HetCasl

Heterogeneous Specification Language Summary

CoFI Tentative Document: HetCasl Summary
Version: 0.11 02 March 2004

Till Mossakowski
E-mail address for comments: till@informatik.uni-bremen.de

CoFI: The Common Framework Initiative
http://www.brics.dk/Projects/CoFI

Abstract

Heterogeneous Casl (HetCasl) allows mixing specifications written in different logics (using translations between the logics). It extends Casl only at the level of structuring constructs, by adding constructs for choosing the logic and translating specifications among logics. HetCasl is needed when combining specifications written in Casl with specifications written in its sublanguages and extensions. HetCasl also allows the integration of logics that are completely different from the Casl logic.

This document provides a detailed definition of the HetCasl syntax and an informal description of the semantics, building on the existing Casl Summary [CoF04].

Contents

About this document

This document gives a detailed summary of the syntax and intended semantics of HetCasl. It is intended for readers already familiar with Casl, in particular with Casl structured specifications and libraries, see [CoF04]. Like the Casl Summary [CoF04], this document provides little or nothing in the way of discussion or motivation of design decisions; for such matters, see in particular [Mos03].

Structure

The document consists of a chapter explaining the semantic concepts needed for heterogeneous specification (Chap. 1), and a chapter (Chap. 2) describing the language constructs of HetCasl (which extend Casl structured specifications and libraries).

Like the Casl Summary [CoF04], this document provides appendices containing the abstract syntax (Appendices A and B) and the concrete syntax (Appendix C) of HetCasl specifications.

Acknowledgements

The author wishes to thank Bernd Krieg-Brückner, Klaus Lüttich, Christian Maeder, Lutz Schröder and Andrzej Tarlecki for discussions about the design of HetCasl and about heterogeneous specification in general, as well as the participants of CoFI for the joint work on Casl, which eventually laid the grounds for the development of HetCasl.

Chapter 1

Heterogeneous Concepts

1.1 Institutions

HetCasl exploits the fact that Casl structured and architectural specifications are defined independently of the underlying framework of basic specifications, formalized in terms of so-called ***institutions*** [GB92] (some category-theoretic details are omitted below) and *proof systems*.

A *basic specification framework* may be characterized by:

- a class **Sig** of ***signatures*** Σ, each determining the set of ***symbols*** $|\Sigma|$ whose intended interpretation is to be specified, with ***morphisms*** between signatures;
- a class $\mathbf{Mod}(\Sigma)$ of ***models***, with ***homomorphisms*** between them, for each signature Σ;
- a set $\mathbf{Sen}(\Sigma)$ of ***sentences*** (or ***axioms***), for each signature Σ;
- a relation $\models$ of ***satisfaction***, between models and sentences over the same signature; and
- (optionally) a ***proof system***, for inferring sentences from sets of sentences.

A ***signature morphism*** $\sigma : \Sigma \to \Sigma'$ determines a ***translation*** function $\mathbf{Sen}(\sigma)$ on sentences, mapping $\mathbf{Sen}(\Sigma)$ to $\mathbf{Sen}(\Sigma')$, and a ***reduct*** function $\mathbf{Mod}(\sigma)$ on models, mapping $\mathbf{Mod}(\Sigma')$ to $\mathbf{Mod}(\Sigma)$.[1] Satisfaction is required to be preserved by translation: for all $S \in \mathbf{Sen}(\Sigma), M' \in \mathbf{Mod}(\Sigma')$,

$$\mathbf{Mod}(\sigma)(M') \models S \iff M' \models \mathbf{Sen}(\sigma)(S).$$

[1] In fact **Sig** is a category, and **Sen**(.) and **Mod**(.) are functors. The categorical aspects of the semantics of Casl are emphasized in its formal semantics [CoF04].

If present, the proof system is required to be sound, i.e., sentences inferred from a specification are always consequences; moreover, inference is to be preserved by translation.

The semantics of a structured specification consists of a signature Σ together with a class of models in $\mathbf{Mod}(\Sigma)$. A specification is said to be ***consistent*** when there are some models that satisfy all the sentences, and ***inconsistent*** when there are no such models. A sentence is a ***consequence*** of a specification if it is satisfied in all the models of the specification.

1.2 Institution Morphisms and Comorphisms

Heterogeneous specifications involve several institutions, which are related by ***institution morphisms*** and ***comorphisms*** [GR02].

