H user manual[1]
– version 1.0.0 –

Mihai Codescu[1] and Răzvan Diaconescu[2]

[1]Institute of Mathematics "Simion Stoilow" of the Romanian Academy, Research Group of the project PED-0494, Bucharest, Romania
[2]Institute of Mathematics "Simion Stoilow" of the Romanian Academy, Bucharest, Romania

[1]This work was supported by a grant of the Romanian National Authority for Scientific Research and Innovation, CNCS/CCCDI – UEFISCDI, project number PN-III-P2-2.1-PED-2016-0494, within PNCDI III.

Chapter 1

Introduction

1.1 Reconfigurable systems in H

A reconfigurable system is one with different modes of operation, called configurations, and with the ability of to commute between these modes during its execution along transitions between these modes. Here, we present **H**, available at `http://imar.ro/~diacon/forver`, a tool for formal specification and verification of reconfigurable systems that supports their correct and efficient development. The key idea is to have a generic language for specifying the configurations and the transitions between them, while the formalism used for specifying the local behavior of the system, for each configuration, can be freely chosen by the user as the most appropriate for the given requirements.

Formally, this is achieved by applying a generic construction, called *hybridization*, that has as parameters the base logic, the kinds of symbols that are allowed to appear as variables in quantifications and constraints on the interpretations of various symbols in the configurations. The result is again a logic, that we will refer as a *hybridized logic*. Its signatures extend the signatures of the base logic with nominals (names of configurations) and modalities (names of events causing reconfigurations). The sentences can be sentences of the base logic, nominals, modal box- and diamond-sentences, retrieve sentences (meant to hold at a given state), combinations of sentences using Boolean connectors and, when the parameter allows, quantification on nominals and/or symbols from the base logic. Models are Kripke frames with each nominal interpreted as one of the possible worlds and the events as an accessibility relation between these worlds.

1.2 The Heterogeneous Tool Set

From an implementation point of view, **H** is an extension of the Heterogeneous Tool Set (Hets), a multi-formalism parsing, static analysis and proof management tool for the specification and verification of formal systems. The multi-

formalism aspect of Hets means that the system supports a variety of logics and languages and integrates their specific state-of-the-art proof tools. Moreover, Hets provides support for proof-by-translation technique: this enables the development of proof capabilities for new logical formalisms, with no dedicated or not sufficiently developed tool support, by means of translations to other tool-supported logics that are integrated into Hets. Hets has been implemented in such a way that a new logic can be integrated with moderate effort, and the implementation of **H** relies on this fact. Since **H** was implemented as an extension of Hets, the interfaces of the **H** and Hets tools are identical, and any component of Hets is also a part of **H**.

1.3 Structure

Chapter 2 provides instructions for downloading and installing the **H** environment. In Chapter 3 we illustrate how to write specifications in **H** and how to use the system to parse and analyze them. Chapter 4 explains how to verify properties of reconfigurable systems specified in **H**, using the first-order provers integrated in Hets. In Chapter 5 we discuss the notations used for the particular case of layered hybridization, when the logic used at the local level is a hybridized one. Chapter 6 shows how to write structured specifications in **H**. Finally, in Chapter 7 we present a more advanced feature of **H**, namely how to set the parameters for the hybridization process in order to obtain a new hybridized institution and make it available for specification and verification in the tool.

Acknowledgements. We thank Till Mossakowski and Fabian Neuhaus for discussions on language design and implementation issues and to Ionuţ Ţuţu for comments on several versions of this document.

Chapter 2

Getting the H environment

To be able to use **H**, choose one of the two binaries available at `http://imar.ro/~diacon/forver` according to the notation that you prefer: one for mathematicians, closer to textbooks on modal and hybrid logic, and one for engineers, closer to natural language. Note that both binaries are large files, about 90 Mb in size. Then install Hets:

```
sudo apt-get install software-properties-common
sudo apt-add-repository ppa:hets/hets
sudo apt-get update
sudo apt-get install hets-desktop
```

and replace the binary file (found at `\usr\bin\hets`) with the one downloaded from the H homepage. Fig. 2.1 below shows the interface for proving a conjecture in the H system.

