Hets User Guide
– Version 0.85 –

Till Mossakowski, Christian Maeder, Klaus Lüttich

DFKI Lab Bremen, Bremen, Germany.

Comments to: hets@informatik.uni-bremen.de
or to: hets-users@informatik.uni-bremen.de
(the latter needs subscription to the mailing list)

February 12, 2009

1 Introduction

The Heterogeneous Tool Set (Hets) is the main analysis tool for the specification language heterogeneous Casl. Heterogeneous Casl (HetCasl) combines the specification language Casl [6, 28] with Casl extensions and sublanguages, as well as completely different logics and even programming languages such as Haskell. HetCasl (see Fig. 1 for a simple subset) extends the structuring mechanisms of Casl: *Basic specifications* are unstructured specifications or modules written in a specific logic. The graph of currently supported logics and logic translations (the latter are also called comorphisms) is shown in Fig. 2, and the degree of support by Hets in Fig. 3.

With *heterogeneous structured specifications*, it is possible to combine and rename specifications, hide parts thereof, and also translate them to other logics. *Architectural specifications* prescribe the structure of implementations. *Specification libraries* are collections of named structured and architectural specifications.

Hets consists of logic-specific tools for the parsing and static analysis of the different involved logics, as well as a logic-independent parsing and static analysis tool for structured and architectural specifications and libraries. The latter of course needs to call the logic-specific tools whenever a basic specification is encountered.

Hets is based on the theory of institutions [11], which formalize the notion of a logic. The theory behind Hets is laid out in [20]. A short overview of Hets is given in [25, 26].

```
SPEC ::= BASIC-SPEC
       | SPEC then SPEC
       | SPEC then %implies SPEC
       | SPEC with SYMBOL-MAP
       | SPEC with logic ID

DEFINITION ::= logic ID
             | spec ID = SPEC end
             | view ID : SPEC to SPEC = SYMBOL-MAP end
             | view ID : SPEC to SPEC = with logic ID end

LIBRARY = DEFINITION*
```

Figure 1: Syntax of a simple subset of the heterogeneous specification language. `BASIC-SPEC` and `SYMBOL-MAP` have a logic specific syntax, while `ID` stands for some form of identifiers.

2 Logics supported by Hets

The following list of logics (formalized as so-called institutions [11]) is currently supported by Hets:

CASL extends many sorted first-order logic with partial functions and subsorting. It also provides induction sentences, expressing the (free) generation of datatypes. For more details on Casl see [28, 6]. We have implemented the Casl logic in such a way that much of the implementation can be re-used for Casl extensions as well; this is achieved via "holes" (realized via polymorphic variables) in the types for signatures, morphisms, abstract syntax etc. This eases integration of Casl extensions and keeps the effort of integrating Casl extensions quite moderate.

CoCASL [27] is a coalgebraic extension of Casl, suited for the specification of process types and reactive systems. The central proof method is coinduction.

ModalCASL [19] is an extension of Casl with multi-modalities and term modalities. It allows the specification of modal systems with Kripke's possible worlds semantics. It is also possible to express certain forms of dynamic logic.

HasCASL is a higher order extension of Casl allowing polymorphic datatypes and functions. It is closely related to the programming language Haskell and allows program constructs being embedded in the specification. An overview of HasCasl is given in [37]; the language is summarized in [34], the semantics in [36, 35].

Haskell is a modern, pure and strongly typed functional programming language. It simultaneously is the implementation language of Hets, such

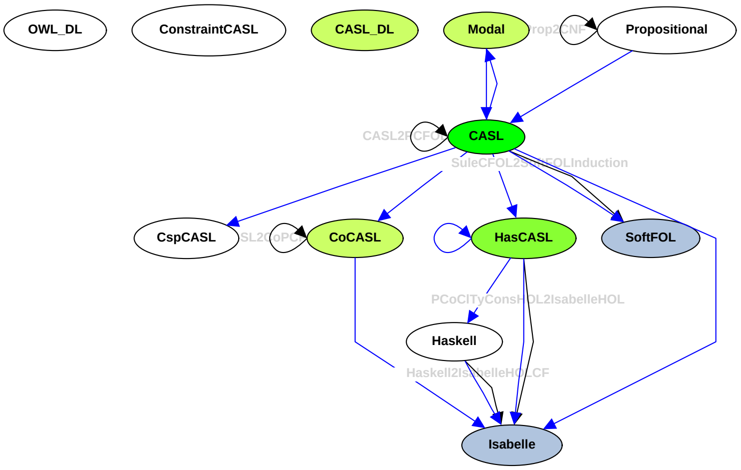


Figure 2: Graph of logics currently supported by HETS. The more an ellipse is filled with green, the more stable is the implementation of the logic. Blue indicates a prover-supported logic.

Language	Parser	Static Analysis	Prover
CASL	x	x	-
CoCASL	x	x	-
ModalCASL	x	x	-
HasCASL	x	x	-
Haskell	x	x	-
Csp-CASL	(x)	-	-
ConstraintCASL	x	(x)	-
CASL-DL	x	-	-
OWL DL basic	x	(x)	-
OWL DL structure	x	(x)	-
Propositional	x	x	x
SoftFOL	-	-	x
ISABELLE	-	-	x

Figure 3: Current degree of HETS support for the different languages.

that in the future, Hets might be applied to itself. The definitive reference for Haskell is [30], see also `www.haskell.org`.

CspCASL [33] is a combination of Casl with the process algebra CSP.

ConstraintCASL is an experimental logic for the specification of qualitative constraint calculi.

OWL DL is the Web Ontology Language (OWL) recommended by the World Wide Web Consortium (W3C, `http://www.w3c.org`). It is used for knowledge representation and the Semantic Web [5]. Hets calls an external OWL DL parser written in JAVA to obtain the abstract syntax for an OWL file and its imports. The JAVA parser is also doing a first analysis classifying the OWL ontology into the sublanguages OWL Full, OWL DL and OWL Lite. Hets only supports the last two, more restricted variants. The structuring of the OWL imports is displayed as Development Graph.

CASL-DL [16] is an extension of a restriction of Casl, realizing a strongly typed variant of OWL DL in Casl syntax. It extends Casl with cardinality restrictions for the description of sorts and unary predicates. The restrictions are based on the equivalence between Casl-DL, OWL DL and $\mathcal{SHOIN}(\mathbf{D})$. Compared to Casl only unary and binary predicates, predefined datatypes and concepts (subsorts of the topsort Thing) are allowed. It is used to bring OWL DL and Casl closer together.

Propositional is classical propositional logic, with the zChaff SAT solver [13] connected to it.

SoftFOL [17] offers three automated theorem proving (ATP) systems for first-order logic with equality: (1) Spass [43]; (2) Vampire [32]; and (3) MathServe Broker[1] [44]. These together comprise some of the most advanced theorem provers for first-order logic.

Isabelle [29] is an interactive theorem prover for higher-order logic.

Propositional, SoftFOL and Isabelle are currently the only logics coming with a prover. Proof support for the other logics can be obtained by using logic translations to a prover-supported logic.