An *institution morphism* from an institution I to an institution J consists of the following components:

- a translation Φ of I-signatures to J-signatures,
- a translation α of J-sentences over $\Phi(\Sigma)$ to I-sentences over Σ,
- a translation β of I-models over Σ to J-models over $\Phi(\Sigma)$,

such that satisfaction is preserved by translation along the institution morphism: for all $\Sigma \in \mathbf{Sig}^I$, $M \in \mathbf{Mod}^I(\Sigma)$ and $\varphi' \in \mathbf{Sen}^J(\Phi(\Sigma))$,

$$M \models^I_\Sigma \alpha_\Sigma(\varphi') \iff \beta_\Sigma(M) \models^J_{\Phi(\Sigma)} \varphi$$

While institution morphisms often are projections expressing the fact that a "richer" institution is built over a "poorer" one, *institution comorphisms* often formalize inclusions or encodings between institution. An institution comorphism is similar to an institution morphism; only the directions of sentence and model translation change. It consists of the following components:

- a translation Φ of I-signatures to J-signatures,
- a translation α of I-sentences over Σ to J-sentences over $\Phi(\Sigma)$,
- a translation β of J-models over $\Phi(\Sigma)$ to I-models over Σ,

such that satisfaction is preserved by translation along the institution comorphism: for all $\Sigma \in \mathbf{Sig}^I$, $M' \in \mathbf{Mod}^J(\Phi(\Sigma))$ and $\varphi \in \mathbf{Sen}^I(\Sigma)$:

$$M' \models^J_{\Phi(\Sigma)} \alpha_\Sigma(\varphi) \iff \beta_\Sigma(M') \models^I_\Sigma \varphi.$$

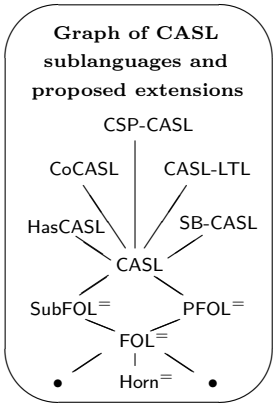

Figure 1.1: A sample logic graph.

Simple theoroidal institution morphisms and comorphisms [GR02] admit extra flexibility: signatures may be mapped to theories (where a theory consists of a signature and a set of sentences over that signature). In the sequel, we allow simple theoroidal (co)morphisms when we talk about (co)morphisms.

An institution comorphism is said to be a ***subinstitution comorphism***, if its signature and sentence translation components are embeddings, and its model translation component is an isomorphism. An institution I is said to be a ***subinstitution*** of an institution J if there is a subinstitution comorphism from I to J.[2]

Finally, a ***transformation*** τ between two institution morphisms $(\Phi_1, \alpha_1, \beta_1) : I \to J$ and $(\Phi_2, \alpha_2, \beta_2)) : I \longrightarrow J$ consists of a family of signature morphisms $\tau_\Sigma : \Phi_1(\Sigma) \longrightarrow \Phi_2$, indexed by signature Σ in I, and satisfying some natural compatibility requirements. Transformations between comorphisms are defined similarly.

1.3 Logic Graphs

Heterogeneous specification is based on an arbitrary but fixed graph of institutions, morphisms, comorphisms and transformations, which we call the ***logic graph***. We will henceforth speak of logics when speaking about the institutions of the logic graph. Each logic, morphism and comorphism in the logic graph has a unique ***name***, which is needed for referring to it.[3]

[2]The dual notion, subinstitution morphism, does not cover typcial examples, e.g. the inclusion of equational algebra into first-order logic.

[3]The logic graph is implicitly extended with identities and compositions, yielding a 2-category of morphisms and a 2-category of comorphisms.

We will assume that the logic graph comes with a ***default logic*** (which for the purposes of HETCASL is the institution underlying CASL). We also will assume that some of the subinstitution comorphisms in the logic graph are marked as (default) ***logic inclusions***. However, between a given pair of logics, at most one logic inclusion is allowed. The source logic of a logic inclusion is said to be a ***sublogic*** of the target logic. The logic inclusions are subject to a ***coherence condition***: given two paths of inclusions between two logics, there must be a comorphism transformation between the composites of the paths.[4]

Similarly, we assume that some of the institution morphisms are marked as (default) ***logic projections***, again with the proviso that between a given pair of logics, at most one logic projection is allowed, and also with a coherence condition similar to that of logic inclusions.

A subset of the logics of the logic graph is marked as ***main logics***. Each main logic comes with an associated set of sublogics.