To check that **H** has been installed properly, type in a terminal: `hets`. This will display information about the command flags available in Hets We will mention here only those Hets flags that directly relevant for **H**:

- `-vX` (where `X` is between 0 and 5) – sets the verbosity level. Use `-v2` to display warnings.
- `-g` – displays the development graph of the specifications being analyzed.

The general syntax for running **H** is

```
hets [OPTION] [FILE]
```

where `OPTION` is a command flag and `FILE` is the file to be analyzed. The file sent as an argument is checked for correctness w.r.t. syntax (i.e. do all declarations of every specification in the file respect the grammar of the language?) and static semantics (e.g. have all used symbols been declared before use?). If both checks are passed, Hets builds a *development graph*, consisting of a representation of the import structure between the specifications of the argument file and including the open conjectures. Informally, the nodes in development graphs correspond to theories. The edges can be of several kinds, divided in two categories: *import*

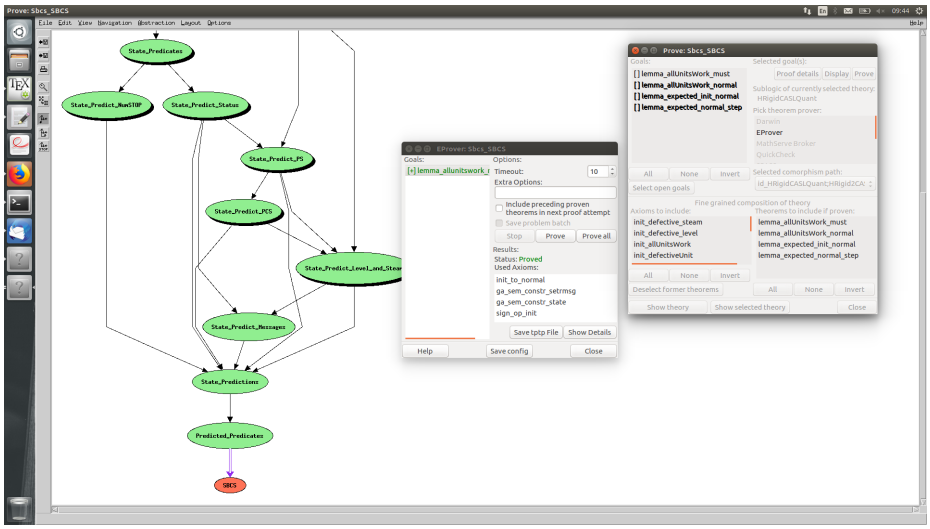

Figure 2.1: The H system

links, which can be within the same logic or along a logic translation, and *theorem links*, carrying the intended semantics that the sentences at the source of the link are logical consequences of those of the target of the link. Conjectures are marked in the graph by the red color of the nodes corresponding to the theories with open proof goals. We will see in Chapter 4 how to use the logic-specific provers integrated in Hets to discharge them.

The hybridized logics currently supported by **H** are:

HProp hybridization of propositional logic, no quantification, no constraints.

HPropNoms hybridization of propositional logic, admits quantification on nominals, no constraints.

HPropT like HProp, accessibility relation between worlds must be reflexive.

HPropS4 like HProp, accessibility relation between worlds must be a preorder.

HPropS5 like HProp, accessibility relation between worlds must be an equivalence.

HFOL hybridization of first-order logic, admits quantification on nominals.

HFOLR hybridization of first-order logic with rigid symbols, quantification on rigid constants and nominals.

HFOLRC like HFOLR, rigid symbols are interpreted in the same way in each world.

HRigidCASL hybridization of logic of partial algebras with rigid symbols, quantification on rigid total constants and nominals.