An introduction to Casl can be found in the Casl User Manual [6]; the detailed language reference is given in the Casl Reference Manual [28]. These documents explain both the Casl logic and language of basic specifications as well as the logic-independent constructs for structured and architectural specifications. The corresponding document explaining the HetCasl language constructs for *heterogeneous* structured specifications is the HetCasl language summary [18]; a formal semantics as well as a user manual with more examples are in preparation. Some of HetCasl's heterogeneous constructs will be illustrated in Sect. 6 below.

[1] which chooses an appropriate ATP upon a classification of the FOL problem

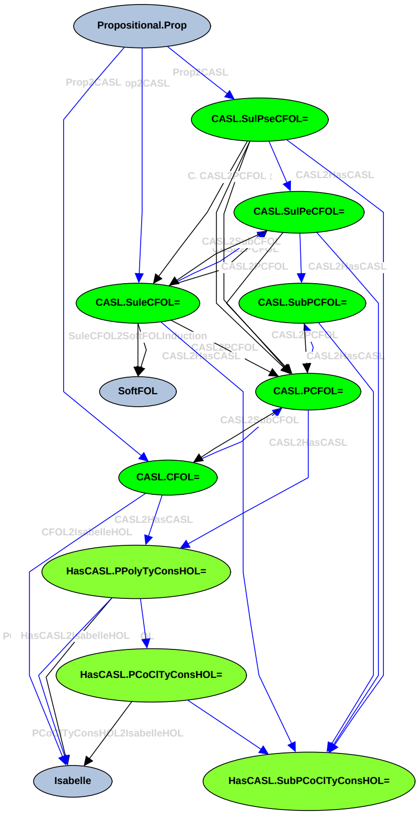


Figure 4: Graph of most important sublogics currently supported by Hets, together with their comorphisms.

3 Logic translations supported by Hets

Logic translations (formalized as institution comorphisms [10]) translate from a given source logic to a given target logic. More precisely, one and the same logic translation may have several source and target *sub*logics: for each source sublogic, the corresponding sublogic of the target logic is indicated. A graph of the most important logics and sublogics, together with their comorphisms, is shown in Fig. 4.

In more detail, the following list of logic translations is currently supported by Hets:

CASL2CoCASL	inclusion
CASL2CspCASL	inclusion
CASL2HasCASL	inclusion
CASL2Modal	inclusion
CASL2PCFOL	coding of subsorting by injections, see Chap. III:3.1 of the CASL Reference Manual [28]
CASL2SubCFOL	coding of partial functions by error elements (translation $(4a')$ of [23], but extended to subsorting)
CASL2TopSort	coding of subsorting by a top sort and unary predicates for the subsorts
CFOL2IsabelleHOL	coding of Casl to Isabelle (translation (7) of [23])
CoCASL2CoPCFOL	coding of subsorting by injections, similar to CASL2PCFOL
CoCASL2CoSubCFOL	coding of partial functions by error supersorts, similar to CASL2SubCFOL
CoCFOL2IsabelleHOL	coding of CoCasl to Isabelle, similar to CFOL2IsabelleHOL
HasCASL2HasCASL	coding of HasCasl axiomatic recursive definitions as HasCasl recursive program definitions
HasCASL2Haskell	translation of HasCasl recursive program definitions to Haskell
HasCASL2IsabelleHOL	coding of HasCASL to Isabelle/HOL [12]
Haskell2IsabelleHOL	coding of Haskell to Isabelle/HOL [41]
Haskell2IsabelleHOLCF	coding of Haskell to Isabelle/HOLCF [41]
Modal2CASL	the standard translation of modal logic to first-order logic [7]
PCoClTyConsHOL2IsabelleHOL	coding of HasCasl to Isabelle/HOL
SuleCFOL2SoftFOL	coding of CASL to SoftFOL [17], mapping types to soft types
SuleCFOL2SoftFOLIndcuction	dto., but with instances of induction axioms for all proof goals
Prop2CASL	inclusion
Prop2CNF	conversion of propositional formulas to conjunctive normal form

4 Getting started

The latest Hets version can be obtained from the Hets tools home page

`http://www.dfki.de/sks/hets`

Since Hets is being improved constantly, it is recommended always to use the latest version.

Hets currently is available for Linux, Solaris and Mac OS-X.

There are three possibilities to install Hets:

1. The Java-based Hets installer. Download a `.jar` file and start it with

   ```
   java -jar file.jar
   ```

   Note that you need Sun Java 1.4.2 or later. On a Mac, you can just double-click on the `.jar` file.

   The installer will lead you through the installation with a graphical interface. It will download and install further software (if not already installed on your computer):

CASL-lib	specification library	`http://www.cofi.info/Libraries`
uDraw(Graph)	graph drawing	`http://www.informatik.uni-bremen.de/uDrawGraph/en/`
Tcl/Tk	graphics widget system	`http://www.scriptics.com/software/tcltk/`
Spass	theorem prover	`http://spass.mpi-sb.mpg.de/`
Isabelle	theorem prover	`http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`
(X)Emacs	editor (for Isabelle)	(must be installed manually)

2. If you do not have Sun Java, you can just download the hets binary. You have to unpack it with `bunzip2` and then put it at some place coverd by your `PATH`. You also have to install the above mentioned software and set several environment variables, as explained on the installation page.

3. If you want to compile Hets from the sources, please follow the link "Hets: source code and information for developers" on the Hets web page, download the sources (as tarball or from svn), and follow the instructions in the `INSTALL` file.

5 Analysis of Specifications

Consider the following Casl specification:

spec Strict_Partial_Order =
  sort *Elem*
  pred __ < __ : *Elem* × *Elem*

$\forall x, y, z : \mathit{Elem}$

- $\neg(x < x)$ %(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

Hets can be used for parsing and checking static well-formedness of specifications.

Let us assume that the example is in a file named `Order.casl` (actually, this file is provided with the Hets distribution). Then you can check the well-formedness of the specification by typing (into some shell):

```
hets Order.casl
```

Hets checks both the correctness of this specification with respect to the Casl syntax, as well as its correctness with respect to the static semantics (e.g. whether all identifiers have been declared before they are used, whether operators are applied to arguments of the correct sorts, whether the use of overloaded symbols is unambiguous, and so on). The following flags are available in this context:

`-p, --just-parse` Just do the parsing – the static analysis is skipped.

`-s, --just-structured` Do the parsing and the static analysis of (heterogeneous) structured specifications, but leave out the analysis of basic specifications. This can be used for prototyping issues, namely to quickly produce a development graph showing the dependencies among the specifications (cf. Sect. 7) even if the individual specifications are not correct yet.

`-L DIR, --hets-libdir=DIR` Use `DIR` as the directory for specification libraries (equivalently, you can set the variable `HETS_LIB` before calling Hets).

`--casl-amalg=ANALYSIS` For the analysis of architectural specification (a quite advanced feature of Casl), the `ANALYSIS` argument specifies the options for amalgamability checking algorithm for Casl logic; it is a comma-separated list of zero or more of the following options:

`sharing` perform sharing analysis for sorts, operations and predicates.

`cell` perform cell condition check; implies `sharing`. With this option on, the subsort embeddings are analyzed.

`colimit-thinness` perform colimit thinness check; implies `sharing`. The colimit thinness check is less complete and usually takes longer than the full cell condition check (`cell` option), but may prove more efficient in case of certain specifications.

