HETS for Common Logic Users
– Version 0.99 –

Till Mossakowski, Christian Maeder, Mihai Codescu, Eugen Kuksa, Christoph Lange

DFKI GmbH, Bremen, Germany.

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

January 25, 2012

Contents

1 Introduction **1**

2 The Heterogeneous Tool Set and Its Input Languages **2**

3 Logics supported by Hets **3**

4 Logic translations supported by Hets **6**

5 Getting started **6**

6 Analysis of Specifications **7**

7 Heterogeneous Specification **8**

8 Development Graphs **9**

9 Proofs with HETS **18**

10 Reading, Writing and Formatting **23**

11 Hets as a web server **25**

12 Miscellaneous Options **26**

1 Introduction

Common Logic (CL) is an ISO standard published as "ISO/IEC 24707:2007 — Information technology — Common Logic (CL): a framework for a family of logic-based languages" [9]. CL is based on untyped first-order logic, but extends first-order logic in two ways: (1) any term can be used as function or predicate, and (2) sequence markers allow for talking about sequences of individuals directly.[1]

The Heterogeneous Tool Set (HETS) is an open source software providing several kinds of tool support for Common Logic:

[1] Strictly speaking, only the second feature goes beyond first-order logic.

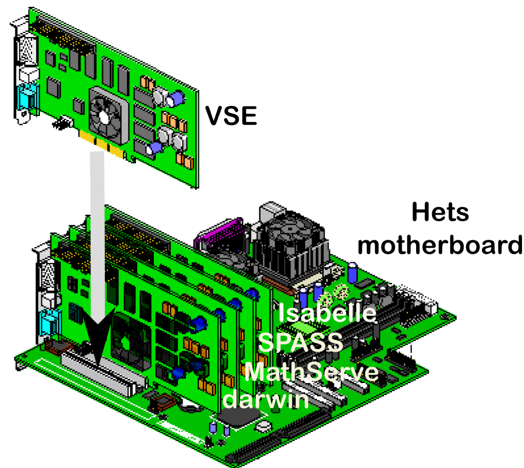

Figure 1: The HETS motherboard and some expansion cards

- a parser for the Common Logic Interchange Format (CLIF) — CLIF is a Lisp-like syntax for CL;
- a connection of CL to well-known first-order theorem provers like SPASS, darwin and Vampire, such that logical consequences of CL theories can be proved;
- a connection of CL to the higher-order provers Isabelle/HOL and Leo-II in order to perform induction proofs in theories involving sequence markers;
- a connection to first-order model finders like darwin that allow one to find models for CL theories;
- support for proving interpretations between CL theories to be correct;
- a translation that eliminates the use of CL modules[2]. Since the semantics of CL modules is special to CL, this elimination of modules is necessary before sending CL theories to a standard first-order prover;
- a translation of the Web Ontology Language OWL to CL;
- a translation of propositional logic to CL.

This guide will introduce you to these functionalities of HETS. For the full functionalities of HETS, see the HETS user guide [37].

2 The Heterogeneous Tool Set and Its Input Languages

The central idea of the Heterogeneous Tool Set (HETS) is to provide a 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. 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 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

[2]Actually, we are using a revised semantics for modules, as proposed recently in [27].

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 formalise the notion of a logic. The theory behind HETS is laid out in [19]. A short overview of HETS is given in [21, 22].

HETS supports a number of input languages directly, such as Common Logic and OWL2 and HETCASL. They will be described in the next sections.

2.1 Common Logic and the Common Logic Interchange Format (CLIF)

CLIF is specified in Annex A of the Common Logic standard [9]. HETS can directly read in files in CLIF syntax, and also recursively reads in any imported files (cf. Sect. 8.1 for the syntax).

Common Logic itself does not support the specification of logical consequences, nor relative theory interpretations, nor other features that speak about structuring and comparing logical theories. Michael Grüninger has suggested certain special annotations comments for this purpose, which are supported by HETS, see Sect. 8.1. Alternatively, CLIF syntax can be used for specifications within HETCASL files, or CLIF files can be referred to within HETCASL files. HETCASL is a structuring language supporting relative theory interpretations and other things, see Sect. 2.3 below.

2.2 OWL2

OWL2 is a W3C standard [3]. HETS can directly read in OWL2 files in all syntaxes (called "serialisations") that the OWL API supports [2], including the native OWL XML syntax [25], the human-readable Manchester syntax [15], as well as RDF [29]. The RDF data model has multiple possible syntaxes itself, including RDF/XML [4] and the text-oriented Turtle syntax [5].

Since OWL2 does not support relative theory interpretations and other structuring features, such things can only be expressed in HETCASL files. For this purpose, OWL2 Manchester syntax can be used within HETCASL files, or OWL2 files can be referred to within HETCASL files.

2.3 HetCASL

For heterogeneous specification, HETS offers the Heterogeneous language HETCASL. HETCASL is not so much a logic, but a meta language that can express relations of theories written in different logics, like logical consequences, relative interpretations of theories, conservative extensions, translations of theories along logic translations, etc.

HETCASL generalises the structuring constructs of CASL (Common Algebraic Specification Language [8, 23]) to arbitrary logics (if they are formalised as institutions and plugged into the HETS motherboard), as well as to heterogeneous combinations of specifications written in different logics. See Fig. 1 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. 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. *Refinements* express the fact the a specification is becoming more specific. All this is supported by HETCASL. For details, see [18, 19, 23].

3 Logics supported by Hets

HETS supports a variety of different logics. The following are most important for use with Common Logic:

Common Logic is an ISO standard published as "ISO/IEC 24707:2007 - Information technology — Common Logic (CL): a framework for a family of logic-based languages" [9]. It is based on first-order

Listing 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.

```
SPEC ::= BASIC-SPEC        %% logic-specific syntax, e.g. CLIF or Manchester syntax
       | SPEC then SPEC          %% extension of a spec with new symbols and axioms
       | SPEC then %implies SPEC %% annotation: extension is logically implied
       | SPEC with SYMBOL-MAP    %% renaming of SPEC along SYMBOL-MAP
       | SPEC with logic ID      %% translation of SPEC to a different logic

DEFINITION ::= logic ID            %% select a new logic for subsequent items
             | spec ID = SPEC end  %% give the name ID to SPEC
             | view ID : SPEC to SPEC = SYMBOL-MAP end
               %% interpretation of theories
             | view ID : SPEC to SPEC = logic ID end
               %% dto., but across different logics

LIBRARY = DEFINITION*
```