HRigidCASLC like HPAR, rigid sorts and rigid total functions are interpreted in the same way in each world, the domain of each rigid partial function is the same in each world.

In Chapter 5 we will add more logics to this list.

Chapter 3

Writing H specifications

Each **H** specification has 3 parts: the current logic, the data part and the configuration part.

3.1 Current logic

For each specification in a hybridized logic, we must specify its underlying logic. This determines which logic-specific parser and static analysis tool are called during the analysis of the specification. The syntax is

```
hlogic: LOGIC-NAME
```

where `LOGIC-NAME` is the name of a hybridized logic supported in **H**.

Example 3.1.1. We can set the current logic to be the hybridization of the logic of propositional logic by writing

```
spec Test =
    hlogic: HProp
    ...
```

3.2 Data part

The syntax is logic-specific. The examples we use involve hybridization of propositional logic, of first-order logic and its sublogics (no function symbols other than constants, no predicate symbols), of the logic of partial algebras and of logics with rigid symbols. In the following we will briefly give a quick reference for writing specifications in these logics: propositional logic has its own syntax, first-order and partial logics appear in Hets as sublogics of the CASL logic and inherit its syntax, and rigid symbols are written, in the appropriate logics, as usual declarations prefixed by the keyword `rigid`. For every logic, sentences start with a dot and they have optional names, written `%(name)%`. Comments start with `%%`. Specifications in a base logic `L` are written

```
spec S =
 logic L :
 {
  <logic-specific declarations>
 }
```

3.2.1 In propositional logic

Propositional symbols are declared using the keyword `props`. Sentences are formed through repeated applications of the Boolean connectors to the propositional symbols. We illustrate the syntax with some well-known tautologies:

```
props a, b

. not (a \/ b) <=> (not a /\ not b)
                          %(deMorgan)%

. (a => b) <=> (not b => not a)
                          %(contraposition)%

. ((not a => b) /\ (not a => not b)) => a
                          %(reductio_ad_absurdum)%
```

Note that negation binds strongest, followed in order by conjunction, disjunction, implication and equivalence.

3.2.2 In a sublogic of CASL

Declarations may include sorts, total or partial function symbols and predicate symbols. Sentences are formed by applying Boolean connectors and quantification over total variables to atomic formulas. These can be predications, strong or existential equalities or definedness assertions. For example, we can specify natural numbers in Peano style, with the constant 0, the successor function `suc`, the partial predecessor function `pre` and the smaller-than predicate `<`, in infix form, as below:

```
sort Nat
ops 0 : Nat;
    suc : Nat -> Nat;
    pre : Nat ->? Nat
pred __<__ : Nat * Nat

. forall x : Nat . not (suc(x) = 0)
. forall x, y : Nat . suc(x) = suc(y) => x = y
. not def(pre(0))
. forall x : Nat . pre(suc(x)) = x
. not (0 < 0)
```

```
. forall x : Nat . 0 < suc(x)
. forall x, y : Nat . x < y => suc(x) < suc(y)
```

3.3 Configuration part

In the configuration part we declare the named configurations of the system together with the axioms that hold for each state, the events causing reconfigurations of the systems and the properties of the entire system. Therefore, we consider three types of declarations.

Declarations of configurations start with the keyword `nominal` (or in the notation for engineers, `state`) and are followed by the name of the state.

Declarations of reconfigurations start with the keyword `modality` (`event` in the notation for engineers) and are followed by the name of the reconfiguration and its arity.

Hybrid logic sentences are either

- base sentences, written in the syntax of the logic being hybridized,
- nominals, thus allowing state names to appear in sentences,
- combination of sentences using the ordinary Boolean connectors,
- modal box and diamond sentences,
- retrieve sentences, that are meant to hold at a given state and,
- depending on the hybrid logic we use, quantifications over nominals and/or symbols from the base logic of a certain kind, e.g. total constants or rigid total constants.