If `ANALYSIS` is empty then amalgamability analysis for Casl is skipped. The default value for `--casl-amalg` is `cell`.

6 Heterogeneous Specification

HETS accepts plain text input files with the following endings:

Ending	default logic	structuring language
`.casl`	CASL	CASL
`.het`	CASL	CASL
`.hs`	Haskell	Haskell
`.owl`	OWL DL, OWL Lite	OWL

Although the endings `.casl` and `.het` are interchangeable, the former should be used for libraries of homogeneous CASL specifications and the latter for HETCASL libraries of heterogeneous specifications (that use the CASL structuring constructs). Within a HETCASL library, the current logic can be changed e.g. to HASCASL in the following way:

```
logic HasCASL
```

The subsequent specifications are then parsed and analysed as HASCASL specifications. Within such specifications, it is possible to use references to named CASL specifications; these are then automatically translated along the default embedding of CASL into HASCASL (cf. Fig. 2). (There are also heterogeneous constructs for explicit translations between logics, see [18].)

The ending `.hs` is available for directly reading in Haskell programs and hence supports the Haskell module system. By contrast, in HETCASL libraries (ending with `.het`), the logic Haskell has to be chosen explicitly, and the CASL structuring syntax needs to be used:

```
library Factorial

logic Haskell

spec Factorial =
fac :: Int -> Int
fac n = foldl (*) 1 [1..n]
end
```

Note that according to the Haskell syntax, Haskell function declarations and definitions need to start with the first column of the text.

7 Development Graphs

Development graphs are a simple kernel formalism for (heterogeneous) structured theorem proving and proof management.

A development graph consists of a set of nodes (corresponding to whole structured specifications or parts thereof), and a set of arrows called *definition links*, indicating the dependency of each involved structured specification on its subparts. Each node is associated with a signature and some set of local axioms. The axioms of other nodes are inherited via definition links. Definition links are usually drawn as black solid arrows, denoting an import of another specification.

Complementary to definition links, which *define* the theories of related nodes, *theorem links* serve for *postulating* relations between different theories. Theorem links are the central data structure to represent proof obligations arising in formal developments. Theorem links can be *global* (drawn as solid arrows) or *local* (drawn as dashed arrows): a global theorem link postulates that all axioms of the source node (including the inherited ones) hold in the target node, while a local theorem link only postulates that the local axioms of the source node hold in the target node.

Both definition and theorem links can be *homogeneous*, i.e. stay within the same logic, or *heterogeneous*, i.e. the logic changes along the arrow. Technically, this is the case for Grothendieck signature morphisms (ρ, σ) where $\rho \neq id$. This case is indicated with double arrows.

Theorem links are initially displayed in red. The *proof calculus* for development graphs [21, 20] is given by rules that allow for proving global theorem links by decomposing them into simpler (local and global) ones. Theorem links that have been proved with this calculus are drawn in green. Local theorem links can be proved by turning them into *local proof goals*. The latter can be discharged using a logic-specific calculus as given by an entailment system for a specific institution. Open local proof goals are indicated by marking the corresponding node in the development graph as red; if all local implications are proved, the node is turned into green. This implementation ultimately is based on a theorem [20] stating soundness and relative completeness of the proof calculus for heterogeneous development graphs.

Details can be found in the CASL Reference Manual [28, IV:4] and in [20, 21, 26].

The following option lets HETS show the development graph of a specification library:

`-g, --gui` Shows the development graph in a GUI window.

Here is a summary of the types of nodes and links occurring in development graphs:

Named nodes correspond to a named specification.

Unnamed nodes correspond to an anonymous specification.

Elliptic nodes correspond to a specification in the current library.

Rectangular nodes are external nodes corresponding to a specification downloaded from another library.

Red nodes have open proof obligations.

Green nodes have all proof obligations resolved.

Black links correspond to reference to other specifications (definition links in the sense of [28, IV:4]).

Blue links correspond to hiding (hiding definition links).

Red links correspond to open proof obligations (theorem links).

Green links correspond to proved proof obligations (theorem links).

Yellow links correspond to open proof obligations involving hiding (hiding theorem links).

Solid links correspond to global (definition or theorem) links in the sense of [28, IV:4].

Dashed links correspond to local (definition or theorem) links in the sense of [28, IV:4].

Single links have homogeneous signature morphisms (staying within one and the same logic).

Double links have heterogeneous signature morphisms (moving between logics).

We now explain the menus of the development graph window. Most of the pull-down menus of the window are uDraw(Graph)-specific layout menus; their function can be looked up in the uDraw(Graph) documentation[2]. The exception is the Edit menu. Moreover, the nodes and links of the graph have attached pop-up menus, which appear when clicking with the right mouse button.

Edit This menu has the following submenus:

undo Undo the last development graph proof step (see under Proofs)

redo Restore the last undone development graph proof step (see under Proofs)

reload Reload the specification library (Attention! all proofs are lost. A change management keep proofs is in preparation.)

Unnamed nodes The "Hide/show names" menu is a toggle: you can switch on or off the display of names for nodes that are initially unnamed. The newly named nodes get names that are derived from named neighbour nodes.

With the "Hide nodes" submenu, it is possible to reduce the complexity of the graph by hiding all unnamed nodes; only nodes corresponding to named specifications remain displayed. Paths between named nodes going through unnamed nodes are displayed as links. With the "Show nodes" submenu, the unnamed nodes re-appear.

[2] see `http://www.informatik.uni-bremen.de/uDrawGraph/en/service/uDG31_doc/`.

Proofs This menu allows to apply some of the deduction rules for development graphs, see Sect. IV:4.4 of the CASL Reference Manual [28] or one of [20, 21, 26]. While support for local and global (definition or theorem) links is stable, support for hiding links and checking conservativity is still experimental. In most cases, it is advisable to use "Automatic", which automatically applies the rules in the correct order. As a result, the the open theorem links (marked in red) will be reduced to local proof goals, that is, they become green, and instead, some of the node will get red, indicating open local proof goals. Besides the deduction rules, the menu contains entries for computing a colimit approximation for the development graph and for computing normal forms of all nodes (needed when dealing with hiding).

Translate Graph translates the whole development graph along a logic comorphism.

Show Logic Graph shows the graph of logics and logic comorphisms currently supported by HETS.

Show Library Graph shows the graph of libraries that have been loaded into HETS, and their dependencies. For library, the corresponding development graphs can be shown using its node menu. Also, a list of specifications and views can be shown.

Pop-up menu for nodes Here, the number of submenus depends on the type of the node:

Show signature Shows the signature of the node.

Show local axioms Shows the local axioms of the node.

Show theory Shows the theory of the node (including axioms imported from other nodes). Warning: axioms imported via hiding links are not part of the theory; they can be made visible only by re-adding the hidden symbols, using the proof rule *Theorem-Hide-Shift*.

Translate theory Translates the theory of a node to another logic. A menu with the possible translation paths will be displayed.

Taxonomy graphs (Only available for some logics) Shows the subsort graph of the signature of the node.

Show sublogic Shows the logic and, within that logic, the minimal sublogic for the signature and the axioms of the node.