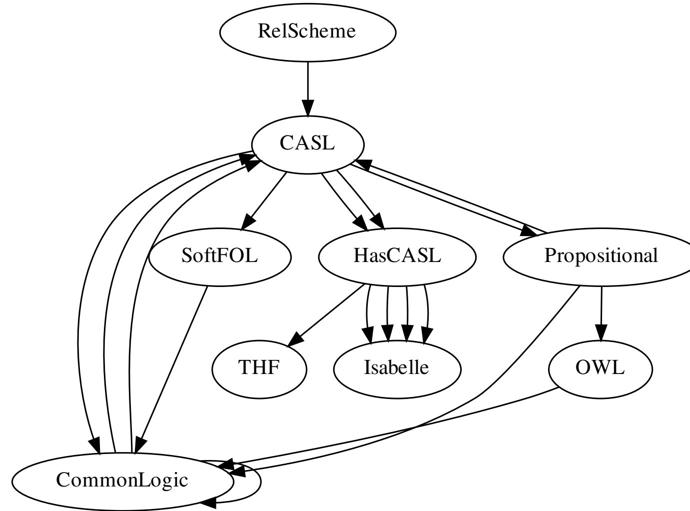

Figure 2: Graph of logics related to Common Logic that are currently supported by HETS. The more an ellipse is filled with green, the more stable is the implementation of the logic. Blue indicates a prover-supported logic.

Language	Parser	Static Analysis	Prover
CASL	x	x	–
Common Logic	x (CLIF)	x	–
OWL2	x	x	x
Propositional	x	x	x
SoftFOL	x	–	x

Figure 3: Current degree of HETS support for some of the languages. Languages without prover can still "borrow" provers via logic translations.

logic, but extends first-order logic in several ways. The Common Logic Interchange Format (CLIF) provides a Lisp-like syntax for Common Logic. HETS currently only supports parsing CLIF. If you need other dialects, send us a message and we will add them.

OWL2 is the Web Ontology Language recommended by the World Wide Web Consortium (W3C, `http://www.w3c.org`); see [3]. It is used for knowledge representation on the Semantic Web [7]. Hets calls an external OWL2 parser written in Java to obtain the abstract syntax for an OWL file and its imports. The Java parser also does a first analysis classifying the OWL ontology into the sublanguages OWL Full (all of OWL, under the RDF semantics, undecidable [32]), OWL DL (all of OWL, under the direct semantics [26]), and the so-called OWL Profiles (i.e. proper sublanguages) OWL EL, OWL QL, and OWL RL [24]. Hets supports all except OWL Full. The structuring of the OWL imports is displayed as a Development Graph.

Propositional is classical propositional logic, with the zChaff SAT solver [14] connected to it.

SoftFOL [17] offers several automated theorem proving (ATP) systems for first-order logic with equality: (1) SPASS [39], see `http://www.spass-prover.org`; (2) Vampire [31] see `http://www.vprover.org`; (3) Eprover [34], see `http://www.eprover.org`; (4) E-KRHyper [30], see `http://www.uni-koblenz.de/~bpelzer/ekrhyper`, and (5) MathServe Broker[3] [40]. These together comprise some of the most advanced theorem provers for first-order logic. SoftFOL is essentially the first-order interchange language TPTP [35], see `http://www.tptp.org`.

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 [23, 8]. For Common Logic, CASL can be seen as kind of transitional hub, linking Common Logic to other logics, most importantly SoftFOL.

ISABELLE [28] is the logic of the interactive theorem prover Isabelle for higher-order logic.

THF is an interchange language for higher-order logic [6], similar to what TPTP is for first-order logic. HETS connects THF to the automated higher-order prover Leo-II.

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. For Common Logic, HASCASL is mainly interesting as a transitional hub for paths to the provers Isabelle and Leo-II.

RelScheme is a logic for relational databases [33].

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. For Common Logic, the paths to SoftFOL are particularly interesting, because this offers an interface to standard first-order provers. Moreover, the paths to THF and Isabelle offer interfaces to higher-order provers, which is essential if you want to prove inductive theorems about sequences.

An introduction to CASL can be found in the CASL User Manual [8]; the detailed language reference is given in the CASL Reference Manual [23]. 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. 7 below.

For further information on logics supported by HETS, see the HETS user guide [37].

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

4 Logic translations supported by Hets

Logic translations (formalised 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.

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

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.
OWL22CommonLogic	Inclusion of OWL2 description logic
Prop2CommonLogic	Inclusion of propositional logic
SoftFOL2CommonLogic	Inclusion of first order logic
CASL2SoftFOL	Coding of CASL.SuleCFOL=E to SoftFOL [17], 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
CASL2Propositional	Translation of propositional FOL

Those comorphisms can be chained, e.g., for theorem proving, you can translate Common Logic to SoftFOL with `CommonLogic2CASLCompact;CASL2SoftFOLInduction` since there is no prover for Common Logic or CASL.

For further information on logic translations supported by HETS, see the HETS user guide [37].

5 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 is currently available (on Intel architectures only) for Linux and Mac OS X.

There are several possibilities to install HETS.

1. The best support is currently given via Ubuntu packages. For Ubuntu Lucid Lynx, enter the following into a terminal:

   ```
   sudo apt-add-repository ppa:hets/hets
   sudo apt-add-repository \
     "deb http://archive.canonical.com/ubuntu lucid partner"
   sudo apt-get update
   sudo apt-get install hets
   ```

   For later Ubuntu versions, replace lucid by maverick, natty or oneiric.

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

2. For Mac OS X 10.6 (Snow Leopard) we provide a meta package `Hets.mpkg` based on MacPorts that will be extended by further tools for proving in the future.

3. Then we have Java based HETS installer that we may drop in the future. Download a `.jar` file and start it with

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

   Note that you need Sun/Oracle Java 1.4.2 or later. On a Mac, you can just double-click on the `.jar` file, but you have to install the MacPorts `libglade2` package (and all its dependencies) yourself. In order to speed this up we provide a meta package `libglade2.mpkg`, too.

