Hets User Guide

– Version 0.99 –

Till Mossakowski, Christian Maeder, Mihai Codescu

DFKI GmbH, Bremen, Germany.

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

August 14, 2016

1 Introduction

The central idea of the Heterogeneous Tool Set (Hets) is to provide an open source general framework for formal methods integration and proof management. One can think of Hets acting like a motherboard where different expansion cards can be plugged in, the expansion cards here being individual logics (with their analysis and proof tools) as well as logic translations. Individual logics and their analysis and proof tools can be plugged into the Hets motherboard using an object-oriented interface based on institutions [19]. The Hets motherboard already has plugged in a number of expansion cards (e.g., the theorem provers Isabelle, SPASS and more, as well as model finders). Hence, a variety of tools is available, without the need to hard-wire each tool to the logic at hand.

Hets supports a number of input languages directly, such as Casl, Common Logic, OWL 2, LF, THF, HOL, Haskell, and Maude. For heterogeneous specification, Hets offers the language heterogeneous Casl. Heterogeneous Casl (HetCasl) generalises the structuring constructs of Casl [11, 38] to arbitrary logics (if they are formalised as institutions and plugged into the Hets motherboard), as well as to heterogeneous combination of specification written in different logics. See Fig. 2 for a simple subset of the HetCasl syntax, where *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. 3, and the degree of support by Hets in Fig. 4.