Show origin Shows the kind of CASL structuring construct that led to the node.

Show proof status Show open and proven local proof goals.

Prove Try to prove the local proof goals. See Section 10 for details.

Check consistency Check the consistency of the theory of the node.

Show just subtree (Only for named nodes) Reduce the complexity of the graph by just showing the subtree below the current node.

Undo show just subtree (Only for named nodes) Undo the reduction.

Show referenced library (Only for external nodes) Open a new window showing the development graph for the library the external node refers to.

Show number of node Show the internal number of the node.

Pop-up menu for links Again, the number of submenus depends on the type of the link:

Show morphism Shows the signature morphism of the link. It consists of two components: a logic translation and a signature morphism in the target logic of the logic translation. In the (most frequent) case of an intra-logic signature morphism, the logic translation component is just the identity.

Show origin Shows the kind of Casl structuring construct that led to the link.

Show proof status (Only for theorem links) Show the proof status.

Check conservativity (Experimental) Check whether the theory of the target node of the link is a conservative extension of the theory of the source node.

Show ID of this edge Shows the internal number of the edge. These numbers are also used in the proof status information for edges.

8 Reading, Writing and Formatting

Hets provides several options controlling the types of files that are read and written.

`-i ITYPE, --input-type=ITYPE` Specify `ITYPE` as the type of the input file. The default is `het` (HetCasl plain text). `ast` is for reading in abstract syntax trees in ATerm format, while `ast.baf` reads in the compressed ATerm format. The possible input types are:

```
(casl|het|owl|hs|prf|omdoc|hpf|[tree.]gen_trm[.baf])
```

`-O DIR, --output-dir=DIR` Specify `DIR` as destination directory for output files.

`-o OTYPES, --output-types=OTYPES` `OTYPES` is a comma separated list of output types:

```
  prf
| env
| omdoc
| hs
| thy
| comptable.xml
| (sig|th)[.delta]
| pp.(het|tex)
| graph.(exp.dot|dot)
| dfg[.c]
| tptp[.c]
```

The default is `prf` (a synonym for `dg.taf`), which means that the development graph of the library is stored in textual ATerm format (`taf`). This format can be read in when a library is downloaded from another library, avoiding the need to re-analyse the downloaded library. You can also directly read in the `prf` format (both from the command line, by calling Hets on the `prf` file, and from the GUI, using the File-Open menu.

The `env` format is currently not used.

The `omdoc` format [14] is an XML-based markup format and data model for Open Mathematical Documents. It serves as semantics-oriented representation format and ontology language for mathematical knowledge. Currently, Casl specifications can be output in this format; support for further logics is planned.

The `hs` format is used for Haskell modules. Executable Casl or HasCasl specifications can be translated to Haskell.

When the `thy` format is selected, Hets will try to translate each specification in the library to Isabelle, and write one Isabelle `.thy` file per specification.

When the `comptable.xml` format is selected, Hets will extract the composition and inverse table of a Tarskian relation algebra from specification(s) (selected with the `-n` or `--spec` option). It is assumed that the relation algebra is generated by basic relations, and that the specification is written in the Casl logic. A sample specification of a relation algebra can be found in the Hets library `Calculi/Space/RCC8.het`, available from `www.cofi.info/Libraries`. The output format is XML, the URL of the DTD is included in the XML file.

The `pp` format is for pretty printing, either as plain text (`het`), LaTeXinput (`tex`) or HTML (`html`). A formatter with pretty-printed output currently is available only for the Casl logic. For example, it is possible to generate a pretty printed LaTeX version of `Order.casl` by typing:

```
hets -o pp.tex Order.casl
```

This will generate a file `Order.pp.tex`. It can be included into LaTeX documents, provided that the style `hetcasl.sty` coming with the Hets distribution (`LaTeX/hetcasl.sty`) is used.

The `dfg` format is used by the Spass theorem prover [43].

The `tptp` format (`http://www.tptp.org`) is a standard format for first-order theorem provers.

`-t TRANS`, `--translation=TRANS` chooses a translation option. `TRANS` is a colon-separated list without blanks of one or more comorphism names (see Sect. 3).

`-R`, `--recursive` output also imported libraries.

The other output formats are for future usage.

9 Miscellaneous Options

`-v[Int]`, `--verbose[=Int]` Set the verbosity level according to `Int`. Default is 1.

`-q`, `--quiet` Be quiet – no diagnostic output at all. Overrides -v.

`-V`, `--version` Print version number and exit.

`-h`, `--help`, `--usage` Print usage information and exit.

`+RTS -KIntM -RTS` Increase the stack size to `Int` megabytes (needed in case of a stack overflow). This must be the first option.

`-l LOGIC`, `--logic=LOGIC` chooses the initial logic, which is used for processing the specifications before the first **logic L** declaration. The default is CASL.

`-n SPECS`, `--spec=SPECS` chooses a list of named specifications for processing

`-m FILE`, `--modelSparQ=FILE` model check a qualitative calculus given in SparQ lisp notation [42] against a CASL specification

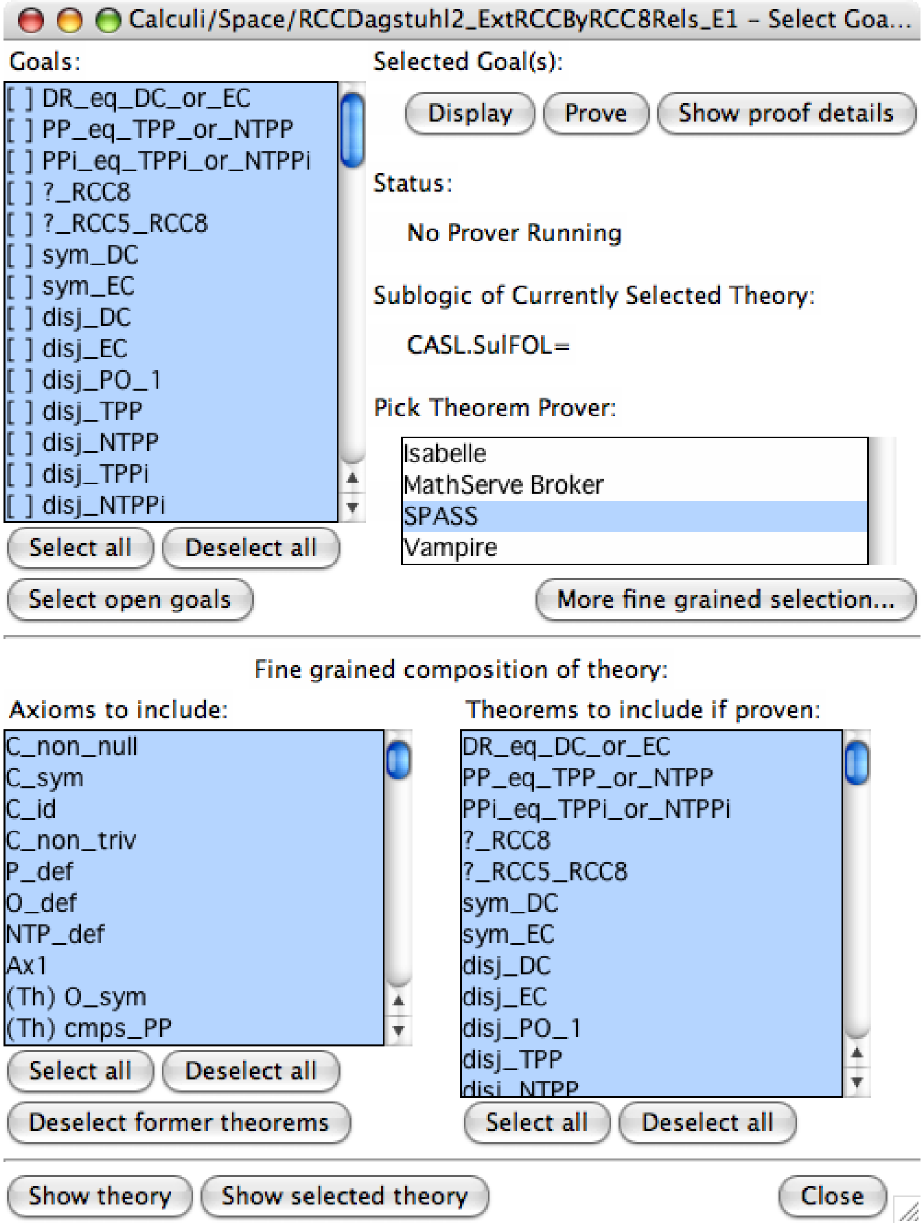
`-I`, `--interactive` run HETS in interactive mode