   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):

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 must be installed before)
SPASS	theorem prover	`http://spass.mpi-sb.mpg.de/`
Darwin	theorem prover	should be installed manually from `http://combination.cs.uiowa.edu/Darwin/`
ISABELLE	theorem prover	`http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`
(X)Emacs	editor (for Isabelle)	(must be installed manually)

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

5. 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:

zChaff	SAT solver	`http://www.princeton.edu/~chaff/zchaff.html`
minisat	SAT solver	`http://minisat.se/`
Pellet	OWL 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/`

6 Analysis of Specifications

Consider the following Common Logic text written in CLIF:

```
(P x)
(and (P x) (Q y))
(or (Cat x) (Mat y))
(not (On x y))
(if (P x) (Q x))
(exists (z) (and (Pet x) (Happy z) (Attr x z)))
```

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

Let us assume that the example is in a file named `Cat-AllInOne.clif`.[4] Then you can check the well-formedness of the specification by typing (into some shell):

```
hets Cat-AllInOne.clif
```

HETS checks both the correctness of this specification with respect to the CLIF syntax, as well as its correctness with respect to the static semantics. The following flags are available in this context:

-p, --just-parse Just do the parsing – the static analysis is skipped and no development graph 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. 8) 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).

There are more flags which can be used with HETS, see [37].

7 Heterogeneous Specification

HETS accepts plain text input files (for the presented logics) with the following endings:

filename extension	default logic	structuring language
`.casl`	CASL	CASL
`.het`	CASL	CASL
`.owl`	OWL2	OWL2
`.clf` or `.clif`	CommonLogic	custom, see Sect. 8.1

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 Common Logic in the following way:

```
logic CommonLogic
```

The subsequent specifications are then parsed and analysed as Common Logic 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 Common Logic (cf. Fig. 2). (There are also heterogeneous constructs for explicit translations between logics, see [18].)

The endings `.clf` and `.clif` are available for directly reading in Common Logic CLIF texts, as in the example of `Cat-AllInOne.clif`. By contrast, in HETCASL libraries (ending with `.het`), the logic Common Logic has to be chosen explicitly, and the CASL structuring syntax needs to be used:

```
library Cat

logic CommonLogic

spec Pred =
. (P x)
  (and (P x) (Q y))
```

[4]This file can be found in the `CommonLogic/Examples` directory in the HETS library [1].

```
spec Cat =
. (or (Cat x) (Mat y))
  (not (On x y))
  (if (P x) (Q x))

spec PetHappy =
Pred and Cat then
. (exists (z) (and (Pet x) (Happy z) (Attr x z)))
end
```

Note that the dot at the beginning of a line indicates that a new text begins. Hence, it is possible to have multiple texts in a CASL specification.

This specification is the HETCASL-structured equivalent to the following three CLIF files:[5]

Pred.clif:

```
(cl:text Pred
  (P x)
  (and (P x) (Q y))
)
```

Cat.clif:

```
(cl:text Cat
  (or (Cat x) (Mat y))
  (not (On x y))
  (if (P x) (Q x))
)
```

Spec.clif:

```
(cl:text PetHappy
  (cl:imports Pred) (cl:imports Cat)
  (exists (z) (and (Pet x) (Happy z) (Attr x z)))
)
```

Both can be directly used with HETS, where the former content would be in a file with the extension `.het` and the latter in a file with one of the extensions `.clf` or `.clif`.[6] This specification is divided into three parts, which are linked to each other. These links and some more information can be seen in the development graph of the file.

8 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 edges called *definition links*, indicating the dependency of each involved

[5]Note that where the Common Logic specification requires “cl:text”, some samples available on the Web use “cl-text”. Therefore, HETS also supports the latter.

[6]These files can be found in the `CommonLogic/Examples` directory in the HETS library [1].

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.

Theorem links are initially displayed in red. The *proof calculus* for development graphs [20, 19] 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 by HETS. 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 [19] stating soundness and relative completeness of the proof calculus for heterogeneous development graphs.

Details can be found in the CASL Reference Manual [23, IV:4] and in [19, 20, 22].

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

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

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

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

For a summary of the types of nodes and links occurring in development graphs, see the HETS user guide [37].

Most of the pull-down menus of the development graph window are uDraw(Graph)-specific layout menus; their function can be looked up in the uDraw(Graph) documentation[7]. The Edit menu is the only exception. With choosing Edit→Proofs→Auto-DG prover, you can can prove red theorem links (which may be generated by relative interpretations of theories). Actually, this will generate new proof obligations at some node, which then can be discharged there. Moreover, the nodes and links of the graph have attached pop-up menus, which appear when clicking (and holding) the right mouse button. The node menus "Prove" and "Check consistency" are the most important ones. With "Add sentence", you can add axioms and proof goals on the fly. For a detailed explanation of the menus see the HETS User Guide [37].

8.1 Relations between Common Logic Texts

HETS supports several relations between Common Logic Texts. However only one of them is defined in ISO/IEC 24707:2007 [9]. All the other relations are unofficial extensions used e.g. by the Common Logic Repository COLORE [12].

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

Importation is defined in ISO/IEC 24707:2007 [9] as virtual copying of a resource. In HETS a whole file is "copied" into the importing specification. Hets cannot currently handle cyclic imports. If you really need them, send us a message at hets@informatik.uni-bremen.de, and we will fix it.

Using CLIF, you can import `someFile.clif` via

```
(cl:imports someFile.clif)
```

Omitting the file extension will also succeed. In this case HETS will look for a file called `someFile.clif` in first place and then for `someFile.clf` in the current directory and then in the library paths.

HETS also supports URIs for importing resources. The allowed URI schemes are `file:`, `http:` and `https:`.

```
(cl:imports file:///absolute/path/to/someFile.clif)
(cl:imports http://someDomain.com/path/to/someFile.clif)
(cl:imports https://someDomain.com/path/to/someFile.clif)
```

The importation is indicated by a global definition link (black arrow) in the development graph.

Relative interpretation is described in [13]. It is represented by a theorem link (red arrow) in the development graph. In a Common Logic file it is specified inside of a comment on text-level, that is a comment in the uppermost level of the (optionally named) Common Logic text instead of a comment in a sentence or term:

```
(cl:text someText
  (cl:comment '(relatively-interprets someTranslationFile someTargetFile)')
  (someAxiom)
)
```

Just as with imports (8.1), HETS supports different types of references to resources here, such as URIs. Alternatively, the HETCASL syntax for relative interpretations is

view *v* : *sp1* **to** *sp2* **end**

Views are declared outside of specifications, as can be seen from the syntax specification in List. 1. We provide a concrete example in Sect. 8.2.2 below.