We further assume an (associative, symmetric, idempotent) partial ***union*** operation on the logics of the logic graph. If the union of two logics is defined, we require that both logics are included in the union via logic inclusions, and that the union is minimal (w.r.t. the sublogic relation) with this property. With the help of this binary union it is easy to define unions of finite lists of logics.

For proof-theoretic purposes, it is also required that each logic in the logic graph can be mapped (via some comorphism in the logic graph) to a logic equipped with a proof system, such that this mapping preserves and reflects semantic consequence.

The ***Grothendieck logic***[5] of the logic graph puts all signatures of all involved logics side by side (hence, Grothendieck signatures are pairs consisting of a logic and a signature in that logic). A signature morphism in this large realm of signatures consists of an intra-institution signature morphism plus an inter-institution translation (along some institution morphism or comorphism). Sentences, models and satisfaction for a signature of the Grothendieck logic are just the sentences, models and satisfaction of that signature in the respective logic. Translation of sentences and models is given by composing the intra-institution translation induced by the signature morphism with the inter-institution translation given by the institution morphism or comorphism.

[4] This ensures that between two given logics in the 2-category of comorphisms, there is only one logic inclusion up to connectedness via 2-cells. Note that 2-cells are factorized out in the Grothendieck construction below.

[5] Technically, this construction corresponds to a quotient in the sense of [Mos02] of a Bi-Grothendieck institution [Mos03] — the latter can be regarded as a Grothendieck institution in the sense of [Dia02] by regarding institution morphisms as spans of comorphisms.

The (co)morphism transformations in the logic graph lead to identification of certain signature morphism in the Grothendieck logic (this concerns signature morphisms that are conceptually "the same", and in particular are known to have identical induced sentence and model translations).

A ***signature inclusion*** in the Grothendieck logic is a signature morphism that consists of an intra-institution inclusion and a logic inclusion. The *union* of two signatures in the Grothendieck logic is constructed by translating the two signatures in the union of the underlying logics, and uniting them there (note that either of these steps may be undefined, leading to undefinedness of the signature union in the Grothendieck logic).

Some logics in the logic graph may be marked as ***process logics***. Each process logic has an associated ***data logic***, which is required to be included in the process logic by means of a logic inclusion.

Chapter 2

Heterogeneous Constructs

This chapter indicates the abstract and concrete syntax of the constructs of heterogeneous specifications, extending those for CASL specifications. The semantics of a heterogeneous specification consists of a signature in the Grothendieck logic and a class of models over that signature. It is assumed that for any of the logics in the logic graph, there is an abstract syntax and semantics for basic specifications as well as for symbol lists and mappings.

For an introduction to the form of grammar used here to define the abstract syntax of language constructs, see Appendix A, which also provides the grammar defining the abstract syntax of the HETCASL specification language (as an extension of the CASL grammar).

The central slogan is: heterogeneous specification is just ordinary specification over the Grothendieck logic. The rest of this chapter details how this works.

2.1 The Current Logic

Within a homogeneous CASL structured specification, the ***current signature*** (also called local environment) may vary. Within a heterogeneous structured specification, also the ***current logic*** may vary. Since Grothendieck signatures consist of a logic and an ordinary signature, the current logic may be regarded as part of the local environment. However, there is also a current logic at the level of libraries, and a construct for changing the current logic. This is necessary in order to determine the logic in which the empty local environment (which is the empty signature in the current logic) is formed.

At some places, (implicit) ***coercions*** into the current logic may take place. More precisely, this happens for logic qualifications and data specifications

as introduced below. A specification is coerced into the current logic by translating its logic into the current logic using the corresponding logic inclusion. If there is no logic inclusion between the two logics, the construct involving the coercion is ill-formed.

Note that at other places, implicit logic coercions are induced by the definition of unions of Grothendieck signatures in Sect. 1.3 above. E.g., the semantics of instantiations of generic specifications in CASL is such that the resulting signature of the instantiation is united with the local environment. When e.g. CASL specification downloaded from a CASL library is referenced in a library written in a CASL extension, this has the effect that the CASL specification is coerced to the logic of the CASL extension.

The local environment of a heterogeneous specification may be translated only along logic inclusions, and may not be affected by other logic translations. This in particular means that translations and reductions involving non-inclusion (co)morphisms may not affect the local environment. Otherwise, the heterogeneous specification is ill-formed.