10 Proofs with HETS

The proof calculus for development graphs (Sect. 7) reduces global theorem links to local proof goals. Local proof goals (indicated by red nodes in the development graph) can be eventually discharged using a theorem prover, using the "Prove" menu of a red node.

The graphical user interface (GUI) for calling a prover is shown in Fig. 5 — we call it "Proof Management GUI". The list on the left shows all goal names prefixed with the proof status in square brackets. A proved goal is indicated by a '+', a '-' indicates a disproved goal, a space denotes an open goal, and a '×' denotes an inconsistent specification (aka a fallen '+'; see below for details).

If you open this GUI when processing the goals of one node for the first time, it will show all goals as open. Within this list you can select those goals that should be inspected or proved. The button 'Display' shows the selected



Calculi/Space/RCCDagstuhl2_ExtRCCByRCC8Rels_E1 – Select Goa...

Goals:

[ ] DR_eq_DC_or_EC
[ ] PP_eq_TPP_or_NTPP
[ ] PPi_eq_TPPi_or_NTPPi
[ ] ?_RCC8
[ ] ?_RCC5_RCC8
[ ] sym_DC
[ ] sym_EC
[ ] disj_DC
[ ] disj_EC
[ ] disj_PO_1
[ ] disj_TPP
[ ] disj_NTPP
[ ] disj_TPPi
[ ] disj_NTPPi

Selected Goal(s):

Status:

No Prover Running

Sublogic of Currently Selected Theory:

CASL.SulFOL=

Pick Theorem Prover:

Isabelle
MathServe Broker
SPASS
Vampire

Fine grained composition of theory:

Axioms to include:

C_non_null
C_sym
C_id
C_non_triv
P_def
O_def
NTP_def
Ax1
(Th) O_sym
(Th) cmps_PP

Theorems to include if proven:

DR_eq_DC_or_EC
PP_eq_TPP_or_NTPP
PPi_eq_TPPi_or_NTPPi
?_RCC8
?_RCC5_RCC8
sym_DC
sym_EC
disj_DC
disj_EC
disj_PO_1
disj_TPP
disj_NTPP

Figure 5: Hets Goal and Prover Interface

goals in the ASCII syntax of this theory's logic in a separate window. With the 'Prove' button the actual prover is launched. This is described in more detail in the paragraphs below. By pressing the 'Show Proof Details' button a window is opened where for each proved goal the used axioms, its proof script, and its proof are shown — the level of detail depends on the used theorem prover. The 'Status:' shows either 'No prover running' or 'Waiting for prover' in black or blue. If you press the 'Close' button the window is closed and the status of the goals' list is integrated into the development graph. If all goals have been proved, the selected node turns from red into green.

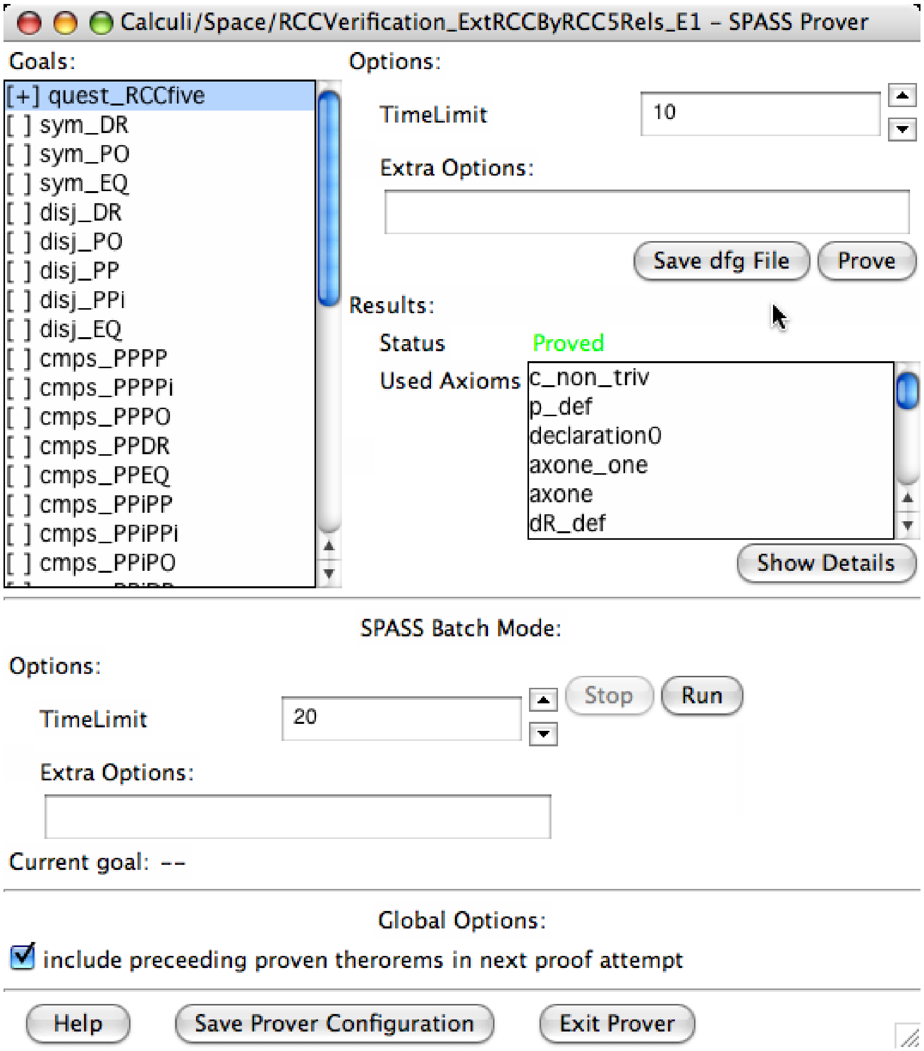
The list 'Pick Theorem Prover:' lets you choose one of the connected provers (currently, these are Isabelle, MathServe Broker, Spass, Vampire, and zChaff, described below). By pressing 'Prove' the selected prover is launched and the theory along with the selected goals is translated via the shortest possible path of comorphisms into the provers logic. The button 'More fine grained selection...' lets you pick a (composed) comorphism in a separate window from where the prover is launched then.