The "Hello world"-like example below introduces a system with three states, all explicitly named, an accessibility relation between them, and such that a proposition `p` declared in the data part `S` of the specification holds in two of them. The underlying logic of the specification is the hybridization of propositional logic with quantification on nominals.

```
data:
 S

configuration:

 %% three state names
nominals s1, s2, s3
```

```
%% one binary modality
modality gamma : 2

%% there are no states other than the named ones
. s1 \/ s2 \/ s3

%% the states are different
. @ s1 : not s2 /\ not s3
. @ s2 : not s1 /\ not s3

%% transitions and labeling function

. @ s1 : <gamma> s2
. @ s1 : <gamma> s3
. @ s1 : p

. @ s2 : [gamma] s2
. @ s2 : <gamma> s2
. @ s2 : not p

. @ s3 : <gamma> s1
. @ s3 : <gamma> s3
. @ s3 : p
```

In the notation for engineers, the same example is written as follows:

```
data:
 S

configuration:

 %% three state names
states s1, s2, s3

%% one binary event
event gamma : 2

%% there are no states other than the named ones
. s1 \/ s2 \/ s3

%% the states are different
. At state s1 : not s2 /\ not s3
. At state s2 : not s1 /\ not s3

%% transitions and labeling function

. At state s1 : Through gamma, sometimes s2
```

```
. At state s1 : Through gamma, sometimes s3
. At state s1 : p

. At state s2 : Through gamma, always s2
. At state s2 : Through gamma, sometimes s2
. At state s2 : not p

. At state s3 : Through gamma, sometimes s1
. At state s3 : Through gamma, sometimes s3
. At state s3 : p
```

3.4 Running Hets

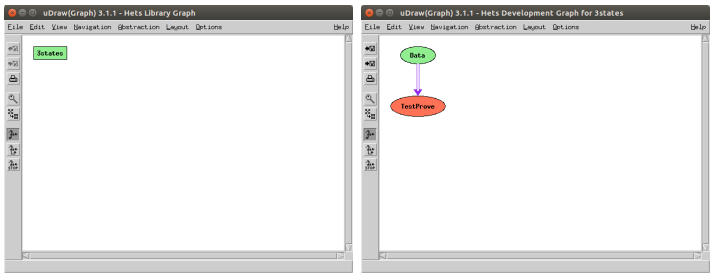

Figure 3.1: H windows.

If the specification is correct, when executed with the `-g` flag, **H** will successfully analyze it and produce two windows: the library window and the development graph window. The library window shows the structure of imported files (left of Fig. 3.1) and the development graph window shows the structure of the specification analyzed (right of Fig. 3.1).

3.5 Typical properties for reconfigurable systems

In the following we present a list of patterns of how properties of reconfigurable systems are written in the **H** syntax.

- The only possible configurations are the named ones:

$$i_1 \vee i_2 \vee ... \vee i_n$$

  where the $i_1, \ldots, i_n$ are all state names declared in the configuration part.

- A property P holds in the state denoted by a nominal i:

```
@i :  P
```

- There is a transition from state i to state j along the event M:

```
@i :  <M> j
```

- In all states reachable from a state i, the property P holds:

```
@i :  [M] P
```

- A property P holds for each state:

```
forallH nominal i .  P
```

  This construction is valid only if the logic must admit quantification on nominals.

3.6 Troubleshooting

If the specification contains errors, **H** will display appropriate error messages or warnings, together with a pointer (file line and column of the start and end in the input file) to the most likely cause of the error. We give a list of the most common errors below:

- `undeclared nominal NAME` – a state name appears in a sentence without having been previously declared.
- `undeclared modality NAME` – an event name appears in a sentence without having been previously declared.
- `no operation with N arguments found for 'NAME'` – an N-ary function symbol appears in a sentence without having been previously declared.
- `unknown sort NAME` – a sort name appears in the arity of a function or predicate symbol without having been previously declared.
- `The sublogic of the analyzed theory should be ..., but it is ...` – occurs if we apply the hybridization method to a sublogic of a Hets logic and we write a specification in a more expressive sublogic of that logic.