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. *Specifi-*

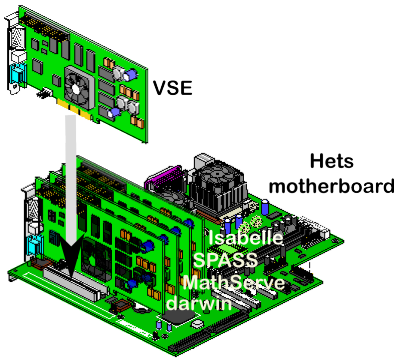

Figure 1: The Hets motherboard and some expansion cards

```
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 2: 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.

cation 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 [19], which formalize the notion of a logic. The theory behind Hets is laid out in [30]. A short overview of Hets is given in [35, 36].

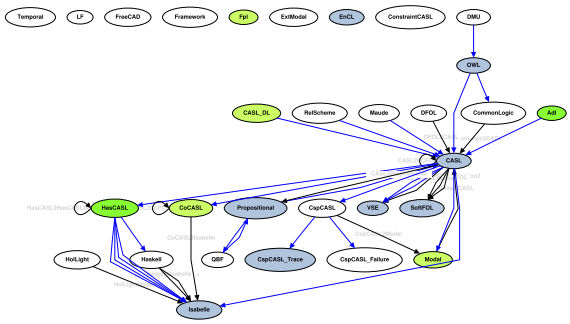

Figure 3: Graph of logics currently supported by Hets. The more an ellipse is filled with green, the more stable is the implementation of the logic. Blue indicates a prover-supported logic.

2 Logics supported by Hets

The following list of logics (formalized as so-called institutions [19]) 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 [38, 11]. We have implemented the Casl logic in such a way that much of the implementation can be reused 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 [37] is a coalgebraic extension of Casl, suited for the specification of process types and reactive systems. The central proof method is coinduction.

ModalCASL [29] 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.

ExtModal is an extended modal logic, subsuming and replacing ModalCASL. It features – apart from multiple modalities and dynamic logic – also time (and term) modalities, grading, hybrid modal logic, and the μ-calculus [17]. Comorphisms to CASL and THF have been implemented.

Language	Parser	Static Analysis	Prover
CASL	x	x	-
COCASL	x	x	-
MODALCASL	x	x	-
HASCASL	x	x	-
Haskell	(x)	x	-
CspCASL	x	x	-
CspCASL_Trace	-	-	x
CspCASL_Failure	-	-	-
CommonLogic	x	x	-
CONSTRAINTCASL	x	(x)	-
Temporal	x	(x)	-
RelScheme	x	(x)	-
DFOL	x	(x)	-
ExtModal	x	(x)	-
LF	x	(x)	-
CASL-DL	x	-	-
DMU	x	-	-
FreeCAD	-	x	-
OWL 2	x	x	x
Propositional	x	x	x
QBF	x	x	x
SoftFOL	x	-	x
Maude	x	x	-
VSE	x	x	x
THF	x	x	x
ISABELLE	(x)	-	x
HolLight	x	x	-
Adl	x	x	-
Fpl	x	x	-
EnCL	x	x	x
Hybrid	x	x	-
Hybridize	x	-	-

Figure 4: Current degree of HETS support for the different languages. Languages without prover can still "borrow" provers via logic translations.

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 [52]; the language is summarized in [49], the semantics in [51, 50].

Haskell is a modern, pure and strongly typed functional programming language. With various language extensions it simultaneously is the implementation language of HETS. As a logic we only supports the older Haskell 98 standard. Yet, in principle, HETS might be applied to itself – in some more distant future. The definitive reference for Haskell is [42], see also `www.haskell.org`.

CspCASL [46] is a combination of CASL with the process algebra CSP.

CommonLogic `http://en.wikipedia.org/wiki/Common_logic`

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

OWL 2 is the Web Ontology Language (OWL 2) recommended by the World Wide Web Consortium (W3C, `http://www.w3c.org`). It is used for knowledge representation and the Semantic Web [10]. Hets calls an external OWL 2 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 [24] is an extension of a restriction of CASL, realizing a strongly typed variant of OWL 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 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 and CASL closer together.

Propositional is classical propositional logic, with the zChaff SAT solver [21] connected to it.

QBF are quantified boolean formulas, with DepQBF `http://fmv.jku.at/depqbf/` connected to it.

RelScheme is a logic for relational databases [48].

SoftFOL [25] offers several automated theorem proving (ATP) systems for first-order logic with equality:

- SPASS [60], see `http://www.spass-prover.org`;

- Vampire [45] see `http://www.vprover.org`;
- Darwin [8], see `http://combination.cs.uiowa.edu/Darwin`;
- Eprover [53], see `http://www.eprover.org`;
- E-KRHyper [41], see `http://www.uni-koblenz.de/~bpelzer/ekrhyper`, and
- MathServe Broker[1] [61].

These together comprise some of the most advanced theorem provers for first-order logic. SoftFOL is essentially the first-order interchange language TPTP [54], see `http://www.tptp.org`.

THF simply typed lambda calculus, is an interchange language for (typed) higher-order logic [9], similar to what TPTP is for (untyped) first-order logic. HETS connects THF to the automated higher-order provers Leo-II, Satallax and Isabelle's nitpick, refute and sledgehammer.

ISABELLE [40] is an interactive theorem prover for higher-order logic.

HolLight `http://www.cl.cam.ac.uk/~jrh13/hol-light/` is John Harrison's interactive theorem prover for higher-order logic.

VSE is an interactive theorem prover, see 11.4.

DMU is a dummy logic to read output of "Computer Aided Three-dimensional Interactive Application" (Catia).

FreeCAD is a logic to read design files of the CAD system FreeCAD `http://sourceforge.net/projects/free-cad`.

Maude is a rewrite system `http://maude.cs.uiuc.edu/` for first-order logic. In order to use this logic the environment variable `HETS_MAUDE_LIB` must be set to a directory containing the files `full-maude.maude`, `hets.prj`, `maude2haskell.maude` and `parsing.maude`.

DFOL is an extension of first-order logic with dependent types [44].

LF is the dependent type theory of Twelf `http://twelf.plparty.org/`. Hets calls Twelf on `.elf` files (for this, the environment variable `TWELF_LIB` must be set) and reads in the OMDoc generated by Twelf. Moreover, LF can be used as a logical framework to add new logics in Hets [14]. Logic definitions in LF are based in the logic atlas of the Latin project [27] and therefore the environment variable `LATIN_LIB` must be set to the repository with the Latin logic definitions.

Framework is a dummy logic added for declarative logic definitions [14].

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

Adl is "A Description Language" based on relational algebra originally designed for requirements engineering of business rules `https://lab.cs.ru.nl/BusinessRules/Requirements_engineering`.

Fpl is a "logic for functional programs" as an extension of a restriction of CASL (predicates are disabled) [47].

EnCL is an "engineering calculation language" based on first order theory of real numbers with some predefined binders [15]. It allows the formulation of executable specifications of engineering calculation methods. For the execution of these specifications Hets provides connections to the computer algebra systems Mathematica, Maple and Reduce.

Hybrid HybridCASL [39] extends ModalCASL by implementing the characteristic features of hybrid logic, both at the level of syntax and semantics. A comorphism from HybridCASL to CASL is provided.

The method of hybridisation of arbitrary institutions documented in [26], providing a method to automatically combine the standard properties of hybrid logics with any logic already integrated in Hets. Some manual intervention is still required: a parser and a semantics analyser for the sentences of the base logic need to be declared at source code level. Currently, the hybridised versions of the following logics may be used in a fully automatic way: Propositional, CASL, CoCASL, and any other logic already hybridised.

Various logics are supported with proof tools. 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 [11]; the detailed language reference is given in the CASL Reference Manual [38]. 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 [28]; 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.

3 Logic translations supported by Hets

Logic translations (formalized as institution comorphisms [18]) 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. 5.

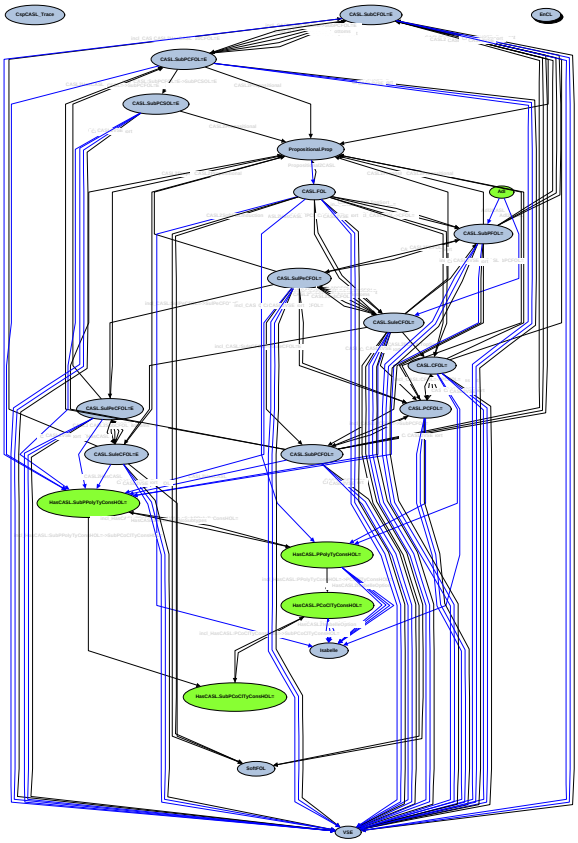

Figure 5: Graph of most important sublogics currently supported by Hets, together with their comorphisms.

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

Adl2CASL	inclusion taking relations to CASL predicates
CASL2CoCASL	inclusion
CASL2CspCASL	inclusion
CASL2HasCASL	inclusion
CASL2Isabelle	inclusion on sublogic CFOL= (translation (7) of [33])
CASL2Modal	inclusion
CASL2PCFOL	coding of subsorting (SubPCFOL=) by injections, see Chap. III:3.1 of the CASL Reference Manual [38]
CASL2PCFOLTopSort	coding of subsorting (SulPeCFOL=) by a top sort and unary predicates for the subsorts
CASL2Propositional	translation of propositional FOL
CASL2SoftFOL	coding of CASL.SuleCFOL=E to SoftFOL [25], mapping types to soft types
CASL2SoftFOLInduction	same as CASL2SoftFOL but with instances of induction axioms for all proof goals
CASL2SoftFOLInduction2	similar to CASL2SoftFOLInduction but replaces goals with induction premises
CASL2SubCFOL	coding of partial functions by error elements (translation ($4a'$) of [33], but extended to subsorting, i.e. sublogic SubPCFOL=)
CASL2VSE	inclusion on sublogic CFOL=
CASL2VSEImport	inclusion on sublogic CFOL=
CASL2VSERefine	refining translation of CASL.CFOL= to VSE
CASL_DL2CASL	inclusion
CoCASL2CoPCFOL	coding of subsorting by injections, similar to CASL2PCFOL
CoCASL2CoSubCFOL	coding of partial functions by error supersorts, similar to CASL2SubCFOL
CoCASL2Isabelle	extended translation similar to CASL2Isabelle
CommonLogic2CASL	Coding Common Logic to CASL.Module elimination is applied before translating to CASL.
CommonLogic2CASLCompact	Coding compact Common Logic to CASL. Compact Common Logic is a sublogic of Common Logic where no sequence markers occur. Module elimination is applied before translating to CASL. We recommend using this comorphism whenever possible because it results in simpler specifications.
CommonLogicModuleElimination	Eliminating modules from a Common Logic theory resulting in an equivalent specification without modules.
CspCASL2CspCASL_Failure	inclusion
CspCASL2CspCASL_Trace	inclusion
CspCASL2Modal	translating the CASL data part to ModalCASL
DFOL2CASL	translating dependent types
DMU2OWL	interpreting Catia output as OWL

HasCASL2HasCASLNoSubtypes	coding out subtypes
HasCASL2HasCASLPrograms	coding of HasCasl axiomatic recursive definitions as HasCasl recursive program definitions
HasCASL2Haskell	translation of HasCasl recursive program definitions to Haskell
HasCASL2IsabelleOption	coding of HasCASL to Isabelle/HOL [20]
Haskell2Isabelle	coding of Haskell to Isabelle/HOL [58]
Haskell2IsabelleHOLCF	coding of Haskell to Isabelle/HOLCF [58]
HolLight2Isabelle	coding of HolLight to Isabelle/HOL
Maude2CASL	encoding of rewrites as predicates
Modal2CASL	the standard translation of modal logic to first-order logic [12]
OWL2CASL	inclusion
OWL2CommonLogic	inclusion
Propositional2CASL	inclusion
Propositional2QBF	inclusion
QBF2Propositional	inclusion
RelScheme2CASL	inclusion

4 Getting started

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