Since the amount and kind of sentences sent to an ATP system is a major factor for the performance of the ATP system, it is possible to select the axioms and proven theorems that will comprise the theory of the next proof attempt. Based on this selection the sublogic may vary and also the available provers and comorphisms to provers. Former theorems that are imported from other specifications are marked with the prefix '(Th)'. Since former theorems do not add additional logical content, they may be safely removed from the theory.

10.1 Automated Theorem Proving Systems (Logic SoftFOL)

All ATPs integrated into Hets share the same GUI, with only a slight modification for the MathServe Broker: it does not have input fields for extra options. Figure 6 shows the instantiation for Spass, where in the lower part of the window the batch mode can be controlled. The upper part shows on the left the list of goals (with the same status indicators as in the Proof Management GUI), and on the right a proof attempt of the selected goal is controlled and the result of the last proof attempt is displayed. The status line indicates 'Open', 'Running', 'Proved', 'Disproved', 'Open (Time is up!)', and 'Proved (Theory inconsistent!)'. The list of used axioms is actually only filled by Spass. The help button displays information about the extra options accepted by the ATP system. The button 'Show Details' shows the whole output of the ATP system. 'Save Prover Configuration' allows you to save the configuration and status of each proof for documentation. By pressing the button 'Exit Prover' the status of these proofs and goals is transferred back to the Proof Management GUI.

The MathServe system [44] developed by Jürgen Zimmer provides a unified interface to a range of different ATP systems; the most important systems are listed in Table 1, along with their capabilities. These capabilities are derived from the *Specialist Problem Classes* (SPCs) defined upon the basis of logical, language and syntactical properties by Sutcliffe and Suttner [38]. Only two



Calculi/Space/RCCVerification_ExtRCCByRCC5Rels_E1 – SPASS Prover

Goals:

[+] quest_RCCfive
[ ] sym_DR
[ ] sym_PO
[ ] sym_EQ
[ ] disj_DR
[ ] disj_PO
[ ] disj_PP
[ ] disj_PPi
[ ] disj_EQ
[ ] cmps_PPPP
[ ] cmps_PPPPi
[ ] cmps_PPPO
[ ] cmps_PPDR
[ ] cmps_PPEQ
[ ] cmps_PPiPP
[ ] cmps_PPiPPi
[ ] cmps_PPiPO

Options:

TimeLimit 10

Extra Options:

Results:

Status Proved

Used Axioms
c_non_triv
p_def
declaration0
axone_one
axone
dR_def

SPASS Batch Mode:

Options:

TimeLimit 20

Extra Options:

Current goal: --

Global Options:

☑ include preceeding proven therorems in next proof attempt

Figure 6: Interface of the Spass prover

ATP System	Version	Suitable Problem Classes[a]
DCTP	10.21p	effectively propositional
EP	0.91	effectively propositional; real first-order, no equality; real first-order, equality
Otter	3.3	real first-order, no equality
SPASS	2.2	effectively propositional; real first-order, no equality; real first-order, equality
Vampire	8.0	effectively propositional; pure equality, equality clauses contain non-unit equality clauses; real first-order, no equality, non-Horn
Waldmeister	704	pure equality, equality clauses are unit equality clauses

[a] The list of problem classes for each ATP system is not exhaustive, but only the most appropriate problem classes are named according to benchmark tests made with MathServe by Jürgen Zimmer.

Table 1: ATP systems provided as Web services by MathServe

of the Web services provided by the MathServe system are used by HETS currently: Vampire and the brokering system. The ATP systems are offered as Web Services using standardized protocols and formats such as SOAP, HTTP and XML. Currently, the ATP system Vampire may be accessed from HETS via MathServe; the other systems are only reached after brokering.

SPASS

The ATP system SPASS [43] is a resolution-based prover for first-order logic with equality. Furthermore, it provides a soft typing mechanism with subsorting that treats sorts as unary predicates. The ATP SPASS should be installed locally and available through your `$PATH` environment variable.

Vampire

The ATP system Vampire is the winner of the last 5 CADE ATP System Competitions (CASC) (2002–2006) in the devisions `FOF` and `CNF`. It is a resolution based ATP system supporting the calculi of ordered binary resolution and superposition for handling equality. See `http://www.cs.miami.edu/~tptp/CASC/J3/SystemDescriptions.html#Vampire---8.0` for detailed information. The connection to Vampire is achieved by using an Web service of the MathServe system.

MathServe Broker

The brokering service chooses the most appropriate ATP system upon a classification based on the SPCs, and on a training with the library Thousands of Problems for Theorem Provers (TPTP) [44]. The TPTP format has been introduced by Sutcliffe and Suttner for the annual competition CASC [39] and provides a unified syntax for untyped FOL with equality, but without any symbol declaration.

10.2 Isabelle

Isabelle [29] is an interactive theorem prover, which is more powerful than ATP systems, but also requires more user interaction.

Isabelle has a very small core guaranteeing correctness, and its provers, like the simplifier or the tableaux prover, are built on top of this core. Furthermore, there is over fifteen years of experience with it, and several mathematical textbooks have been partially verified with Isabelle.

Isabelle is a tactic based theorem prover implemented in standard ML. The main Isabelle logic (called Pure) is some weak intuitionistic type theory with polymorphism. The logic Pure is used to represent a variety of logics within Isabelle; one of them being Hol (higher-order logic). For example, logical implication in Pure (written `==>`, also called meta-implication), is different from logical implication in Hol (written `-->`, also called object implication).

It is essential to be aware of the fact that the Isabelle/Hol logic is different from the logics that are encoded into it via comorphisms. Therefore, the formulas appearing in subgoals of proofs with Isabelle will not conform to the syntax of the original input logic. They may even use features of Isabelle/Hol such as higher-order functions that are not present in an input logic like Casl.

Isabelle is started with ProofGeneral [2, 1] in a separate Emacs [9, 40]. The Isabelle theory file conforms to the Isabelle/Isar syntax [29]. It starts with the theory (encoded along the selected comorphism), followed by a list of theorems. Initially, all the theorems have trivial proofs, using the ‘oops‘ command. However, if you have saved earlier proof attempts, Hets will patch these into the generated Isabelle theory file, ensuring that your previous work is not lost. (But note that this patching can only be successful if you do not rename specifications, or change their structure.) You now can replace the ’oops’ commands with real Isabelle proofs, and use Proof General to step through the proofs. You finish your session by saving your file (using the Emacs file menu, or the Ctrl-x Ctrl-s key sequence), and by exiting Emacs (Ctrl-x Ctrl-c).