Nonconservative extension is represented by a theorem link (red arrow) in the development graph. In a Common Logic file it is specified inside of a comment on text-level:

```
(cl:comment '(nonconservative-extension someTargetFile)')
```

Just as with imports (8.1), HETS supports different types of references to resources here, as e.g. URIs.

Inclusion is not a relation between theories. However inclusion can useful. It is used to show other theories in the development graph, without any connection to the current theory. In a Common Logic file inclusion is specified in a text-level comment:

```
(cl:comment '(include-libs someFile someOtherFile nextFile andSoOn)')
```

The keyword `include-libs` is followed by a whitespace-separated list of resources to be shown in the development graph. The resource-names can be of different type here, too.

Except for importation and inclusion, you can specify an optional symbol map (name map) in a relation.[8] Names from the target file are mapped to names from the current file (including the translation file, if the relation uses one). Note that is is possible to use cyclic relations in HETS. Only the cyclic importation is not supported.

[8]While the "copy" semantics of Common Logic importations does not permit renamings, HETCASL's extension mechanism offers an alternative possibility to reuse ontologies and rename some of their symbols, using the "*importedSpec* **with** *name1Old* |-> *name1New*, *name2Old* |-> *name2New* **then** *importingSpec*" syntax.

8.2 Examples

This section introduces several typical examples of using Common Logic ontologies with HETS.

8.2.1 Renaming Symbols with Symbol Maps

This example has two almost identical files, `upper.clif` and `lower.clif`; these can be found in the `CommonLogic/Examples` directory in the HETS library [1]. The only difference in the actual axioms is that `upper.clif` uses uppercase predicates while `lower.clif` uses lowercase predicates. The symbol is added at the end of the relation definition in parentheses. Only those names that differ between source and target need to be listed. The other names are implicitly the same. A mapping of a single name is defined with "`nameInTargetFile |-> nameInCurrentFile`"; multiple mappings are separated by commas. Note that in Common Logic, a comma can be part of a name. Hence a space must be placed between the separation-comma and a name.

upper.clif:

```
(cl:text upper
  (cl:comment '(nonconservative-extension lower
    ( a |-> A
    , b |-> B
    ))'
  )

  (forall (x y) (iff (A x y)
                     (B x y)))
)
```

lower.clif:

```
(cl:text lower
  (cl:comment '(nonconservative-extension upper
    (A |-> a
   , B |-> b))')

  (forall (x y) (iff (a x y)
                     (b x y)))
)
```

8.2.2 Relative Interpretation in COLORE

We give two examples for relative interpretation: one from COLORE (in this section), and one standalone one (in Sect. 8.2.3).

The COLORE [12] module `RegionBooleanContactAlgebra` relatively interprets the module `Atomless-BooleanLattice`. These two modules specify axioms about booleans; thus, they have the same signature. In the graph of imports, they have several common imported modules (e.g. `BoundedDistributive-Lattice`), but no common importing module, as can be seen from Fig. 4.

For use with HETS, we have made a dump of the COLORE contents available in `CommonLogic/colore` in the HETS library [1].

We first show how to express the relative interpretation using COLORE's CLIF annotations, then how to express it using HETCASL syntax. The HETCASL variant can be found in the `CommonLogic/Examples` directory.

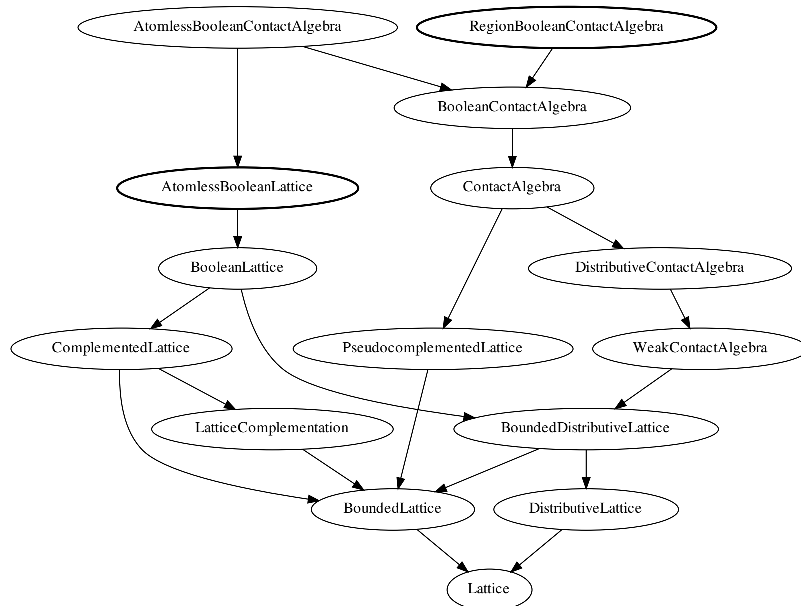

Figure 4: Boolean algebras and lattices in COLORE (subgraph)

Common Logic Syntax:

```
(cl:module AtomlessBooleanLattice
  (cl:imports BooleanLattice)
  (forall (x)
          (exists (y)
                  (and (not (= y 0))
                       (leq y x)))))

(cl:module RegionBooleanContactAlgebra
  (cl:imports BooleanContactAlgebra)
  (cl:comment '(relatively-interprets ??? AtomlessBooleanLattice)')
  (forall (x)
          (if (and (not (= x 0))
                   (not (= x 1)))
              (exists (y)
                      (and (complement x y)
                           (C x y))))))
```

HetCasl Syntax:

```
logic CommonLogic

spec AtomlessBooleanLattice =
  BooleanLattice
```

```
then
. (forall (x)
          (exists (y)
                  (and (not (= y 0))
                       (leq y x))))
end

spec RegionBooleanContactAlgebra =
  BooleanContactAlgebra
then
. (forall (x)
          (if (and (not (= x 0))
                   (not (= x 1)))
              (exists (y)
                      (and (complement x y)
                           (C x y)))))
end

view v : AtomlessBooleanLattice to RegionBooleanContactAlgebra
```

8.2.3 Relative Interpretation (Standalone Example)

This example defines a partial order twice: once as an extension of a strict partial order, and once directly. Then, we connect both definitions by a "relative interpretation" link. The sources are available from the `CommonLogic/Examples` directory in the HETS library [1].