```
http://hets.eu
```

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

Hets is currently available (on Intel architectures only) for Linux and with limited functionality for Mac OSX.

There are several possibilities to install Hets.

1. The best support is currently given via Ubuntu packages.

   ```
   sudo apt-add-repository ppa:hets/hets
   sudo apt-get update
   sudo apt-get install hets
   ```

   This will also install quite a couple of tools for proving requiring about 800 MB of disk space. For a minimal installation `apt-get install hets-core` instead of `hets`.

   The following software will be installed, too:

Hets-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	(version 8.4 or 8.5)
SPASS	theorem prover	`http://spass.mpi-sb.mpg.de/`
Darwin	theorem prover	`http://combination.cs.uiowa.edu/Darwin/`

A daily snapshot of `hets` can be installed by:

```
sudo hets -update
```

In case the binary (under `/usr/lib/hets/hets`) is broken it can be replaced manually or by reverting an update:

```
sudo hets -revert
```

2. For Mac OSX we provide disk images `Hets-<date>.dmg` without GTK support.

3. You may compile HETS from the sources (they are licensed under GPL), 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, but be prepared to take some time.

Depending on your application further tools are supported and may be installed in addition:

ISABELLE	theorem prover	`http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`
(X)Emacs	editor (for Isabelle)	
zChaff	SAT solver	`http://www.princeton.edu/~chaff/zchaff.html`
minisat	SAT solver	`http://minisat.se/`
Pellet	OWL 2 reasoner	`http://clarkparsia.com/pellet/`
E-KRHyper	theorem prover	`http://userpages.uni-koblenz.de/~bpelzer/ekrhyper/`
Reduce	computer algebra system	`http://www.reduce-algebra.com/`
Maude	rewrite system	`http://maude.cs.uiuc.edu/`
VSE	theorem prover	(non-public)
Twelf		`http://twelf.plparty.org/`

5 Analysis of Specifications

Consider the following CASL specification:

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

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 as `Hets-lib/UserManual/Chapter3.casl`). 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 and no development is created.

`-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 a colon separated list of directories for specification libraries (equivalently, you can set the variable `HETS_LIB` before calling HETS).

`-a ANALYSIS, --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
`.hol`	HolLight	HolLight
`.hs`	Haskell	Haskell
`.owl`	OWL 2	OWL
`.elf`	LF	Twelf
`.clf` or `.clif`	CommonLogic	CASL
`.maude`	Maude	Maude

Furthermore, `.xml` files are accepted as Catia output if the default logic is set to DMU before a library import or by the "`-l DMU`" command line option of HETS. In all other cases `.xml` files are assumed to be development graph files as produced by "`-o xml`".

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. 3). (There are also heterogeneous constructs for explicit translations between logics, see [28].)

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 [31, 30] 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 [30] stating soundness and relative completeness of the proof calculus for heterogeneous development graphs.

Details can be found in the CASL Reference Manual [38, IV:4] and in [30, 31, 36].

The following options let HETS show the development graph of a specification library:

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

`-u, --uncolored` no colors in shown graphs

The following additional options also apply typical rules from the development graph calculus to the final graph and save applying these rule via the GUI.

`-A, --apply-automatic-rule` apply the automatic strategy to the development graph. This is what you usually want in order to get goals within nodes for proving.

`-N, --normal-form` compute all normal forms for nodes with incoming hiding links. (This may take long and may not be implemented for all logics.)

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.

Yellow nodes have an open conservativity proof obligations.

Green nodes have all proof obligations resolved.

Black links correspond to reference to other specifications (definition links in the sense of [38, IV:4]).

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

Green links correspond to proven theorem links.

Yellow links correspond to proven theorem links with open conservativity or to open hiding theorem links.

Blue links correspond to hiding, free, or cofree definition links.

Violett links correspond to a mixture of links becoming visible after "expand" or "Show unnamed nodes with open proofs".

Solid links correspond to global (definition or theorem) links in the sense of [38, IV:4].

Dashed links correspond to local (theorem) links in the sense of [38, IV:4]. These are usually created after "Global-Decomposition" or only be visible after "Show newly added proven edges".

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

Double line 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)

Hide/show names/nodes/edges The "Hide/show names/nodes/edges" menu is a toggle: you can switch on or off the display of node names, unnamed nodes or proven theorem links.

With the "Hide/show internal node names" option, the nodes that are initially unnamed get derived names.

With the "Hide/show unnamed nodes without open proofs" option, it is possible to reveal the unnamed nodes which do not have open proof goals. Initially, the complexity of the graph is reduced by hiding all these nodes; only nodes corresponding to named specifications are displayed. Paths between named nodes going through unnamed nodes are displayed as edges; these paths are then expanded when showing the unnamed nodes.

When applying the development graph calculus rules, theorem links that have been proven are removed from the graph. With the "Hide/Show newly added proven edges" option, it is possible to re-display these links; they are marked as proven in the link info (see *Pop-up menu for links*, below).

Focus node This menu is particularly useful when navigating in a large development graph, which does not fit on a single screen. The list of all nodes is displayed: the nodes are identified by the internal node number and the internal node name. Once a node is selected, the view centers on it.

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

Select Linktypes This menu allows to select the type of links that are displayed in the development graph. A selection window appears, where links are grouped into three categories: definition links, proven theorem links and unproven theorem links. It is possible to select/deselect all links or to invert the current selection.

Consistency checker Checks whether the theories of the nodes of the graph are consistent i.e. have a model. The model finders currently interfaced are Isabelle-refute, darwin and E-KRHyper, with best support for darwin.

Proofs This menu allows to apply some of the deduction rules for development graphs, see Sect. IV:4.4 of the CASL Reference Manual [38] or one of [30, 31, 36]. 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 "Auto-DG-Prover", which automatically applies the rules in the correct order. As a result, the open theorem links (marked in red) will be reduced to local proof goals, that is, they become green, and instead, some target nodes may 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). Also, a CASL-specific normalisation of free links has been implemented.

Dump Development Graph This option is available only for debugging purposes; it outputs a textual representation of the development graph.

Show Library Graph This menu displays the library graph, in a separate window, if the library graph window has been closed after HETS has been called.

Save Graph for uDrawGraph Saves the development graph in a .udg file which can be later read by uDrawGraph.

Save proof-script This menu saves the proof rules that have been applied to the current development graph in a .hpf file which can be later read by HETS and thus the action performed on the graph are saved.

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

Show node info Shows the local informations of the node: the internal node name and node number, the xpath that denotes the location of the node within an XML representation, information about consistency of the node, origin of the node and the local theory i.e. axioms declared locally.

Show theory Shows the theory of the node (including axioms imported from other nodes). Notice that 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 normal form of the node, by calling *Proofs/Compute Normal Form*. For such nodes, a warning is displayed.

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 proof status Show open and proven local proof goals.

Prove Try to prove the local proof goals. See Section 11 for details.

Prove VSE structured Allows to send a development graph below the current node to the interactive `hetsvse` prover if that binary is available, see 11.4.

Disprove Negates selected goals and tries to disprove them using consistency checkers. Other goals will be treated like axioms if "Include Theorems" is selected. (If a theory is consistent with a negated goal, the goal is disproven.)

Add sentence This menu allows to add a sentence on the fly. The (possibly named) sentence will be parsed and analysed using the underlying logic.

Check consistency Simply calls the global "Consistency checker" menu for the current node, see 11.1.

Check conservativity Checks conservativity of the inclusion morphism from the empty theory to the theory of the node (see **Check conervativity** for edges).

For the nodes which are references to specifications from an external library, the pop-up menu options are reduced to **Show node info, Show theory, Show proof status** and **Prove** and moroever, the option **Show referenced library** is added: on selection, it displays in a new window the development graph of the external library from which the specification has been downloaded.

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

Show info Shows informations about the edge: internal number and internal nodes it links, the link type and origin and the signature morphism of the link. The latter 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.

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.

Expand This menu is available only for paths going through unnamed nodes which are not displayed and it expands the path to the links forming it.

Besides development graphs there are library graph windows displaying the library hierarchy. The Edit menu has the following submenus:

Edit This menu for library graphs has the following submenus:

Reload Library Reloads all HetCasl sources in order to avoid closing and restarting the application after sources have changed. However, all previous proof steps will be lost, therefore you have to confirm this action. (A change management tool to keep proofs is in preparation.)

Experimental reload changed Library This button is supposed to interface our change management tool (in order to preserve proof information) but does not work yet.

Translate Library Translates a library along a comorphism to be chosen. This only works for a homogeneous library hierarchy. A finer grained alternative is to use "Translate theory" for individual nodes. The original state and proof steps will be lost, therefore you have to confirm this action.

Show Logic Graph Shows the graph of logics and logic comorphisms currently supported by Hets. The Edit menu of a logic graph window has the following submenu:

Show detailed logic graph Shows the important sublogics and comorphims between them, i.e. translation (blue links) and inclusion (black links).

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 explicit type of the input file. By default `env`, `casl`, or `het` extensions are tried in this order. An `env` file contains a shared ATerm of a development graph, whereas `casl` or `het` files contain plain HetCasl text. An `env` file will always be read if it exists and is consistent (aka newer) than the corresponding HetCasl file.

`exp` files contain a development graph in a new experimental omdoc format. `prf` files contain additional development steps (as shared ATerms)

to be applied on top of an underlying development graph created from a corresponding `env`, `casl`, or `het` file. `hpf` files are plain text files representing heterogeneous proof scripts. The contents of a `hpf` file must be valid input for Hets in interactive mode. (`gen_trm` formats are currently not supported.)

The possible input types are:

```
  casl
| het
| owl
| hs
| exp
| maude
| elf
| hol
| prf
| omdoc
| hpf
| clf
| clif
| xml
| [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
| omn
| omdoc
| xml
| exp
| hs
| thy
| comptable.xml
| (sig|th)[.delta]
| pp.(het|tex|xml|html)
| graph.(exp.dot|dot)
| dfg[.c]
| tptp[.c]
```

The `env` and `prf` formats are for subsequent reading, avoiding the need to re-analyse downloaded libraries. `prf` files can also be stored or loaded via the GUI's File menu.

The `omn` option [22] will produce OWL files in Manchester Syntax for each specification of a structured OWL library.

The `omdoc` format [22] 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 `xml` option will produce an XML-version of the development graph for our change management broker.

The `exp` format is the new experimental omdoc format.

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 `sig` or `th` option will create HETCASL signature or theory files for each development graph node. (The `.delta` extension is not supported, yet.)

The `pp` format is for pretty printing, either as plain text (`het`), LaTeXinput (`tex`), HTML (`html`) or XML (`xml`). For example, it is possible to generate a pretty printed LaTeX version of `Order.casl` by typing:

```
hets -v2 -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 format `pp.xml` represents just a parsed library in XML.

Formats with `graph` are for future usage.

The `dfg` format is used by the SPASS theorem prover [60].

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

Appending `.c` to `dfg` or `tptp` will create files for consistency checks by SPASS or Darwin respectively.

For all output formats it is recommended to increase the verbosity to at least level 2 (by using the option `-v2`) to get feedback which files are actually written. (`-v2` also shows which files are read.)

`-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)

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

`-w NVIEWS`, `--view=NVIEWS` chooses a list of named views for processing

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

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

`-X`, `--server` run Hets as web server (see 9)

`-x`, `--xml` use xml-pgip packets to communicate with Hets in interactive mode

`-S PORT`, `--listen=PORT` communicate with Hets in interactive mode vy listining to the port `PORT`

`-c HOSTNAME:PORT`, `--connect=HOSTNAME:PORT` communicate with Hets in interactive mode via connecting to the port on host `HOSTNAME`

`-d STRING`, `--dump=STRING` produces implementation dependent output for debugging purposes only (i.e. `-d LogicGraph` lists the logics and comorphisms)

9 Hets as a web server

Large parts of Hets are now also available via a web interface. A running server should be accessible on `http://pollux.informatik.uni-bremen.de:8000/`. It allows to browse the Hets library, upload a file or just a HetCasl specification. Development graphs for well-formed specifications can be displayed in various formats where the `svg` format is supposed to look like the graphs displayed by uDrawGraph. Besides browsing, the web server is supposed to be accessed by other programs using queries. The possible queries are described on `http://trac.informatik.uni-bremen.de:8080/hets/wiki/RESTfulInterface`.

A development graph is addressed by the *path* following the port number and the slash of the URL, i.e. `http://localhost:8000/Basic/Numbers.casl`. Once a development has been created it can be accessed via a (fairly unique) session id (consisting of nine digits) that can be used as *path*.

A *path* may be followed by a query string that begins with a question mark and consists of *entries* (usually field-value pairs) separated by ampersands. The queries control the information to be extracted from the development graph given by the *path* or they allow to perform commands on the graph.

Usually, query string are made up of `field=value` pairs, but in some cases the field name or the value may be omitted and in that case the equal sign must be omitted, too.

For instance strings denoting formats, like `xml`, `svg`, `pdf`, etc., do not need to be preceded by `format=`. Some formats, like `pdf`, only pretty print specification and basically ignore the underlying development graph.

A special *entry* is just `session` which only returns a fresh session id for a development graph that is given by a file name, i.e. `http://localhost:8000/Basic/Numbers.casl?session`. These session ids must be used to perform commands (of the development graph calculus) that *change* the underlying graph.

Given a graph, nodes and edges can be addressed by numbers via entries like `node=0` or `edge=0`. (Nodes can also be given by name.) For nodes, prover actions are possible by further entries. `http://localhost:8000/123456789?prove=Nat__E1&prover=SPASS&timeout=5` would try to prove the goals of the node `Nat__E1` using the prover `SPASS` with a timeout of 5 seconds for the development graph that happened to have the (unlikely) session id `123456789`. Individual goals can be given via a `theorems` field and special translations by a `translation` field. The available provers and translations can be queried by `?node=0&translations` and `?node=0&provers` or shorter by `?translations=0` and `?provers=0`, where instead of the node number (here `0`) also a node name can be used.