10.3 zChaff

zChaff is a solver for satisfiabily problems of boolean formulas (Sat) in CNF. It is connected as a prover for propositional logic to Hets. The prover Spass is used to transform arbitrary boolean formulas to CNF. zChaff implements the Chaff algorithm. We are using the property, that a conjecture under the assumption of a set of axioms is true, if the variables of axioms together with the negation of the conjecture have no satisfying assignment, to prove theorems with zChaff. That is why you see the result Unsat in the proof details, if a theorem has been proved to be true. zChaff uses the same ATP GUI as the provers for SoftFOL (ref. to section 10.1). zChaff does not accept any options apart from the time-limit. The current integration of zChaff into Hets has been tested with zChaff 2004.11.15.

11 Limits of Hets

Hets is still intensively under development. In particular, the following points are still missing:

- There is no proof support for architectural specifications.
- Distributed libraries are always downloaded from the local disk, not from the Internet.
- Version numbers of libraries are not considered properly.
- The proof engine for development graphs provides only experimental support for hiding links and for conservativity.

12 Architecture of Hets

The architecture of Hets is shown in Fig. 8. How is a single logic implemented in the Heterogeneous Tool Set? This is depicted in the left column of Fig. 8.

Hets provides an abstract interface for institutions, so that new logics can be integrated smoothly. In order to do so, a parser, a static checker and a prover for basic specifications in the logic have to be provided.

Each logic is realized in the programming language Haskell [30] by a set of types and functions, see Fig. 7, where we present a simplified, stripped down version, where e.g. error handling is ignored. For technical reasons a logic is *tagged* with a unique identifier type (`lid`), which is a singleton type the only purpose of which is to determine all other type components of the given logic. In Haskell jargon, the interface is called a multiparameter type class with functional dependencies [31]. The Haskell interface for logic translations is realised similarly.

The logic-independent modules in Hets can be found in the right half of Fig. 8. These modules comprise roughly one third of Hets' 100.000 lines of Haskell code.

The heterogeneous parser transforms a string conforming to the syntax in Fig. 1 to an abstract syntax tree, using the `Parsec` combinator parser [15]. Logic and translation names are looked up in the logic graph — this is necessary to be able to choose the correct parser for basic specifications. Indeed, the parser has a state that carries the current logic, and which is updated if an explicit specification of the logic is given, or if a logic translation is encountered (in the latter case, the state is set to the target logic of the translation). With this, it is possible to parse basic specifications by just using the logic-specific parser of the current logic as obtained from the state.

```
class Logic lid sign morphism sentence basic_spec symbol_map
    | lid -> sign morphism sentence basic_spec symbol_map where
  identity :: lid -> sign -> morphism
  compose :: lid -> morphism -> morphism -> morphism
  dom, codom :: lid -> morphism -> sign
  parse_basic_spec :: lid -> String -> basic_spec
  parse_symbol_map :: lid -> String -> symbol_map
  parse_sentence   :: lid -> String -> sentence
  empty_signature :: lid -> sign
  basic_analysis :: lid -> sign -> basic_spec -> (sign, [sentence])
  stat_symbol_map :: lid -> sign -> symbol_map -> morphism
  map_sentence :: lid -> morphism -> sentence -> sentence
  provers ::
    lid -> [(sign, [sentence]) -> [sentence] -> Proof_status]
  cons_checkers :: lid -> [(sign, [sentence]) -> Proof_status]

class Comorphism cid
          lid1 sign1 morphism1 sentence1 basic_spec1 symbol_map1
          lid2 sign2 morphism2 sentence2 basic_spec2 symbol_map2
      | cid -> lid1 lid2 where
  sourceLogic :: cid -> lid1       targetLogic :: cid -> lid2
  map_theory :: cid -> (sign1, [sentence1]) -> (sign2, [sentence2])
  map_morphism :: cid -> morphism1 -> morphism2
```

Figure 7: The basic ingredients of logics and logic comorphisms

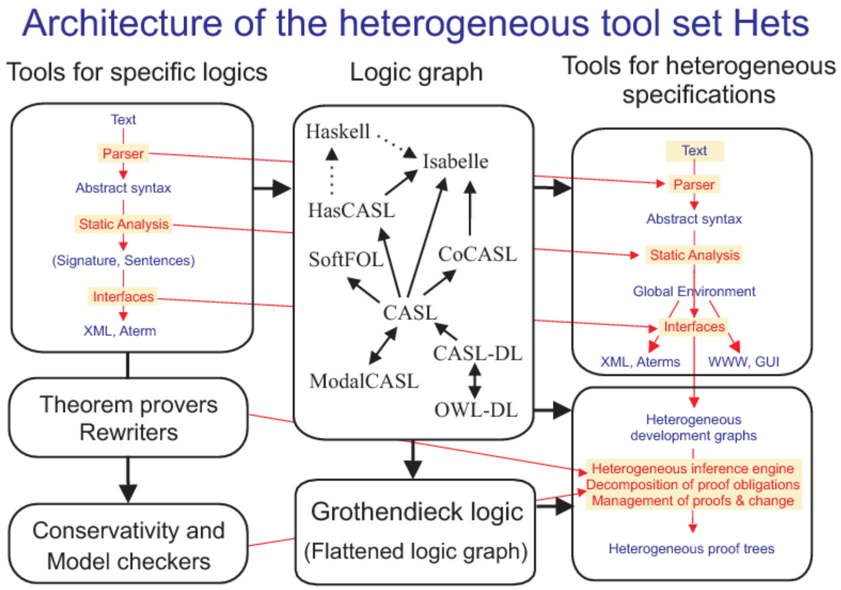


Figure 8: Architecture of the heterogeneous tool set.

The static analysis is based on the static analysis of basic specifications, and transforms an abstract syntax tree to a development graph (cf. Sect. 7 above). Starting with a node corresponding to the empty theory, it successively extends (using the static analysis of basic specifications) and/or translates (along the intra- and inter-logic translations) the theory, while simultaneously adding nodes and links to the development graph.

Heterogeneous proof management is done using heterogeneous development graphs, as described in Sect. 7. For local proof goals, logic-specific provers are invoked, see Sect. 10.

Hets can store development graphs, including their proofs. Therefore, Hets uses the so-called ATerm format [8], which is used as interchange format for interfacing with other tools.

More details can be found in [20, 26] and in the overview of modules provided in the developers section of the Hets home page at `http://www.dfki.de/sks/hets`.

Hets is mainly maintained by Christian Maeder (Christian.Maeder@dfki.de) and Till Mossakowski (Till.Mossakowski@dfki.de). The mailing list is

`hets-users@informatik.uni-bremen.de`

the homepage is

http://www.informatik.uni-bremen.de/mailman/listinfo/hets-users.

You need to subscribe to the list before you can send a mail. But note that subscription is very easy!

If your favourite logic is missing in HETS, please tell us (hets-users@informatik.uni-bremen.de). We will take account your feedback when deciding which logics and proof tools to integrate next into HETS. Help with integration of more logics and proof tools into HETS is also welcome.