Common Logic Syntax:

```
(cl:module Partial_Order
  (cl:comment 'reflexive')
  (forall (x)
          (le x x))
  (cl:comment 'asymmetric')
  (forall (x y)
          (if (and (le x y)
                   (le y x))
              (= x y)))
  (cl:comment 'transitive')
  (forall (x y z)
          (if (and (le x y)
                   (le y z))
              (le x z))))

(cl:module Partial_Order_From_Strict_Partial_Order
  (cl:imports Strict_Partial_Order.clif)
  (cl:comment 'define "less or equal" in terms of "less than"')
  (forall (x y)
          (iff (le x y)
               (or (lt x y)
                   (= x y)))))

(cl:module Partial_Order_From_Strict_Partial_Order
```

```
(cl:imports Strict_Partial_Order.clif)
(cl:comment '(relatively-interprets ??? Partial_Order)')
(cl:comment 'define "less or equal" in terms of "less than"')
(forall (x y)
        (iff (le x y)
             (or (lt x y)
                 (= x y)))))
```

HetCasl Syntax:

```
logic CommonLogic

spec Strict_Partial_Order =
%% strict
. (forall (x)
          (not (lt x x)))
%% asymmetric
. (forall (x y)
          (if (lt x y)
              (not (lt y x))))
%% transitive
. (forall (x y z)
          (if (and (lt x y)
                   (lt y z))
              (lt x z)))
end

spec Partial_Order_From_Strict_Partial_Order =
    Strict_Partial_Order
then
%% define "less or equal" in terms of "less than"'
. (forall (x y)
          (iff (le x y)
               (or (lt x y)
                   (= x y))))
end

spec Partial_Order =
%% reflexive
. (forall (x)
          (le x x))
%% antisymmetric
. (forall (x y)
          (if (and (le x y)
                   (le y x))
              (= x y)))
%% transitive
. (forall (x y z)
          (if (and (le x y)
                   (le y z))
              (le x z))))
end
```

```
view v : Partial_Order to Partial_Order_From_Strict_Partial_Order
```

8.2.4 Ontology-based Ambient Assisted Living Services and Devices

Consider the following ambient assisted living (AAL) scenario:

> Clara instructs her **wheelchair** to get her to the **kitchen** (**next door** to the **living room**. For **dinner**, she would like to take a pizza from the **freezer** and bake it in the **oven**. (Her diet is vegetarian.) **Afterwards** she needs to rest in **bed**.

Existing ontologies for ambient assisted living (e.g. the OpenAAL[9] OWL ontology) cover the *core* of these concepts; they provide at least classes (or generic superclasses) corresponding to the concepts highlighted in **bold**. However, that does not cover the scenario completely. In particular, there are relevant concepts (here: space and time, underlined), which are not covered at the required level of complexity. OpenAAL says that appointments have a date and that rooms can be connected to each other, but not what exactly that means. Foundational ontologies and spatial calculi, often formalized in first-order logic, cover space and time at the level of complexity required by a central controller of an apartment and by an autonomously navigating wheelchair.

More concretely, Common Logic is useful in this scenario for expressing knowledge on the arrangement of rooms, e.g. as follows in a first-order formalization of an RCC-style spatial calculus:

$$\forall a_1, a_2.\mathrm{equal}(a_1, a_2) \veebar \mathrm{overlapping}(a_1, a_2) \veebar \mathrm{bordering}(a_1, a_2) \veebar \mathrm{disconnected}(a_1, a_2) \veebar \\ \mathrm{proper_part_of}(a_1, a_2) \veebar \mathrm{proper_part_of}(a_2, a_1)$$

("Two areas in a house (e.g. a working area in a room) are either the same, or intersecting, or bordering, or separated, or one is a proper part of the other.")

The following listing shows a relevant excerpt of the heterogeneous specification, which can be found under `Ontology/Examples/AAL.het` in the HETS library [1]. The key features include:

- Heterogeneous specification allows for reusing the OpenAAL OWL ontology, but at the same time formalizing a first-order spatial calculus.
- In particular, the compact representation of mutual disjointness chosen here makes use of Common Logic's sequence markers.
- As Common Logic module extends the previously imported OWL ontology, it has access to all entities of the OWL ontology by name; in particular, we can specify that two rooms are connected (in terms of the OpenAAL terminology) if certain conditions in terms of our Common Logic module, or certain conditions in terms of OpenAAL hold.

```
library AAL

logic OWL

from OpenAALOntology get httpwwwdfkideskshetsontologyunnamed
%% this is the default name that Hets assigns to unnamed ontologies

spec OurAAL =
  %% Import the OpenAAL OWL ontology.
  httpwwwdfkideskshetsontologyunnamed
  ... %% some other extensions not shown here
  %% Extend it by an RCC-style spatial calculus
```

[9] http://openaal.org

```
  %% formalized in first order logic.
  then logic CommonLogic : {
    . (forall (a1 a2)
        (or (equal a1 a2)
            (overlapping a1 a2)
            (bordering a1 a2)
            (disconnected a1 a2)
            (proper_part_of a1 a2)
            (proper_part_of a2 a1)))
    %% mutual disjointness of predicates (need this for an exclusive or)
    . (forall (p)
        (mutually-disjoint p))
    . (forall (p q ...)
       (iff (mutually-disjoint p q ...)
          (and (forall (...x)
                  (not (and (p ...x) (q ...x))))
               (mutually-disjoint p ...)
               (mutually-disjoint q ...))))
    %% a utility predicate for talking about inverse relations
    %% (similar to owl:inverseOf)
    . (forall (r x y)
         (iff ((converse r) x y) (r y x)))
    %% make the above "or" exclusive
    . (mutually-disjoint equal overlapping bordering disconnected
                         proper_part_of (converse proper_part_of))
    %% if some RCC relations hold (so far it would also work in OWL)
    %% or if there is a door that connects two rooms, then ...
    %% (the latter would only work in OWL if we used an explicit subproperty
    %% is-door-of of is-in-room; then we could chain "inverse is-door-of"
    %% and "is-door-of", but otherwise we wouldn't be able to restrict the
    %% "connecting element" to a Door)
    . (forall (a1 a2)
         (if (or (equal a1 a2)
                 (overlapping a1 a2)
                 (proper_part_of a1 a2)
                 (proper_part_of a2 a1)
                 (exists (door)
                   (and (Door door)
                        (is-in-room door a1)
                        (is-in-room door a2)))
             )
             (is-connected-to-room a1 a2)))
```

8.2.5 Heterogeneous Views from OWL to Common Logic

In the previous example, we established a link between an OWL ontology and a Common Logic ontology by reusing elements of the signature of the OWL ontology (concretely: OpenAAL's `is-in-room` predicate) in the Common Logic ontology.

