax is

```
--casl-amalg=<analysis option list>
```

where `<analysis option list>` is a comma-separated list of zero or more of following options:

- `sharing` — perform sharing analysis,
- `cell` — analyse the cell condition (both general and restricted); this option implies `sharing`,
- `colimit-thinness` — check colimit thinness; this option implies `sharing`.

For example, specifying `--casl-amalg=colimit-thinness,cell` causes colimit thinness and cell condition checks to be performed during amalgamability analysis; sharing condition check is a prerequisite for these, hence is performed as well. `--casl-amalg=` (i.e. specifying empty list of options) turns the amalgamability analysis for CASL off completely. If not specified, `--casl-amalg=cell` is used by default.

B.4. Accessing HETS CVS Repository

Should you require the up-to-date version of HETS, it can be acquired from the CVS repository at Bremen. Please follow the steps below:

1. set the `CVSROOT` environment variable to

   ```
   :pserver:cvsread@cvs-agbkb.informatik.uni-bremen.de:/repository
   ```

2. enter the command

   ```
   cvs login
   ```

   and when prompted for password enter `cvsread`

3. enter the command

   ```
   cvs export -r stable uni
   ```

   this will download the sources of UniForm Workbench

4. enter the command

   ```
   cvs export HetCATS
   ```

   this will download the sources of HETS

Bibliography

[GB92] J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *J. ACM* 39(1), pp. 95-146, 1992

[Hets04] Till Mossakowski. HETS User Guide, 2004

[HM96] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996

[Hoff01] Piotr Hoffman. Verifying Architectural Specifications. In Maura Cerioli, Gianna Reggio (eds.): *Recent Trends in Algebraic Development Techniques, 15th Intnl. Workshop, WADT 2001*. LNCS 2267, pp. 152-175, Springer 2001

[Klin00] Bartosz Klin. Implementacja semantyki statycznej specyfikacji architekturalnych w formalizmie CASL, Masters thesis, 2000 (in Polish)

[KHTSM01] Bartek Klin, Piotr Hoffman, Andrzej Tarlecki, Lutz Schröder and Till Mossakowski. Checking Amalgamability Conditions for CASL Architectural Specifications. In Jiri Sgall, Ales Pultr, Petr Kolman (eds.): *Mathematical Foundations of Computer Science, 26th Intnl. Symp., MFCS 2001*. LNCS 2136, pp. 451-463, Springer 2001

[Moss01] Till Mossakowski. Implementing logics: from genericity to heterogeneity. Technical report

[NO80] Greg Nelson and Derek C. Oppen. Fast Decision Procedures Based on Congruence Closure, *J. ACM* 27(2), pp. 356-364, 1980

[RM04] Michel Bidoit and Peter D. Moses. CASL Reference Manual.LNCS Vol. 2960 (IFIP Series), Springer 2004

[SMTKH01] Lutz Schröder, Till Mossakowski, Andrzej Tarlecki, Bartek Klin and Piotr Hoffman. Semantics of Architectural Specifications in CASL. In Heinrich Hussmann (ed.): *Fundamental Approaches to Software Engineering, 4th Intnl. Conf., FASE 2001*. LNCS 2029, pp. 253-268, Springer 2001

[UM04] CoFI (The Common Framework Initiative). CASL User Manual. LNCS Vol. 2900 (IFIP Series), Springer 2004