2.2 Heterogeneous Structured Specifications

```
SPEC ::= ... | LOGIC-QUALIFICATION | DATA-SPEC
```

A ***logic qualification*** selects a particular logic. A ***data specification*** is a concise notation for writing the data and process parts of a specification in a process logic. The syntax of CASL ***symbol lists*** and ***symbol mappings*** is extended in HETCASL in such a way that also inter-logic translations, reductions, fitting maps and views are allowed. The remaining CASL structuring constructs are available unchanged in HETCASL, but now with a heterogeneous meaning. Revealings and local specifications must be homogeneous, however. The semantics of basic specifications is determined by the semantics of basic specifications for the current logic.

2.2.1 Logic Qualifications

```
LOGIC-QUALIFICATION ::= logic-qual LOGIC SPEC
```

A logic qualification is written:

logic *L SP*

L must denote a logic in the logic graph. The specification *SP* gets the empty signature for that logic as local environment (this is similar to closed specifications). The result is then coerced into the enclosing current logic.

2.2.2 Logics

```
LOGIC        ::= SIMPLE-LOGIC | SUBLOGIC
SIMPLE-LOGIC ::= simple-logic LOGIC-NAME
SUBLOGIC     ::= sublogic LOGIC-NAME LOGIC-NAME
LOGIC-NAME   ::= SIMPLE-ID
```

A `SIMPLE-LOGIC` is written:

LN

LN must be the name of a main logic in the logic graph.

A `SUBLOGIC` is written

LN_1 . LN_2

LN_1 and LN_2 must be a logic names in the logic graph, such that LN_1 is a main logic and LN_2 is a sublogic of LN_1.

2.2.3 Data Specifications

```
DATA-SPEC ::= data-spec SPEC SPEC
```

A data specification is written:

data SP_1 SP_2

The current logic is required to be a process logic. SP_1 gets as local environment the empty signature in the data logic of the current logic. The resulting signature is then coerced into the current logic, and the result of this coercion is added to the local environment for SP_2.

2.2.4 Institution Morphisms and Comorphisms

The same syntax is used for both institution morphisms and comorphisms. It is determined by the context whether a morphism or a comorphism is needed. In the sequel, we will sometimes use 'morphism' when both a morphism or a comorphism can be meant.

```
MORPHISM           ::= NAMED-MORPHISM | QUALIFIED-MORPHISM
                     | ANONYMOUS-MORPHISM | DEFAULT-MORPHISM
NAMED-MORPHISM     ::= named-mor MORPHISM-NAME
QUALIFIED-MORPHISM ::= qual-mor MORPHISM-NAME LOGIC LOGIC
ANONYMOUS-MORPHISM ::= anonymous-mor LOGIC LOGIC
DEFAULT-MORPHISM   ::= default-mor LOGIC
```

```
MORPHISM-NAME      ::= SIMPLE-ID
```

A named morphism `NAMED-MORPHISM` is written

MN

MN must be the name of an institution morphism or comorphism in the logic graph.

A qualified morphism `QUALIFIED-MORPHISM` is written

$MN : LN_1 \to LN_2$

The sign '→' is input as '`->`'.

LN_1 and LN_2 must be names of logics in the logic graph, and *MN* must be the name of an institution morphism or comorphism in the logic graph, with source LN_1 and target LN_2.

An anonymous morphism `ANONYMOUS-MORPHISM` is written

$LN_1 \to LN_2$

LN_1 and LN_2 must be names of logics in the logic graph, and there must be a unique institution morphism or comorphism in the logic graph having source LN_1 and target LN_2.

An default (inclusion or projection) morphism `DEFAULT-MORPHISM` is written

$\to L$

L must be the name of a logic in the logic graph. If the enclosing construct requires an institution comorphism, there must be a (necessarily unique) logic inclusion from the source logic (as determined by the enclosing construct) to *L*. If the enclosing construct requires an institution morphism, there must be a (necessarily unique) logic projection from the source logic (as determined by the enclosing construct) to *L*.