HETCASL's view mechanism offers an alternative to that. A view from one ontology to another ontology in the same logic has been shown in Sect. 8.2.2, but it is also possible to have views across logics, as long as there is a translation between these logics that is known to HETS (cf. Sect. 4).

The following example establishes a view between the OWL Time ontology and its reimplementation in Common Logic, using the "OWL22CommonLogic" translation:

```
logic OWL
spec TimeOWL =
  Class: TemporalEntity
```

```
  ObjectProperty: before
    Domain: TemporalEntity
    Range: TemporalEntity
    Characteristics: Transitive
end

logic CommonLogic
spec TimeCL =
%% CommonLogic equivalent of Domain and Range above
. (forall (t1 t2)
          (if (before t1 t2)
              (and (TemporalEntity t1)
                   (TemporalEntity t2))))
%% CommonLogic equivalent of Transitive above
. (forall (t1 t2 t3)
          (if (and (before t1 t2)
                   (before t2 t3))
              (before t1 t3)))
%% A new axiom that cannot be expressed in OWL
. (forall (t1 t2)
          (or (before t1 t2)
              (before t2 t1)
              (= t1 t2)))
end

view TimeOWLtoCL : TimeOWL to TimeCL
```

9 Proofs with Hets

The proof calculus for development graphs (Sect. 8) reduces global theorem links to local proof goals. You can do this reduction by clicking on the Edit-menu in the development graph window and selecting Proofs/Auto-DG-Prover. 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 their 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. The provers are described in more detail in the Hets user guide [37].
- 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

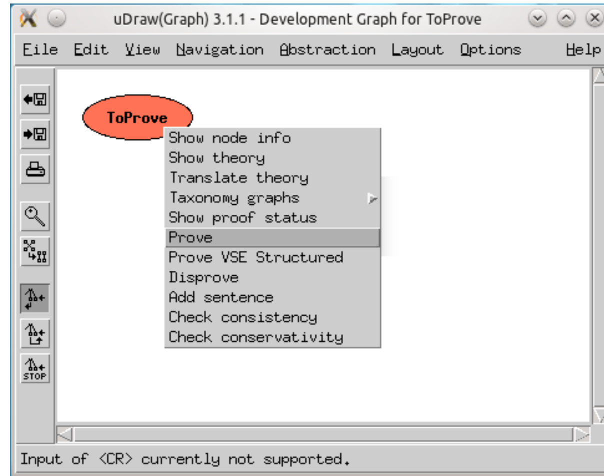

Figure 5: Prove local proof obligation

prover is launched and the theory along with the selected goals is translated via the shortest possible path of comorphisms into the prover's logic.

- The pop-up choice box below 'Selected comorphism path:' lets you pick a (composed) comorphism to be used for the chosen prover. If the specification does not contain any sequence markers, it is possible to use the comorphism `CommonLogic2CASLCompact` which results in a simpler CASL specification. We recommend using this comorphism whenever possible.

- 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 so do the available provers and the comorphism paths leading 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.

- 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.

In order to prove or disprove a theorem, it needs to be declared as proof obligation. This is done by the keyword `%implied` at the end of a text:

```
logic CommonLogic

spec ToProve =
. (P x)
  (and (P x) (Q y))
. (Q y) %implied %(correct)%
. (P y) %implied %(incorrect)%
end
```

In this specification[10] the theorems, annotated (named) by `correct` and `incorrect` are the ones, that can be proven or disproven. Note that they are separate texts inside the specification `ToProve`. The

[10] `HelloWorldExamples/ToProve.het` in the HETS library [1]

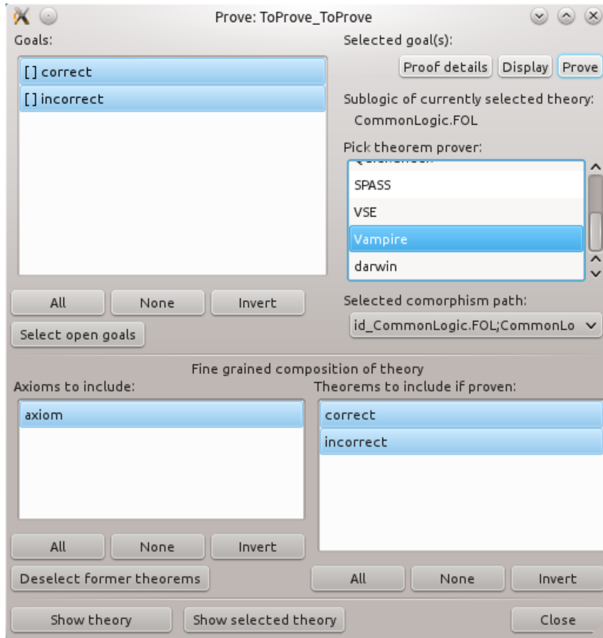

Figure 6: HETS Goal and Prover Interface

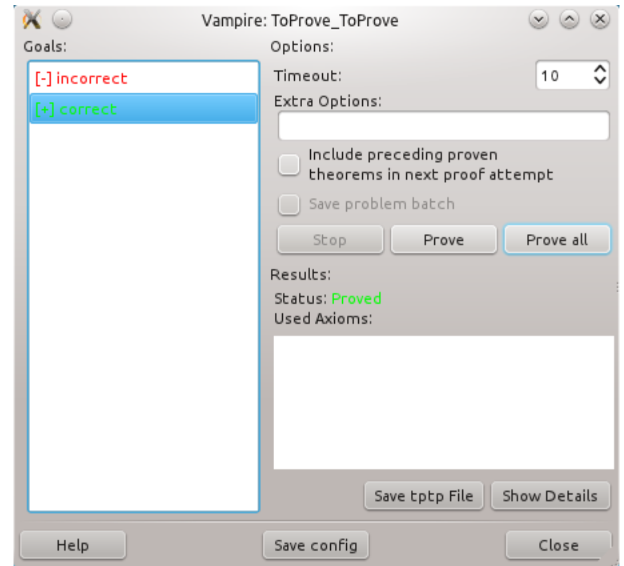

Figure 7: Interface of Vampire Prover

annotations are optional. For proving, they are the names shown in the "Axioms to include" section of the prover interface (Fig. 6).