10 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.

`-e ENCODING, --encoding=ENCODING` Read input files using latin1 or utf8 encoding. The default is still latin1.

`--unlit` Read literate input files.

`--relative-positions` Just uses the relative library name in positions of warning or errors.

`-U FILE, --xupdate=FILE` update a development graph according to special xml update information (still experimental).

`-m FILE, --modelSparQ=FILE` model check a qualitative calculus given in SparQ lisp notation [59] against a CASL specification

11 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, i.e. by using the "Prove" menu of a red node.

The graphical user interface (GUI) for calling a prover is shown in Fig. 6 — we call it "Proof Management GUI". The top 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 GUI elements are the following:

- The button 'Display' shows the selected goals in the ASCII syntax of this theory's logic in a separate window.
- By pressing the '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.
- With the 'Prove' button the actual prover is launched. This is described in more detail in the paragraphs below.
- The list 'Pick Theorem Prover:' lets you choose one of the connected provers (among them 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 pop-up choice box below 'Selected comorphism path:' lets you pick a (composed) comorphism to be used for the chosen prover.
- 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 in the bottom lists 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.

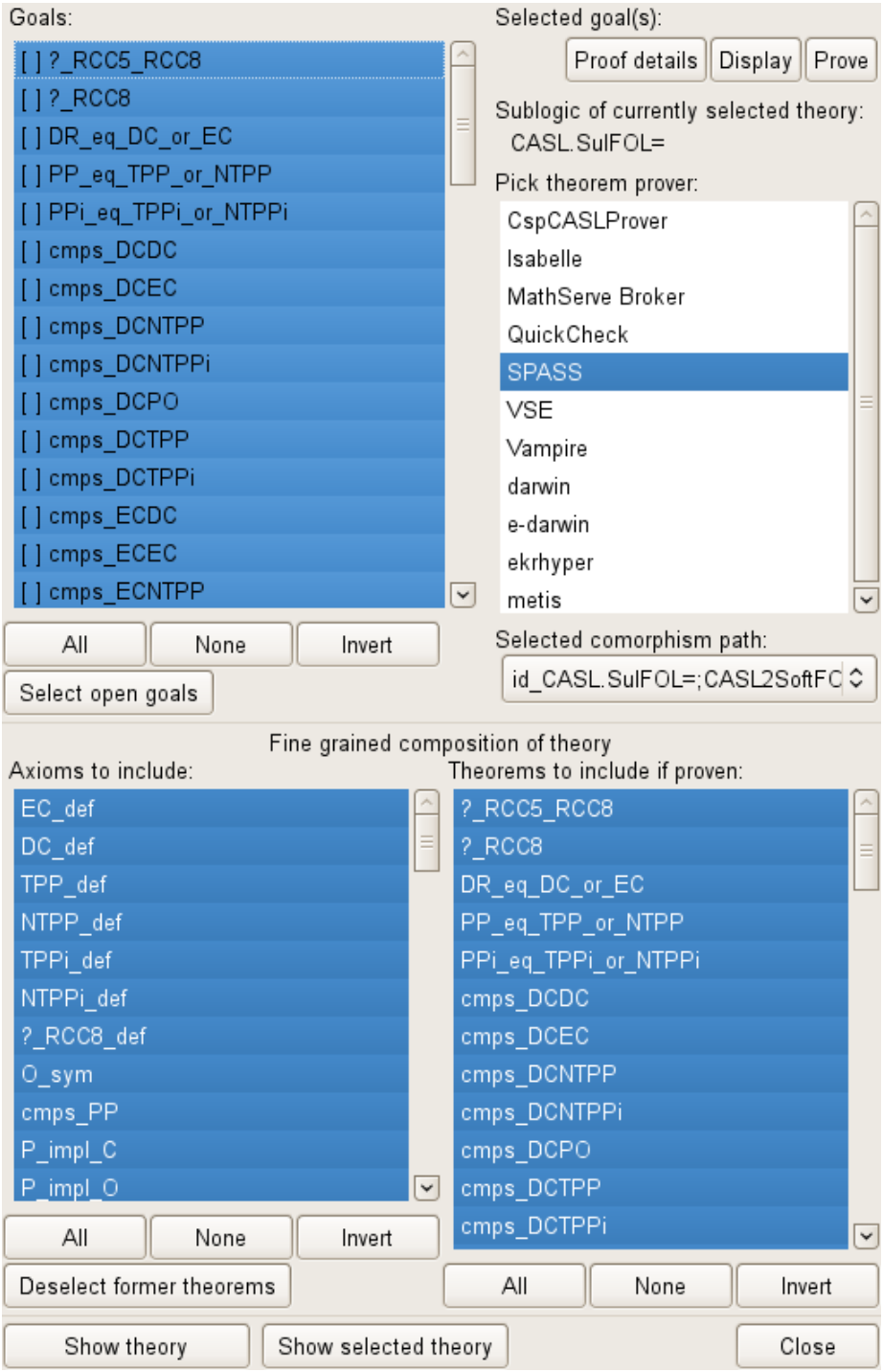

Figure 6: Hets Goal and Prover Interface

- If you press the bottom-right '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.
- All other buttons control selecting list entries

11.1 Consistency Checker

Since proofs are void if specifications are inconsistent, the consistency should be checked (if possible for the given logic) by the "Consistency checker" shown in Fig. 7. This GUI is invoked from the 'Edit' menu as it operates on all nodes.

The list on the left shows all node names prefixed with a consistency status in square brackets that is initially empty. A consistent node is indicated by a '+', a '-' indicates an inconsistent node, a 't' denotes a timeout of the last checking attempt.

For some selection of nodes (of a common logic) a model finder should be selectable from the 'Pick Model finder:' list. Currently only for "darwin" some CASL models can be re-constructed. When pressing 'Check', possibly after 'Select comorphism path:', all selected nodes will be checked, spending at most the number of seconds given under 'Timeout:' on each node. Pressing 'Stop' allows to terminate this process if too many nodes have been chosen. Either by 'View results' or automatically the 'Results of consistency check' (Fig. 8) will pop up and allow you to inspect the models for nodes, if they could be constructed.

11.2 Automated Theorem Proving Systems (Logic SoftFOL)

All ATPs integrated into HETS share the same GUI, with only a slight modification for the MathServe Broker: the input field for extra options is inactive. Figure 9 shows the instantiation for SPASS, where in the top right part of the window the batch mode can be controlled. The left side shows the list of goals (with status indicators). If goals are timed out (indicated by 't') it may help to activate the check box 'Include preceeding proven theorems in next proof attempt' and pressing 'Prove all' again.

On the bottom right the result of the last proof attempt is displayed. The 'Status:' indicates 'Open', 'Proved', 'Disproved', 'Open (Time is up!)', or 'Proved (Theory inconsistent!)'. The list of 'Used Axioms:' is filled by SPASS. The button 'Show Details' shows the whole output of the ATP system. The 'Save' buttons allow you to save the input and configuration of each proof for documentation. By 'Close' the results for all goals are transferred back to the Proof Management GUI.