2.2.5 Symbol Lists

```
HET-SYMB-ITEMS  ::= HOM-SYMB-ITEMS | LOGIC-REDUCTION
HOM-SYMB-ITEMS  ::= hom-symb-items SYMB-ITEMS*
LOGIC-REDUCTION ::= logic-reduction MORPHISM
```

A heterogeneous symbol list `HET-SYMB-ITEMS*` denotes a signature morphism in the Grothendieck logic. Each `HET-SYMB-ITEMS` denotes such a signature morphism, and the signature morphism for a `HET-SYMB-ITEMS*` is

obtained by composing all these signature morphisms. The composition may involve both homogeneous and heterogeneous components, e.g. as follows:

$$(L_1, \Sigma_1) \supseteq (L_1, \Sigma_2) \mapsto (L_2, \Phi(\Sigma_2)) \supseteq (L_2, \Sigma_3) \mapsto (L_3, \Phi'(\Sigma_3)) \ldots,$$

where the "$\mapsto$" components denote institution morphisms and the "$\supseteq$" components denote intra-institution signature inclusions. Each `HET-SYMB-ITEMS` gets a required target signature, which initially is the signature of the specification of the enclosing `REDUCTION`, and then is the source signature of the Grothendieck signature morphism constructed from the preceding list of `HET-SYMB-ITEMS`.

A logic reduction `LOGIC-REDUCTION` is written:

logic *MOR*

MOR must determine an institution morphism. The source logic of the institution morphism must match the required target signature as determined by the list of preceding `HET-SYMB-ITEMS`. The institution morphism contributes to the Grothendieck signature morphism denoted by the enclosing symbol list by mapping to its target logic. The resulting signature is the new required target signature.

Note that institution morphisms are defined in a way that models are mapped along their signature translation. The signature translation of the morphism is analoguous to the signature reduction as determined by a homogeneous `SYMB-ITEMS*`, and the model translation of the morphism is analoguous the model reduction as determined by a homogeneous `SYMB-ITEMS*`.

2.2.6 Reductions

The abstract syntax of reductions is changed as follows:

```
REDUCTION   ::= reduction SPEC RESTRICTION
RESTRICTION ::= HIDDEN | REVEALED
HIDDEN      ::= hidden HET-SYMB-ITEMS+
REVEALED    ::= revealed SYMB-MAP-ITEMS+
```

In this way, heterogeneous reductions can be formed. Heterogeneous symbol lists are not allowed within revealings (i.e. revealings are always required to be homogeneous).

2.2.7 Symbol Mappings

```
HET-SYMB-MAP-ITEMS ::= HOM-SYMB-ITEMS | LOGIC-TRANSLATION
```

```
HOM-SYMB-MAP-ITEMS ::= hom-symb-map-items SYMB-MAP-ITEMS*
LOGIC-TRANSLATION  ::= logic-translation MORPHISM
```

A heterogeneous symbol mapping `HET-SYMB-MAP-ITEMS*` denotes a signature morphism in the Grothendieck logic. Each `HET-SYMB-MAP-ITEMS` denotes such a signature morphism, and the signature morphism for a `HET-SYMB-MAP-ITEMS*` is obtained by composing all these signature morphisms. The composition may involve both homogeneous and heterogeneous components, e.g. as follows:

$$(L_1, \Sigma_1) \to (L_1, \Sigma_2) \mapsto (L_2, \Phi(\Sigma_2)) \to (L_2, \Sigma_3) \mapsto (L_3, \Phi'(\Sigma_3)) \ldots$$

where the "$\mapsto$" components denote institution morphisms and the "$\to$" components denote intra-institution signature inclusions. Each `HET-SYMB-MAP-ITEMS` gets a required source signature, which initially is the signature of the (source) specification of the enclosing construct, and then is the source signature of the Grothendieck signature morphism constructed from the preceding list of `HET-SYMB-MAP-ITEMS`.

A logic translation `LOGIC-TRANSLATION` is written:

logic *MOR*

MOR must determine an institution comorphism. The source logic of the institution comorphism must match the required source logic as determined by the list of preceding `HET-SYMB-MAP-ITEMS`. The institution comorphism contributes to the Grothendieck signature morphism denoted by the enclosing symbol mapping by mapping to its target logic. The resulting signature is the new required source signature. Note that institution comorphisms are defined in a way that models are mapped against their signature translation. The signature translation of the comorphism is analoguous to the signature translation as determined by a homogeneous `SYMB-MAP-ITEMS*`, and the model translation of the comorphism is analoguous to the model reduction as determined (also in a contravariant way) by a homogeneous `SYMB-MAP-ITEMS`.

2.3 Translations

```
TRANSLATION ::= translation SPEC RENAMING
RENAMING    ::= renaming  HET-SYMB-MAP-ITEMS+
```

In this way, heterogeneous translations can be formed.