The same specification can be written down in CLIF:

```
(cl:text axiom
  (P x)
  (and (P x) (Q y))
)

(cl:text correct
  (Q y)
) %implied

(cl:text incorrect (P y)) %implied
```

In CLIF, there is no notion of proof obligation. Hence the `%implied` keyword of HETS must be used, and thus the specification is not pure CLIF. Because the texts have names, these are also used in the prover interface. Otherwise, HETS invents names.

9.1 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. 9. 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

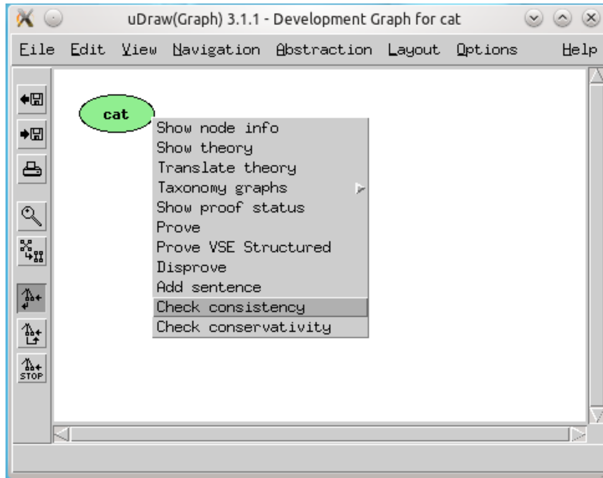

Figure 8: Selection of consistency checker

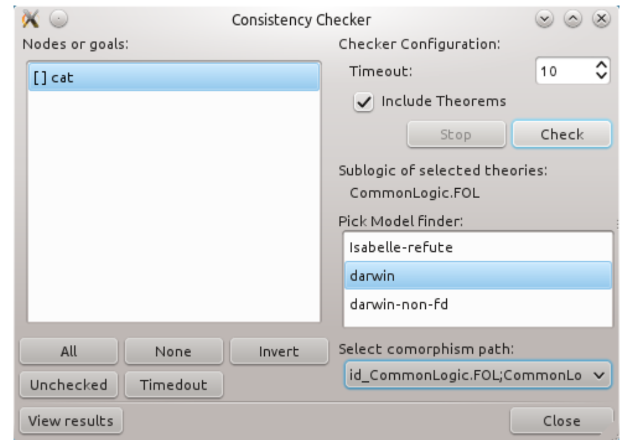

Figure 9: HETS Consistency Checker Interface

if too many nodes have been chosen. Either by 'View results' or automatically the 'Results of consistency check' (Fig. 10) will pop up and allow you to inspect the models for nodes, if they could be constructed.

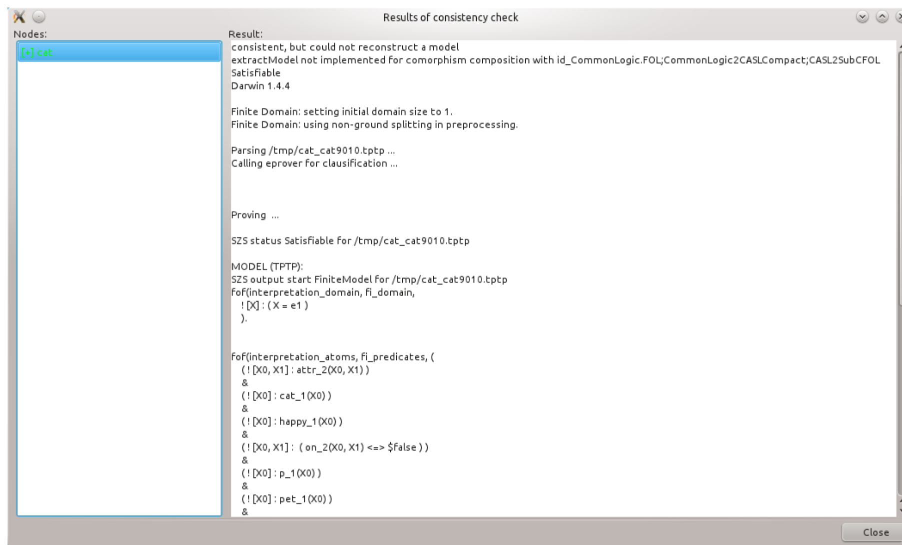

Figure 10: Consistency checker results

9.2 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. Fig. 11 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 preceding 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