Chapter 4

Proofs in H

Conjectures are specified as an extension of specifications, using the syntax

```
spec S' = S then %implies {<list-of-axioms>}
```

where `S` is the specification in a hybrid logic whose properties (given in the `<list-of-axioms>`) we want to verify. In the development graph built during the static analysis of the specification, this is represented as a theorem link with the source in `S'` and target in `S`. After the user selects Edit > Proofs > Auto-DG-Prover, the development graph calculus introduces the sentences from `<list-of-axioms>` as new conjectures in `S`.

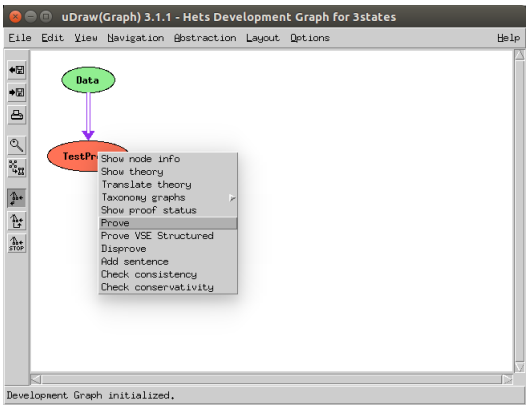

Figure 4.1: Proving at a node.

These conjectures can be discharged by translating the specification to first-order logic and then calling one of the first-order provers connected to **H**. To do this, the user selects "Prove" in the pop-up menu that appears when clicking the right button of the mouse over a node with open proof goals.

The system selects a translation from the current hybrid logic to first-order

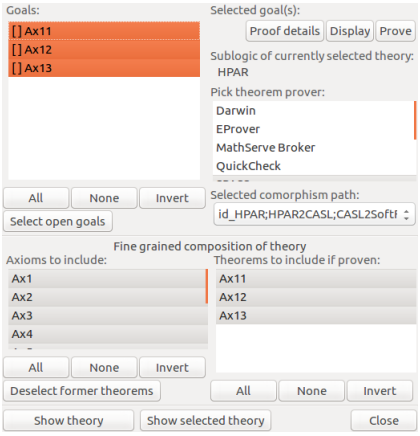

Figure 4.2: Selecting the prover and the conjectures to be proven.

logic[1]. As we see in Fig. 4.2, the user can then select from a list the desired prover and set the proof parameters: which conjectures will be proven and which axioms can be used during the proof (in both cases, by default all). Finally, by pushing the "Prove" button we attempt to prove the selected conjectures. If this is successful, Hets will display a '+' sign before it, if not, a '-' sign, and if the prover returns no answer within the time limit set by the user a 't' sign.

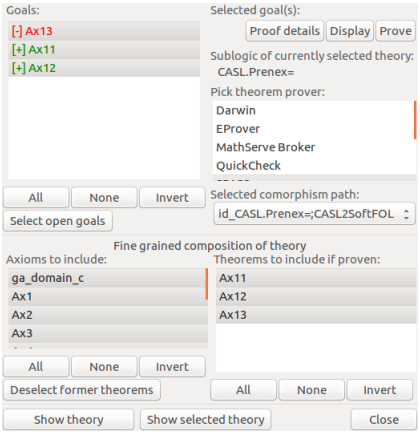

Figure 4.3: The result of proving.

For example, in the case of the system with 3 states introduced in Chap. 3.3, we can check that there is a state where the property p does not hold and moreover that there is a state where after a transition we reach a state where

[1]Actually, this is composed with a translation from first-order logic to the input logic of the provers.

the property p does not hold. We also include a conjecture that is not true, namely that for each state the property $p(c)$ holds:

```
. existsH nominal i . @ i : not p(c) %implied
. existsH nominal i . @ i : <shift> not p(c) %implied
. forallH nominal i . @ i : p(c) %implied
```

In Fig. 4.3 the results of proving are displayed: the first two conjectures are proven (marked with green and '+') while the third is, as expected, not a logical consequence of the specification (marked with red and '-').