The MathServe system [61] 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 [55]. Only two

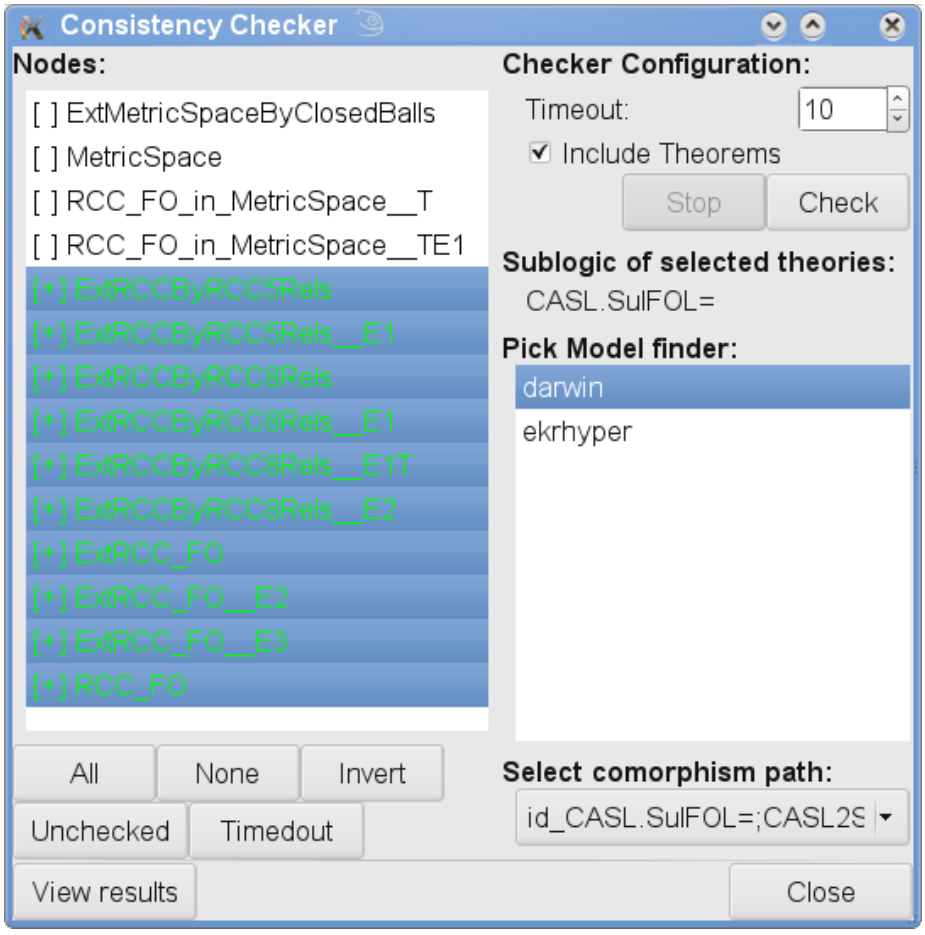

Figure 7: Hets Consistency Checker Interface

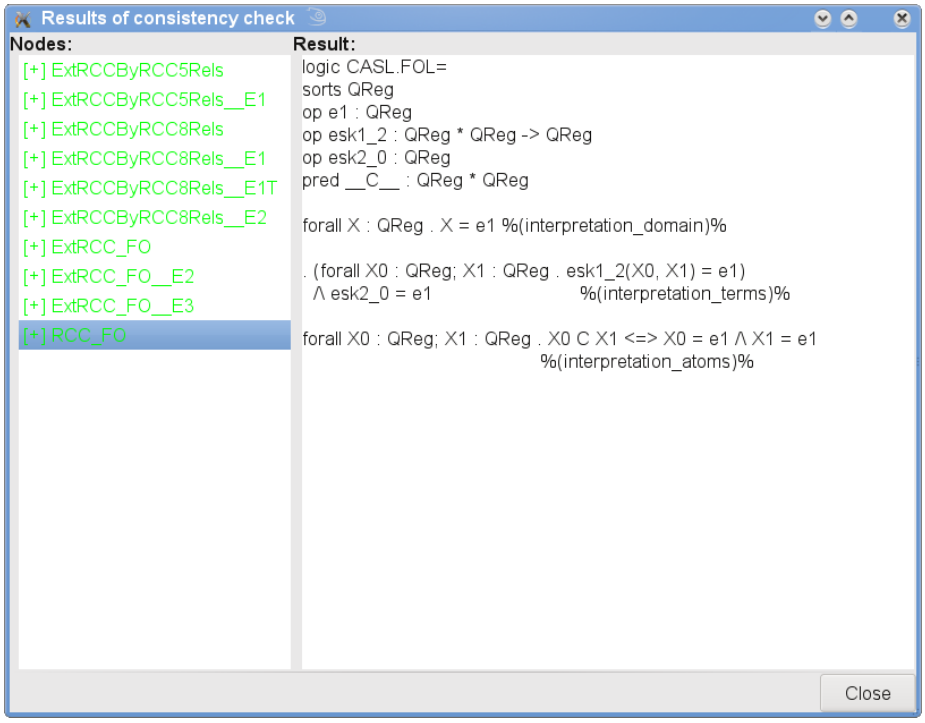

Figure 8: Consistency Checker Results

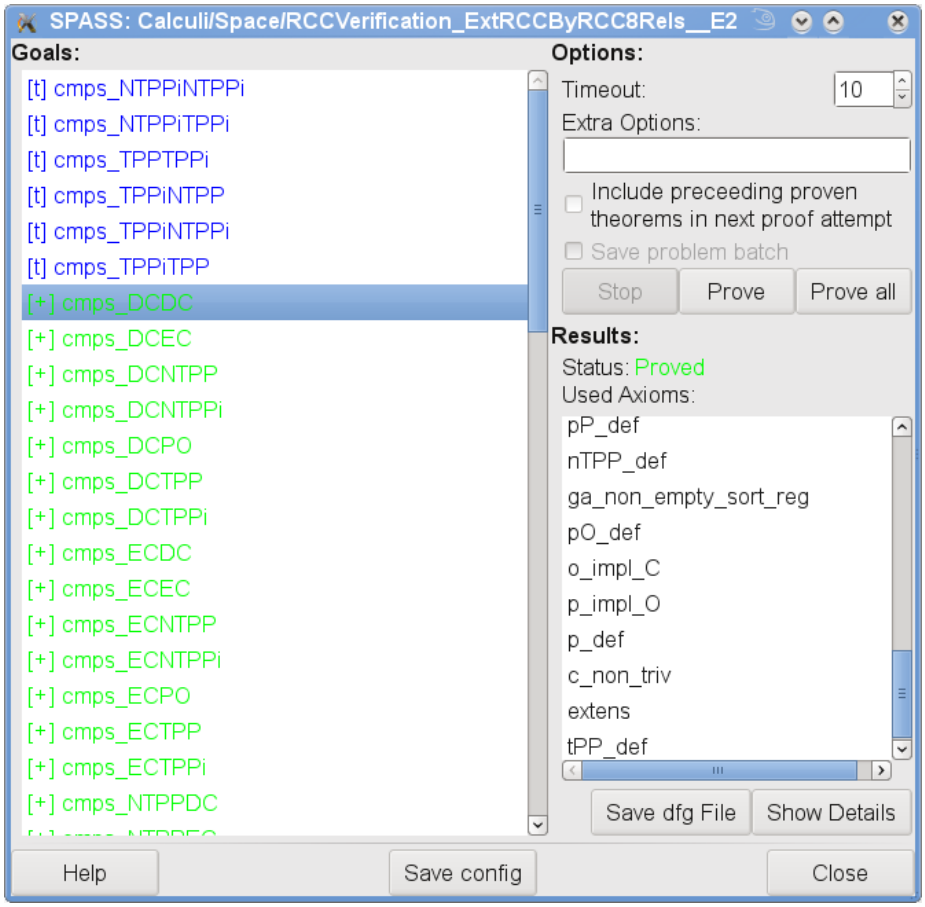

Figure 9: Interface of the SPASS prover

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 [60] 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) [61]. The TPTP format has been introduced by Sutcliffe and Suttner for the annual competition CASC [56] and