Acknowledgement The heterogeneous tool set HETS would not have possible without cooperation with many people. Besides the authors, the following people have been involved in the implementation of HETS: Katja Abu-Dib, Mihai Codescu, Carsten Fischer, Jorina Freya Gerken, Rainer Grabbe, Sonja Gröning, Daniel Hausmann, Wiebke Herding, Hendrik Iben, Cui "Ken" Jian, Heng Jiang, Anton Kirilov, Tina Krausser, Martin Kühl, Mingyi Liu, Dominik Lücke, Maciek Makowski, Immanuel Normann, Razvan Pascanu, Daniel Pratsch, Felix Reckers, Markus Roggenbach, Pascal Schmidt, Paolo Torrini, René Wagner, Jian Chun Wang and Thiemo Wiedemeyer.

HETS has been built based on experiences with its precursors, CATS and MAYA. The CASL Tool Set (CATS) [22, 24] provides parsing and static analysis for CASL. It has been developed by the first author with help of Mark van den Brand, Kolyang, Bartek Klin, Pascal Schmidt and Frederic Voisin.

MAYA [4, 3] is a proof management tool based on development graphs. MAYA only supports development graphs without hiding and without logic translations. MAYA has been developed by Serge Autexier and Dieter Hutter.

We also want to thank Agnès Arnould, Thibaud Brunet, Pascale LeGall, Kathrin Hoffmann, Katiane Lopes, Stefan Merz, Maria Martins Moreira, Christophe Ringeissen, Markus Roggenbach, Dmitri Schamschurko, Lutz Schröder, Konstantin Tchekine and Stefan Wölfl for giving feedback about CATS, HOL-CASL and HETS. Finally, special thanks to Christoph Lüth and George Russell for help with connecting HETS to their UniForM workbench.

References

[1] D. Aspinal, S. Berghofer, P. Callaghan, P. Courtieu, C. Rafalli, and M. Wenzel. Emacs Proof General. Available at http://proofgeneral.inf.ed.ac.uk/.

[2] David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000.

[3] S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The development graph manager MAYA (system description). In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International*

Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, Proceedings, LNCS Vol. 2422, pages 495–502. Springer, 2002.

[4] Serge Autexier and Till Mossakowski. Integrating HOL-CASL into the development graph manager MAYA. In A. Armando, editor, *Frontiers of Combining Systems, 4th International Workshop, FroCoS 2002, Santa Margherita Ligure, Italy, Proceedings*, LNCS Vol. 2309, pages 2–17. Springer, 2002.

[5] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.

[6] M. Bidoit and P. D. Mosses. CASL *User Manual*, volume 2900 of *LNCS*. Springer, 2004. Free online version available at `http://www.cofi.info`.

[7] Patrick BLackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, Cambridge, England, 2001.

[8] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30:259–291, 2000.

[9] Free Software Foundation. Emacs. Available at `http://www.gnu.org/software/emacs/emacs.html`,.

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

[11] 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.

[12] Sonja Gröning. Beweisunterstützung für HasCASL in Isabelle /HOL. Master's thesis, University of Bremen, 2005. Diplomarbeit.

[13] Marc Herbstritt. zChaff: Modifications and extensions. report00188, Institut für Informatik, Universität Freiburg, July 17 2003. Thu, 17 Jul 2003 17:11:37 GET.

[14] Michael Kohlhase. *OMDoc - An Open Markup Format for Mathematical Documents [version 1.2]*, volume 4180 of *Lecture Notes in Computer Science*. Springer, 2006.

[15] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report. UU-CS-2001-35.

[16] K. Lüttich, T. Mossakowski, and B. Krieg-Brückner. Ontologies for the Semantic Web in CASL. In José Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, 17th International Workshop (WADT 2004)*, volume 3423 of *Lecture Notes in Computer Science*, pages 106–125. Springer; Berlin; `http://www.springer.de`, 2005.

[17] Klaus Lüttich and Till Mossakowski. Reasoning Support for CASL with Automated Theorem Proving Systems. WADT 2006, Springer LNCS, to appear.

[18] T. Mossakowski. HetCASL - heterogeneous specification. language summary, 2004.

[19] T. Mossakowski. ModalCASL - specification with multi-modal logics. language summary, 2004.

[20] T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Habilitation thesis, University of Bremen, 2005.

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

[22] Till Mossakowski. CASL: From semantics to tools. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Berlin, Germany, Proceedings*, LNCS Vol. 1785, pages 93–108. Springer, 2000.

[23] Till Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 286:367–475, 2002.

[24] Till Mossakowski, Kolyang, and Bernd Krieg-Brückner. Static semantic analysis and theorem proving for CASL. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, 1997, Selected Papers*, LNCS Vol. 1376, pages 333–348. Springer, 1998.

[25] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Orna Grumberg and Michael Huth, editors, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer-Verlag Heidelberg, 2007.

[26] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Bernhard Beckert, editor, *VERIFY 2007*, volume 259 of *CEUR Workshop Proceedings*. 2007.

[27] Till Mossakowski, Lutz Schröder, Markus Roggenbach, and Horst Reichel. Algebraic-co-algebraic specification in CoCASL. *Journal of Logic and Algebraic Programming*. To appear.

[28] Peter D. Mosses, editor. *CASL Reference Manual*, volume 2960 of *Lecture Notes in Computer Science*. Springer, 2004. Free online version available at `http://www.cofi.info`.

[29] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002.

[30] S. Peyton-Jones, editor. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge, 2003. also: J. Funct. Programming **13** (2003).

[31] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Haskell Workshop*. 1997.

[32] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.

[33] M. Roggenbach. CSP-CASL - a new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42–71, 2006.

[34] L. Schröder, T. Mossakowski, and C. Maeder. HasCasl – Integrated functional specification and programming. Language summary. Available at `http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/HasCASL`, 2003.

[35] Lutz Schröder. Higher order and reactive algebraic specification and development. Summary of papers constituting a cumulative habilitation thesis; available under `http://www.informatik.uni-bremen.de/~lschrode/papers/Summary.pdf`, 2005.

[36] Lutz Schröder. The HasCASL prologue - categorical syntax and semantics of the partial λ-calculus. *Theoret. Comput. Sci.*, 353:1–25, 2006.

[37] Lutz Schröder and Till Mossakowski. HasCasl: Towards integrated specification and development of Haskell programs. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, Proceedings*, LNCS Vol. 2422, pages 99–116. Springer, 2002.

[38] Geoff Sutcliffe and Christian B. Suttner. Evaluating general purpose automated theorem proving systems. *Artificial Intelligence*, 131(1-2):39–54, 2001.

[39] Geoff Sutcliffe and Christian B. Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.

[40] `http://www.xemacs.org/People/index.html`. XEmacs. Available at `http://www.xemacs.org/`.

[41] Paolo Torrini, Christoph Lüth, Christian Maeder, and Till Mossakowski. Translating Haskell to Isabelle. Isabelle workshop at CADE-21, 2007.

[42] Jan Oliver Wallgrün, Lutz Frommberger, Frank Dylla, and Diedrich Wolter. SparQ user manual v0.6.2. University of Bremen, 2006.

[43] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topic. SPASS version 2.0. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 275–279. Springer-Verlag, July 27-30 2002.

[44] Jürgen Zimmer and Serge Autexier. The MathServe System for Semantic Web Reasoning Services. In U. Furbach and N. Shankar, editors, *3rd IJCAR*, LNCS 4130. Springer, 2006.