Chapter 5

Double hybridization

If the base logic parameter of the bybridization process is itself a hybridized logic, special syntactic features are needed to disambiguate between sentences appearing at different levels of hybridization. Let us assume that the logic used at the base level is named `HI` and the new hybridized logic is named `HHI`. In the case of base formulas and of Boolean connectors, no disambiguation is needed. Nominals and modalities appearing in formulas as well as quantified sentences may be qualified with the name of the logic where they belong: `::HHI` for top-level symbols and `::HHI` for base-level symbols. If the qualification is missing, it defaults to the top-level.

The following example illustrates the new syntactic features and shows how the static analysis of **H** solves qualifications and rejects ill-formed sentences, when the sentences legal at the lower/upper level of quantification are no longer legal at the other level. The logic `HHPropNomsQuant` is the hybridization of the hybridization of propositional logic with nominals `HPropNoms`, and it allows itself quantification on propositional symbols, but not on nominals.

```
spec D =
 logic Propositional :
 {
 props p, q
 . p => q
 }
end

spec S =
 hlogic: HPropNoms

 data: D

 configuration:
```

```
  nominals s1, s2
  modality t: 2

  . @ s1 : <t> s2
  . @ s2 : <t> s1

  . @ s1 : p
  . @ s2 : q
end



spec T =
 hlogic: HHPropNomsQuant

 data: S

 configuration:

 nominals w1, w2
 modality l : 2

 . @ w1 : <l> w2
 . @ w2 : <l> w1

 . forallH :: HPropNoms nominal i . @ i : p
     %% works, base logic admits
     %% quantification on nominals

 . forallH :: HHPropNomsQuant p . @ w1 : p
     %% works, top logic admits
     %% quantification on propositions

 . forallH p . @ w1 : p
     %% works, no qualification
     %% defaults to top logic

 . forallH nominal i . @ i : p
     %% incorrect, defaults to top logic

 . forallH :: HHPropNomsQuant nominal i
   . @ i : p
     %% incorrect, no quantification
     %% on nominals in top logic

 . forallH :: HPropNoms nominal i
```

```
    . forallH p
      . @ i : p
      %% incorrect, no quantification
      %% on propositions in base logic

  . forallH :: HPropNoms nominal i
      . forallH :: HHPropNomsQuant p
        . @ i : p
      %% incorrect, formula from top layer
      %% can't appear in a base formula
end
```

The following double hybridizations are already available in our distribution of **H**:

HHProp double hybridization of propositional logic, no quantification, no constraints,

HHPropNom1 double hybridization of propositional logic, quantification on the top-level nominals,

HHPropNom double hybridization of propositional logic, quantification on low-level nominals, no quantification at the top-level,

HHPropNomNom1 double hybridization of propositional logic, quantification on both top-level and low-level nominals.

Chapter 6

Structured specifications

H specifications can be written in a modular way, such that their complexity is reduced. This is particularly useful for large systems. We illustrate the specification structuring constructs of the **H** specification language with a running example in the hybridization of first-order logic.

```
spec SD =
 logic CASL :
 {
 sort s
 op c : s
 }
end

spec TD =
 logic CASL :
 {
 sort t
 op d : t
 }
end

spec S =
 hlogic: HFOL
 data: SD
 configuration:
   nominal ws
   modality lam : 2
end

spec T =
```

```
 hlogic: HFOL
 data: TD
 configuration:
    nominal wt
    modality gamma : 2
end
```

Firstly, symbols of a, **H** specification can be renamed, thus avoiding name clashes or giving the symbol a name that is better suited in a different context. This is done by specifying what the new name of a symbol is, using the syntax `oldName |-> newName`. In the case of nominals and modalities, we must add the keyword `nominal` or `modality` before the old name of the symbol.

```
spec SRen =
 S with nominal ws |-> ws', modality lam |-> lam'
```