2.4 Fitting Arguments

```
FIT-SPEC ::= fit-spec SPEC HET-SYMB-MAP-ITEMS*
```

For heterogeneous (i.e. those involving logic translations) fitting maps and (as well as for heterogeneous views), the rules determining a unique signature morphism between two given signatures (Sect. I:4.1.3 of the CASL Reference Manual [CoF04]) do not apply. Rather, each homogeneous sub-part of the symbol mapping has to explicitly map all the symbols of the appropriate source signature.

2.4.1 View Definitions

```
VIEW-DEFN ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE HET-SYMB-MAP-ITEMS*
```

See the remark about heterogeneous fitting maps above.

2.5 Heterogeneous Architectural Specifications

The syntax and semantics of architectural specifications remains as for CASL, except that the underlying logic is the Grothendieck logic. Like for structured specifications above, the syntax and semantics of fitting maps has changed:

```
FIT-ARG-UNIT ::= fit-arg-unit UNIT-TERM HET-SYMB-MAP-ITEMS*
```

2.6 Heterogeneous Specification Libraries

There is one new construct at the level of specification libraries.

```
LIB-ITEM        ::= ... | LOGIC-SELECTION
LOGIC-SELECTION ::= logic-select LOGIC
```

A logic selection is written:

logic L

L must denote a logic in the logic graph, which is used as current logic for the subsequent library items (until the next `LOGIC-SELECTION`). The selection of the current logic does not affect downloads from other libraries. Vice versa, downloads (as well as other library items that are not logic selections) can change the current logic only locally. That is, the current logic remains unchanged for the next library item (until a logic selection occurs).

Bibliography

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

[CoF04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS Vol. 2960 (IFIP Series). Springer, 2004.

[Dia02] R. Diaconescu. Grothendieck institutions. *Applied categorical structures*, 10:383–402, 2002.

[GB92] 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.

[GR02] J. Goguen and G. Rosu. Institution morphisms. *Formal aspects of computing*, 13:274–307, 2002.

[Mos02] T. Mossakowski. Comorphism-based Grothendieck logics. In K. Diks and W. Rytter, editors, *Mathematical foundations of computer science*, volume 2420 of *LNCS*, pages 593–604. Springer, 2002.

[Mos03] T. Mossakowski. Foundations of heterogeneous specification. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755, pages 359–375. Springer, 2003.

Appendices

Appendix A

Abstract Syntax

The *abstract syntax* is central to the definition of a formal language. It stands between the concrete representations of documents, such as marks on paper or images on screens, and the abstract entities, semantic relations, and semantic functions used for defining their meaning.

The abstract syntax has the following objectives:

- to identify and separately name the abstract syntactic entities;
- to simplify and unify underlying concepts, putting like things with like, and reducing unnecessary duplication.

There are many possible ways of constructing an abstract syntax, and the choice of form is a matter of judgement, taking into account the somewhat conflicting aims of simplicity and economy of semantic definition.

The abstract syntax is presented as a set of production rules in which each sort of entity is defined in terms of its subsorts:

```
SOME-SORT       ::=  SUBSORT-1  |  ...  |  SUBSORT-n
```

or in terms of its constructor and components:

```
SOME-CONSTRUCT  ::=  some-construct COMPONENT-1 ... COMPONENT-n
```

The productions form a context-free grammar; algebraically, the nonterminal symbols of the grammar correspond to sorts (of trees), and the terminal symbols correspond to constructor operations. The notation `COMPONENT*` indicates repetition of `COMPONENT` any number of times; `COMPONENT+` indicates repetition at least once. (These repetitions could be replaced by auxiliary sorts and constructs, after which it would be straightforward to transform the grammar into a CASL `FREE-DATATYPE` specification.)

The context conditions for well-formedness of specifications are not determined by the grammar (these are considered as part of semantics).

The grammar here has the property that there is a sort for each construct (although an exception is made for constant constructs with no components). Appendix B provides an abbreviated grammar defining the same abstract syntax. It was obtained by eliminating each sort that corresponds to a single construct, when this sort occurs only once as a subsort of another sort.

The following nonterminal symbol corresponds to the CASL syntax, and are left unspecified here: `SIMPLE-ID`. The grammars are given as extensions of the corresponding grammars for the CASL syntax, see part II of of the CASL Reference Manual [CoF04].