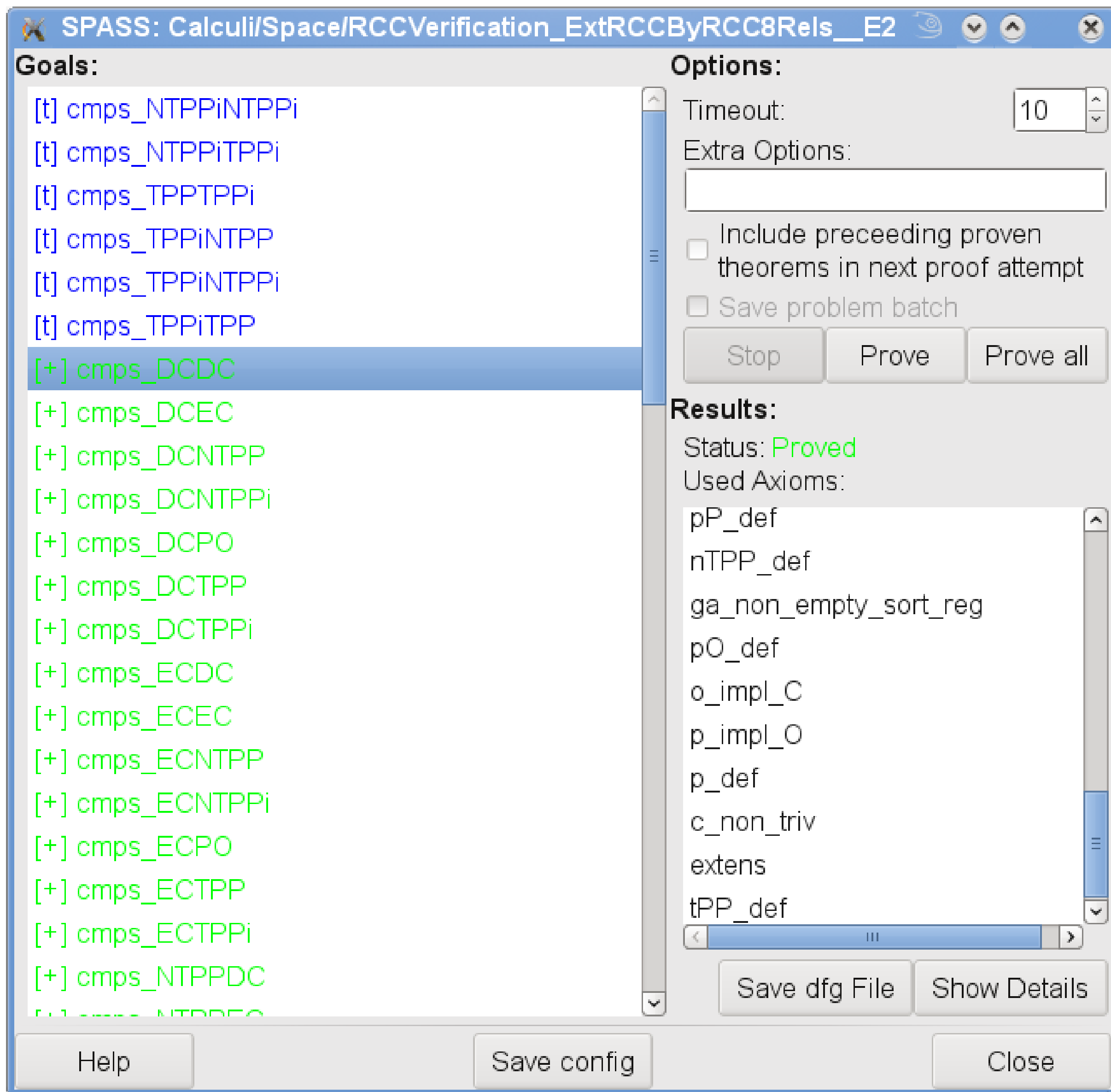

Figure 11: Interface of the SPASS prover

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

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 [40] developed by Jürgen Zimmer provides a unified interface to a range of different ATP systems; the most important systems are listed in Tab. 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 [36]. Only two 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 standardised 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.

For details on the ATPs supported, see the HETS user guide [37].

10 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.

`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
  | clif
  | 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 [15] will produce OWL files in Manchester Syntax for each specification of a structured OWL library.

The `clif` option will produce Common Logic files in CLIF dialect for each specification of a Common Logic library.

The `omdoc` format [16] 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. Although this is still in experimental state, Common Logic theories can be exported to and imported from OMDoc.

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 under `Calculi/Space/RCC8.het` in the HETS library [1]. 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 yet supported.)

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

```
hets -v2 -o pp.tex Cat.clif
```

This will generate a file `Cat.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 reserved for future usage.

The `dfg` format is used by the SPASS theorem prover [39].

The `tptp` format (`http://www.tptp.org`) is a standard exchange 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. 4)

-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 Sect. 11)

-x, --xml use XML-PGIP[11] packets to communicate with HETS in interactive mode

-S PORT, --listen=PORT communicate with HETS in interactive mode by listening 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)

11 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 at `http://trac.informatik.uni-bremen.de:8080/hets/wiki/RESTfulInterface`.

For details on this topic, see the HETS user guide [37].

[11] Proof General Interface Protocol

12 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 [38] against a CASL specification

References

[1] Hets library. `http://www.cofi.info/Libraries`.

[2] The OWL API. `http://owlapi.sourceforge.net`.

[3] OWL 2 web ontology language: Document overview. W3C recommendation, World Wide Web Consortium (W3C), October 2009.

[4] Dave Beckett. RDF/XML syntax specification (revised). W3C recommendation, World Wide Web Consortium (W3C), February 2004.

[5] David Beckett and Tim Berners-Lee. Turtle – terse RDF triple language. W3C team submission, World Wide Web Consortium (W3C), January 2008.

[6] 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.

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

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

[9] Information technology — Common Logic (CL): a framework for a family of logic-based languages. Technical Report 24707:2007, ISO/IEC, 2007. `http://iso-commonlogic.org`.

[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] M. Gruninger. Colore – Common Logic Repository. Available at `http://stl.mie.utoronto.ca/colore/`.

[13] Gruninger, Michael, Hahmann, Torsten, and Hashemi, Ali, and Ong, Darren. Ontology Verification with Repositories. In Antony Galton and Riichiro Mizoguchi, editors, *Formal Ontology in Information Systems, Proceedings of the Sixth International Conference (FOIS 2010)*, number 209 in Frontiers in Artificial Intelligence and Applications, pages 317–330. IOS Press, 2010.

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

[15] Matthew Horridge and Peter F. Patel-Schneider. OWL 2 web ontology language: Manchester syntax. W3C candidate recommendation, World Wide Web Consortium (W3C), 10 2009.

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

[17] 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.

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

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

[20] 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.

[21] 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.

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

[23] 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`.

[24] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 web ontology language: Profiles. W3C recommendation, World Wide Web Consortium (W3C), October 2009.

[25] Boris Motik, Bijan Parsia, and Peter F. Patel-Schneider. OWL 2 web ontology language: XML serialization. W3C recommendation, World Wide Web Consortium (W3C), 10 2009.

[26] Boris Motik, Peter F. Patel-Schneider, and Bernardo Cuenca Grau. OWL 2 web ontology language: Direct semantics. W3C recommendation, World Wide Web Consortium (W3C), October 2009.

[27] Fabian Neuhaus and Pat Hayes. Common logic and the horatio problem. accepted by Applied Ontology, 2011.

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

[29] Peter F. Patel-Schneider and Boris Motik. OWL 2 web ontology language: Mapping to RDF graphs. W3C recommendation, World Wide Web Consortium (W3C), 10 2009.

[30] Björn Pelzer and Christoph Wernhard. System description: E-krhyper. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 508–513. Springer, 2007.

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

[32] Michael Schneider. OWL 2 web ontology language: RDF-based semantics. W3C recommendation, World Wide Web Consortium (W3C), October 2009.

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

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

[35] Geoff Sutcliffe. The TPTP world – infrastructure for automated reasoning. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2010.

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

[37] Mihai Codescu Till Mossakowski, Christian Maeder. Hets user guide, 2011. `http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/UserGuide.pdf`.

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

[39] 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.

[40] 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.