Specifications can be united. Here, the "same name, same thing" principle from CASL applies: common symbols appear in the union only once.

```
spec STUnion =
 S and T
```

Finally, specifications can be extended with new declarations of nominals, modalities or axioms.

```
spec SExt =
 S then
 {nominal wse}
```

Chapter 7

Adding new hybridized logics in H

If a hybridized logic is missing in **H**, it can be added with moderate effort. This is however a rather advanced feature of the system, as it requires recompilation of the **H** sources. The steps for doing this are as follows:

1. download the Hets sources:

   ```
   $ git clone https://github.com/spechub/Hets.git
   $ cd Hets
   ```

2. switch on the **H** development branch:

   ```
   $ git checkout rigid_casl
   ```

   In the near future, this branch will be merged on the main branch of Hets, so this step will no longer be needed.

3. setup the compilation of the Hets sources, following the instructions at `http://hets.eu`, section `Build Hets using Stack`.

4. if the base logic that you want to hybridize is missing in Hets, it should be implemented. This requires knowledge of the Haskell programming language.

5. the following parameters of the hybridization process must be set:

   - name in Hets of the logic being hybridized. If it appears a sublogic of a main logic, the syntax is `logicName.sublogicName`,
   - list of constraints imposed on the local models. These can be restrictions on the interpretation of the accessibility relation or on the interpretation of symbols of a certain kind. For the former, this is specified

using one of the keywords `Reflexive, Transitive, Symmetric, Serial, Euclidean, Functional, Linear, Total`, corresponding to typical constraints on the accessibility relations in modal logics, while for the latter, we can choose from

- `SameInterpretation(nominal)`,
- `SameInterpretation(world)`,
- `SameInterpretation(<kind>)`, where the kinds appearing in `<kind>` can be symbol kinds in the base logic (e.g. proposition, constant, total function, etc.),
- `SameDomain(p)` where `p` is either `partial` or `rigid partial`, to state that the domain of partial/rigid partial functions is the same in each local model

as in the example below:

```
newhlogic HProp =
 base: Propositional .
end

newhlogic HHPropNomsQuant =
 base: HPropNoms .
 quant: prop .
end

newhlogic HRigidCASLC =
 base: RigidCASL .
 constr: SameInterpretation (rigid sort),
         SameInterpretation(rigid total op),
         SameDomain (rigid partial) .
 quant: rigid const, nominal .
end
```

Alternatively we can select a hybridized logic already specified in Hets and add more constraints or more kinds of symbols allowed in quantifications. For example, below we extend the existing definition of the hybridization of propositional logic with quantification on nominals:

```
newhlogic HPropNoms =
 hlogic: HProp .
 quant: nominal .
end
```

The new hybridized logic definition must be written in a file that contains only one such definition. Let us assume the file name is `newlogic.dol`.

6. in the folder with the hets sources, analyze the new definition

```
Hets$ ./hets newlogic.dol
```

This generates a new instance of the Haskell class Logic of Hets, in a new folder in the source code.

7. The generated code must be compiled:

   ```
   make
   ```

   Thus the new logic is made available in **H**, and we can write specifications in that logic.

7.1 Hybridization of comorphisms

To get proof support for such a hybridized logic, we must also lift a translation from its base logic to first-order logic to a translation from the hybridized logic to first-order logic. This is also done by instantiating a generic method, whose parameters are the Hets name of the comorphism being lifted and the name of the hybridized logic (this is needed because the base logic admits more than one hybridization):

```
newhcomorphism HRigid2CASL =
 basecomorphism: Rigid2CASL
 sourcehlogic: HRigidCASLC
end
```

Again this definition must be written in a file `newcom.dol` that is next analysed in **H** by calling `Hets$./hets newcom.dol` and the resulting Haskell code must be compiled with `make`. Thus the new comorphism is made available in **H** and we can use it for making proofs.