provides a unified syntax for untyped FOL with equality, but without any symbol declaration.

11.3 Isabelle

Isabelle [40] 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 [16, 57]. The Isabelle theory file conforms to the Isabelle/Isar syntax [40]. 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).

11.4 VSE

The specification environment Verification Support Environment (VSE) [3], developed at DFKI Saarbrücken, provides an industrial-strength methodology for specification and verification of imperative programs. VSE provides an interactive prover, which supports a Gentzen style natural deduction calculus for dynamic logic. This logic is an extension of first-order logic with two additional kinds of formulas that allow for reasoning about programs. One of them is the box formula $[\alpha]\varphi$, where α is a program written in an imperative language, and φ is a dynamic logic formula. The meaning of $[\alpha]\varphi$ can be roughly put as "After every terminating execution of α, φ holds.". The other new kind of formulas is the diamond formula $\langle\alpha\rangle\varphi$, which is the dual counter part of a box formula.

The meaning of $\langle\alpha\rangle\varphi$ can be described as "After some terminating execution of α, φ holds".

A VSE specification or something that can be translated to VSE (currently only CASL) can be sent to the VSE prover via the node menu of development graph nodes in two different ways. You can either select VSE from the theorem prover choice box shown after "Prove" or you can select "Prove VSE Structured". The first choice will call VSE with a single flattened theory whereas a structured call will translate all nodes with ingoing links to the current one individually.

VSE pops up with a "project" window. In this window you can choose "Work on" and "specification". Besides the builtin specification "boolean" there is at least one specification from your development graph that you can select for proving. For a structured choice you'll have specifications for all underlying nodes that you should work on in a bottom up fashion.

The state created by VSE will be stored in a `.tar` file (within the current directory) that preserves proofs for replay later on as long as you don't change library or node names.

11.5 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 11.2). 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.

11.6 Reduce

This is a connection to the computer algebra system from `http://www.reduce-algebra.com/`. Installation is possible as follows:

```
svn co https://reduce-algebra.svn.sourceforge.net/svnroot/reduce-algebra
cd reduce-algebra/trunk
./configure --with-csl
make
```

The binary `redcsl` will be searched in the `PATH` or is taken from the `HETS_REDUCE` environment variable.

11.7 Pellet

Pellet is a popular open-source DL-reasoner for $\mathcal{SROIQ}(\mathbf{D})$, which is the logic underlying OWL 2, written in Java. A Java Runtime Environment (in version > 1.5) is needed to run Pellet. For the integration into Hets the environment variable `PELLET_PATH` has to be set to the root-directory of the Pellet installation.

Pellet uses the same ATP GUI as the provers for SoftFOL (ref. to section 11.2).

11.8 Fact++

Fact++ is a DL-reasoner for $\mathcal{SROIQ}(\mathbf{D})$, which is the logic underlying OWL 2, written in C++. Fact++ is integrated into Hets via the OWL-API, which is written in Java. A Java Runtime Environment (in version >= 1.6) has to be installed. To use Fact++, the environment variable `HETS_OWL_TOOLS` has to be set to the directory containing the files

```
OWL2Parser.jar
OWLFact.jar
OWLFactProver.jar
OWLLocality.jar
lib/guava-18.0.jar
lib/owlapi-osgidistribution-3.5.2.jar
lib/trove4j-3.0.3.jar
lib/uk.ac.manchester.cs.owl.factplusplus-P5.0-v1.6.3.1.jar
```

as well as

```
lib/native/i686/libFaCTPlusPlusJNI.so
```

on a 32bits-Linux-system or

```
lib/native/x86_64/libFaCTPlusPlusJNI.so
```

in a 64bits-Linux-system. Fact++ does not support options.

Fact++ uses the same ATP GUI as the provers for SoftFOL (ref. to section 11.2).

11.9 E-KRHyper

E-KRHyper[3] is an extension of KRHyper[4] by handling of equality. E-KRHyper is an automatic first order theorem prover and model finder based on the Hyper Tableaux Calculus[6]. E-KRHyper is optimized for being integrated into other systems. In the current implementation we use a default tactics script, that can be influenced by the user. The options of E-KRHyper are written in a Prolog-like syntax as in

[3] `http://www.uni-koblenz.de/~bpelzer/ekrhyper/`
[4] `http://www.uni-koblenz.de/~wernhard/krhyper/`

```
#(set_parameter(timeout_termination_method,0)).
```

the "." at the end of each option is mandatory. To get an overview of E-KRHyper's options, run the command

```
ekrh
```

in a terminal. Then enter the command

```
#(help).
```

at the prompt of E-KRHyper, to display its help information, which is basically a long list of all available parameters. You can exit E-KRHyper by the command

```
#(exit).
```

E-KRHyper uses the same ATP GUI as the other provers for SoftFOL (ref. to section 11.2).

11.10 Darwin

Darwin is an automatic first order prover and model finder implementing the Model Evolution Calculus[7]. The integration of Darwin as a consistency checker supports the display of models (if they can be constructed) in CASL-syntax. Eprover is needed to be in the system-path, if Darwin is used with HETS, since Darwin uses Eprover for clausification of first-order formulae.

Darwin supports a wide range of options, to get an overview of them run the command

```
darwin --help
```

in a terminal.

Darwin uses the same ATP GUI as the other provers for SoftFOL (ref. to section 11.2).