A.1 Structured Specifications

```
SPEC                 ::= ... | LOGIC-QUALIFICATION | DATA-SPEC

LOGIC-QUALIFICATION  ::= logic-qual LOGIC SPEC

LOGIC                ::= SIMPLE-LOGIC | SUBLOGIC
SIMPLE-LOGIC         ::= simple-logic LOGIC-NAME
SUBLOGIC             ::= sublogic LOGIC-NAME LOGIC-NAME

LOGIC-NAME           ::= SIMPLE-ID

DATA-SPEC            ::= data-spec SPEC SPEC

MORPHISM             ::= NAMED-MORPHISM | QUALIFIED-MORPHISM
                         | ANONYMOUS-MORPHISM | DEFAULT-MORPHISM
NAMED-MORPHISM       ::= named-mor MORPHISM-NAME
QUALIFIED-MORPHISM   ::= qual-mor MORPHISM-NAME LOGIC LOGIC
ANONYMOUS-MORPHISM   ::= anonymous-mor LOGIC LOGIC
DEFAULT-MORPHISM     ::= default-mor LOGIC

MORPHISM-NAME        ::= SIMPLE-ID

HET-SYMB-ITEMS       ::= HOM-SYMB-ITEMS | LOGIC-REDUCTION
HOM-SYMB-ITEMS       ::= hom-symb-items SYMB-ITEMS*
LOGIC-REDUCTION      ::= logic-reduction MORPHISM

HET-SYMB-MAP-ITEMS   ::= HOM-SYMB-ITEMS | LOGIC-TRANSLATION
HOM-SYMB-MAP-ITEMS   ::= hom-symb-map-items SYMB-MAP-ITEMS*
LOGIC-TRANSLATION    ::= logic-translation MORPHISM


RESTRICTION  ::= HIDDEN | REVEALED
HIDDEN       ::= hidden HET-SYMB-ITEMS+
REVEALED     ::= revealed SYMB-MAP-ITEMS+
```

```
RENAMING     ::= renaming  HET-SYMB-MAP-ITEMS+

FIT-ARG      ::= FIT-SPEC | FIT-VIEW
FIT-SPEC     ::= fit-spec SPEC HET-SYMB-MAP-ITEMS*
FIT-VIEW     ::= fit-view VIEW-NAME FIT-ARG*

VIEW-DEFN    ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE HET-SYMB-MAP-ITEMS*

FIT-ARG-UNIT ::= fit-arg-unit UNIT-TERM HET-SYMB-MAP-ITEMS*
```

A.2 Specification Libraries

```
LIB-ITEM             ::= ... | LOGIC-SELECTION
LOGIC-SELECTION      ::= logic-select LOGIC
```

Appendix B

Abbreviated Abstract Syntax

B.1 Structured Specifications

```
SPEC               ::= ... | logic-qual LOGIC SPEC
                         | data-spec SPEC SPEC

LOGIC              ::= simple-logic LOGIC-NAME
                     | sublogic LOGIC-NAME LOGIC-NAME

LOGIC-NAME         ::= SIMPLE-ID

MORPHISM           ::= named-mor MORPHISM-NAME
                     | qual-mor MORPHISM-NAME LOGIC LOGIC
                     | anonymous-mor LOGIC LOGIC
                     | default-mor LOGIC

MORPHISM-NAME      ::= SIMPLE-ID

HET-SYMB-ITEMS     ::= hom-symb-items SYMB-ITEMS*
                     | logic-reduction MORPHISM

HET-SYMB-MAP-ITEMS ::= hom-symb-map-items SYMB-MAP-ITEMS*
                     | logic-translation MORPHISM

RESTRICTION  ::= hidden HET-SYMB-ITEMS+
               | revealed SYMB-MAP-ITEMS+

RENAMING     ::= renaming  HET-SYMB-MAP-ITEMS+

FIT-ARG      ::= fit-spec SPEC HET-SYMB-MAP-ITEMS*
               | fit-view VIEW-NAME FIT-ARG*

VIEW-DEFN    ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE HET-SYMB-MAP-ITEMS*

FIT-ARG-UNIT ::= fit-arg-unit UNIT-TERM HET-SYMB-MAP-ITEMS*
```

B.2 Specification Libraries

```
LIB-ITEM     ::= ... | logic-select LOGIC
```

Appendix C

Concrete Syntax

The concrete syntax of HetCasl is based on concrete syntaxes for basic specifications, symbol lists and symbol mappings for each of the logics in the logic graph.