11.11 QuickCheck

11.12 minisat

11.13 Truth tables

11.14 CspCASLProver

12 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.

```
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 10: The basic ingredients of logics and logic comorphisms

- 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.

13 Architecture of Hets

The architecture of Hets is shown in Fig. 11. How is a single logic implemented in the Heterogeneous Tool Set? This is depicted in the left column of Fig. 11.

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 [42] by a set of types and functions, see Fig. 10, 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 [43]. The Haskell interface for logic translations is realised similarly.

The logic-independent modules in Hets can be found in the right half of Fig. 11. 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. 2 to an abstract syntax tree, using the `Parsec` combinator parser [23]. 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.

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. 11.

Hets can store development graphs, including their proofs. Therefore, Hets uses the so-called ATerm format [13], which is used as interchange format for interfacing with other tools.

More details can be found in [30, 36] 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 your feedback into account 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.

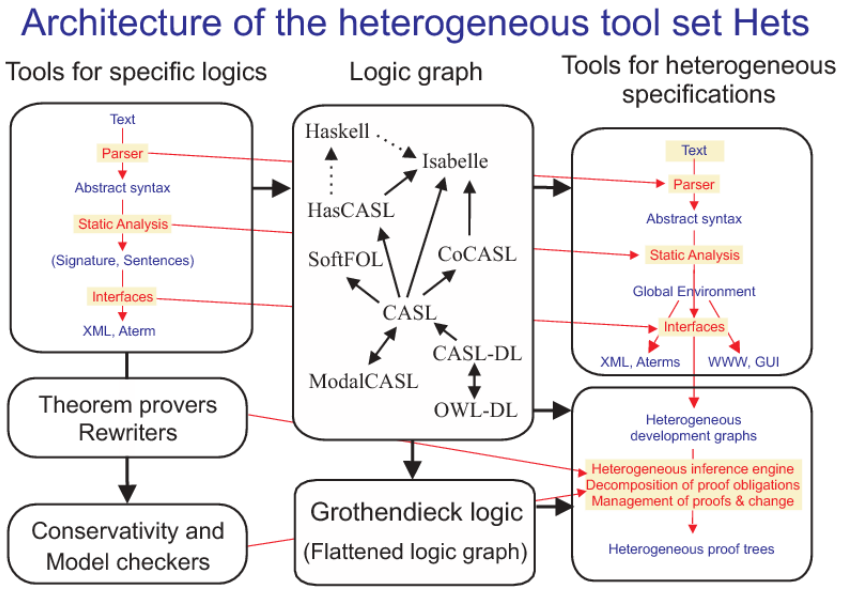

Figure 11: Architecture of the heterogeneous tool set.

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, Francisc Nicolae Bungiu, Michael Chan, Codruţa Gârlea, Dominik Dietrich, Elena Digor, Carsten Fischer, Jorina Freya Gerken, Andy Gimblett, Rainer Grabbe, Sonja Gröning, Markus Gro, Klaus Hartke, Daniel Hausmann, Wiebke Herding, Hendrik Iben, Cui "Ken" Jian, Heng Jiang, Stef Joosten, Anton Kirilov, Tina Krausser, Martin Kühl, Eugen Kuksa, Mingyi Liu, Karl Luc, Klaus Lüttich, Maciek Makowski, Felix Gabriel Mance, Florian Mossakowski, Immanuel Normann, Sebastian Raible, Liam O'Reilly, Razvan Pascanu, Daniel Pratsch, Corneliu-Claudiu Prodescu, Felix Reckers, Adrián Riesco, Markus Roggenbach, Pascal Schmidt, Ewaryst Schulz, Kristina Sojakova, Igor Stassiy, Tilman Thiry, Paolo Torrini, Jonathan von Schroeder, Simon Ulbricht, René Wagner, Jian Chun Wang, Zicheng Wang, and Thiemo Wiedemeyer.

Hets has been built based on experiences with its precursors, Cats and Maya. The Casl Tool Set (Cats) [32, 34] 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 [5, 4] 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, Bruno Langenstein, 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, B. Langenstein, H. Mantel, G. Rock, A. Schairer, W. Stephan, R. Vogt, and A. Wolpers. VSE: Formal methods meet industrial needs. *International Journal on Software Tools for Technology Transfer, Sp ecial issue on Mechanized Theorem Proving for Technology*, 3(1), september 2000.

[4] 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.

[5] 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.

[6] P. Baumgartner, U. Furbach, and I. Niemelä. *Hyper tableaux*, pages 1–17. Lecture Notes in Comput. Sci. Springer, 1996.

[7] P. Baumgartner and C. Tinelli. The Model Evolution Calculus. In F. Baader, editor, *CADE-19 – The 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 350–364. Springer, 2003.

[8] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Implementing the Model Evolution Calculus. In Stephan Schulz, Geoff Sutcliffe, and Tanel Tammet, editors, *Special Issue of the International Journal of Artificial*

Intelligence Tools (IJAIT), volume 15 of *International Journal of Artificial Intelligence Tools*, 2005. Preprint.

[9] Christoph Benzmüller, Florian Rabe, and Geoff Sutcliffe. THF0 – the core of the TPTP language for higher-order logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 491–506. Springer, 2008.

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

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

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

[13] 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.

[14] Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski, Florian Rabe, and Kristina Sojakova. Towards logical frameworks in the Heterogeneous Tool Set Hets. In *Recent Trends in Algebraic Development Techniques, 20th International Workshop, WADT 2010*, Lecture Notes in Computer Science. Springer, 2011.

[15] Dominik Dietrich, Lutz Schröder, and Ewaryst Schulz. Formalizing and operationalizing industrial standards. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Fundamental Approaches to Software Engineering*, volume 6603 of *Lecture Notes in Computer Science*, pages 81–95. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-19811-3_7.

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

[17] Codruţa Gîrlea. An extended modal logic institution. Master's thesis, Department of Computer Science, University of Bremen, 2010.

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

[19] 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.

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

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

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

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

[24] 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.

[25] Klaus Lüttich and Till Mossakowski. Reasoning Support for CASL with Automated Theorem Proving Systems. In J. Fiadeiro, editor, *WADT 2006*, number 4409 in LNCS, pages 74–91. Springer, 2007.

[26] Manuel A. Martins, Alexandre Madeira, Razvan Diaconescu, and Luís Soares Barbosa. Hybridization of institutions. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *CALCO*, volume 6859 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2011.

[27] M.Kohlhase, T.Mossakowski, and F.Rabe. The latin project. see `https://trac.omdoc.org/LATIN/`., 2009.

[28] T. Mossakowski. HetCASL – heterogeneous specification. Language summary, 2004.

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

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

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

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

[37] Till Mossakowski, Lutz Schröder, Markus Roggenbach, and Horst Reichel. Algebraic-co-algebraic specification in CoCASL. *Journal of Logic and Algebraic Programming*, 2004. To appear.

[38] 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`.

[39] Renato Neves, Alexandre Madeira, Manuel A. Martins, and Luís Soares Barbosa. Hybridisation at work. In Reiko Heckel and Stefan Milius, editors, *CALCO*, volume 8089 of *Lecture Notes in Computer Science*, pages 340–345. Springer, 2013.

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

[41] Björn Pelzer and Christoph Wernhard. System description: E-krhyper. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 508–513. Springer, 2007.

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

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

[44] F. Rabe. First-Order Logic with Dependent Types. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2006.

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

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

[47] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. EATCS Monographs on theoretical computer science. Springer, 2012.

[48] W. Marco Schorlemmer and Yannis Kalfoglou. Institutionalising ontology-based semantic integration. *Applied Ontology*, 3(3):131–150, 2008.

[49] 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.

[50] 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.

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

[52] 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.

[53] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[54] Geoff Sutcliffe. The TPTP world – infrastructure for automated reasoning. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2010.

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

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

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

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

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

[60] 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.

[61] 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.