A parser for HetCasl is available via the Heterogeneous Tool Set (Hets) web page

`http://www.tzi.de/cofi/hets`

It is based upon parsers for basic specifications, symbol lists and symbol mappings for each of the logics in the logic graph.

Section C.1 below provides a context-free grammar for the HetCasl input syntax in terms of changes and additions to the context-free grammar of Casl (see Chap. II:3 of [CoF04]). It has been derived from the 'abbreviated' abstract syntax grammar in Appendix B.

The lexical syntax, comments and annotations, the literal syntax, and the display format is identical that of Casl, resp. that of the respective logic in the logic graph.

C.1 Context-Free Syntax

The following meta-notation for context-free grammars is used:

Nonterminal symbols are written as uppercase words, possibly hyphenated, e.g., `SORT`, `BASIC-SPEC`.

Terminal symbols are written as lowercase words, e.g. `free`, `assoc`.

Sequences of symbols are written with spaces between the symbols. The empty sequence is denoted by the reserved nonterminal symbol `EMPTY`.

Optional symbols are underlined, e.g. `end`, `;`. This is used also for the optional plural 's' at the end of some lowercase words used as terminal symbols, e.g. `sorts`.

Repetitions are indicated by ellipsis '`...`', e.g. `MIXFIX...MIXFIX` denotes one or more occurrences of `MIXFIX`, and `[SPEC]...[SPEC]` denotes one or more occurrences of `[SPEC]`. Repetitions often involve separators, e.g. `SORT,...,SORT` denotes one or more occurrences of `SORT` separated by '`,`'.

Alternative sequences are separated by vertical bars, e.g. `idem | unit TERM` where the alternatives are `idem` and `unit TERM`.

Production rules are written with the nonterminal symbol followed by '`::=`', followed by one or more alternatives. When a production extends a previously-given production for the same nonterminal symbol, this is indicated by writing '`...`' as its first alternative.

Start symbols are not specified.

C.2 Structured Specifications

```
SPEC                  ::= ... | logic LOGIC : GROUP-SPEC
                          | data GROUP-SPEC SPEC

LOGIC               ::= LOGIC-NAME
                      | LOGIC-NAME . LOGIC-NAME

LOGIC-NAME            ::= SIMPLE-ID

MORPHISM              ::= MORPHISM-NAME
                        | MORPHISM-NAME : LOGIC -> LOGIC
                        | LOGIC -> LOGIC
                        | -> LOGIC

MORPHISM-NAME         ::= SIMPLE-ID

HET-SYMB-ITEMS        ::= SYMB-ITEMS ,..., SYMB-ITEMS
                        | logic MORPHISM

HET-SYMB-MAP-ITEMS    ::= SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS
                        | logic MORPHISM


RESTRICTION  ::= hide HET-SYMB-ITEMS ,..., HET-SYMB-ITEMS
               | reveal SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS
```

```
RENAMING     ::= with HET-SYMB-MAP-ITEMS ,..., HET-SYMB-MAP-ITEMS

FIT-ARG      ::= SPEC fit HET-SYMB-MAP-ITEMS ,..., HET-SYMB-MAP-ITEMS
                 | SPEC
                 | view VIEW-NAME
                 | view VIEW-NAME [ FIT-ARG ]...[ FIT-ARG ]

VIEW-DEFN    ::= view VIEW-NAME : VIEW-TYPE end
                 | view VIEW-NAME : VIEW-TYPE =
                       HET-SYMB-MAP-ITEMS ,..., HET-SYMB-MAP-ITEMS end
                 | view VIEW-NAME SOME-GENERICS : VIEW-TYPE end
                 | view VIEW-NAME SOME-GENERICS : VIEW-TYPE =
                       HET-SYMB-MAP-ITEMS ,..., HET-SYMB-MAP-ITEMS end


FIT-ARG-UNIT ::= UNIT-TERM
                 | UNIT-TERM fit HET-SYMB-MAP-ITEMS ,..., HET-SYMB-MAP-ITEMS
```

C.3 Specification Libraries

```
LIB-ITEM             ::= ... | logic LOGIC
```

C.4 Lexical Syntax

For parsing outside basic specifications, symbol lists and symbol mappings, the lexical syntax is almost identical to that of CASL. There are two additional keywords:

```
logic data
```

and the keywords specific to the CASL logic are removed from the list of keywords.

For parsing basic specifications, symbol lists and symbol mappings in a logic, the lexical syntax of that logic is used.

Index