Hybridisation of Institutions in HETS

Mihai Codescu
University of Bremen, Collaborative Research Center EASE, Bremen, Germany
Institute of Mathematics "Simion Stoilow" of the Romanian Academy,
Research Group of the project PED-0494, Bucharest, Romania
codescu@uni-bremen.de

Abstract

We present a tool for the specification and verification of reconfigurable systems. The foundation of the tool is provided by a generic method, called hybridisation of institutions, of extending an arbitrary base institution with features characteristic to hybrid logic, both at the syntactic and the semantic level. Automated proof support for hybridised institutions is obtained via a generic lifting of encodings to first-order logic from the base institution to the hybridised institution. We describe how hybridisation and lifting of encodings to first-order logic are implemented in an extension of the Heterogeneous Tool Set in their full generality. We illustrate the formalism thus obtained with the specification and verification of an autonomous car driving system for highways.

2012 ACM Subject Classification Software and its engineering → Specification languages; Theory of computation → Algebraic semantics; Theory of computation → Logic and verification

Keywords and phrases hybrid logics, formal verification, institutions, reconfigurable systems

Digital Object Identifier 10.4230/LIPIcs.CALCO.2019.17

Category Tool Paper

Funding This work was partially supported by the German Research Foundation DFG, as part of Collaborative Research Center (Sonderforschungsbereich) 1320 "EASE - Everyday Activity Science and Engineering", University of Bremen (`http://www.ease-crc.org/`) and 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.

Acknowledgements Răzvan Diaconescu had major contributions to the implementation in HETS of his generic method of hybridisation of institutions. I wish to thank Till Mossakowski, Fabian Neuhaus and Ionuţ Ţuţu for valuable feedback on language design and implementation issues.

1 Introduction

A *reconfigurable system* is one with different modes of operation, called *configurations*, and with the ability to commute between them during its execution along transitions between these modes, called *reconfigurations*. Such systems appear naturally in many domains, including automobile industry, robotics and medical devices. An overview can be found in [9]. We present H [3], a tool for the formal specification and verification of reconfigurable systems that supports their correct and efficient development.

The mathematical foundation underlying the tool is provided by a generic construction on institutions [8], called *hybridisation*, explained briefly in Sec. 2. It has the modalisation of institutions [7] as its source; part of that work was extended to hybrid logics in [10] in a simple form, and it took a rather complete shape in [5]. Hybridisation is done using a two-layered approach: the base layer represents a specific logic for expressing requirements at the configuration (static) level, in other words at the data level. This layer is treated abstractly as an institution that can be instantiated to concrete logical formalisms that are most adequate for the specification of the data part of particular problems. On top of this base layer the characteristic syntactic and semantics features of hybrid logic are

developed, notably a flexible choice of *quantifications* on nominals and/or symbols from the base institution and various *semantic constraints* on the accessibility relations and on the interpretation of symbols in possible worlds of a Kripke model of the hybridised institution (also see [5]). While the base layer deals with the data level, the upper layer deals with the dynamics of the configurations. The choice of hybrid logic for expressing the latter comes naturally as it is a prominent kind of modal logic that provides adequate syntactic capabilities – names of possible worlds, formulas that hold at named states.

The specification part of our tool is complemented by a verification part, whose foundation is given by a general encoding of hybridised institutions into first-order logic. This encoding follows the two-layered structure of the hybridised institutions. If a translation of the logic used at the base level to first-order logic already exists, it is *lifted* to a translation of the hybridised institution to first-order logic via a generic construction, introduced in [6]. At this level, the semantic constraints give rise to first-order formulas. As a result, we obtain a *verification-by-translation* method where a problem in a hybridised institution is translated to first-order logic and solved there using automated first-order theorem provers.

The H tool was implemented as an extension of the Heterogeneous Tool Set (HETS) [13], a tool for the heterogeneous multi-logic specification and modeling of software systems and for ontology development. In all these fields, there is a large number of logics and languages in use, each better suited for a different task or providing better support for a different aspect of a complex system. Instead of trying to integrate the features of all these logics into a single formalism, the paradigm of heterogeneous multi-logic specification is to integrate all logics by means of a so-called Grothendieck construction over a graph of logics and their translations [12, 4]. Thus, for each logic we can make use of its dedicated syntax(es) and proof tools. The specifier has the freedom to choose the logic that suits best the problem to be solved, offers best tool support and is most familiar with. HETS provides an implementation of this paradigm, and also supports the verification-by-translation method. HETS has been designed as a flexible tool: adding a new logic or a new logic translation can be done by instantiating a class and adding the new instance to the list of known logics and translations. First-order logic and several state-of-the-art automated provers for it are already supported by HETS.

A HETS implementation of the hybridisation method was presented in [14]. It was realized in two directions: an implementation of the hybridisation of an extension of CASL logic with rigid symbols (enabling *user-defined sharing*, when only the symbols explicitly marked as rigid are subject to semantic constraints in the models of the hybridised institution), together with a translation from this hybridisation to first-order logic, and a generic construction, similar to a Grothendieck one, that appears as a single logic in HETS, used to hybridise a number of HETS logics. No translation from this logic to first-order logic is available, and no choice can be made on the kind of quantification and the semantic constraints that a hybridised institution should have. In contrast, our implementation supports all variations and is fully generic: the parameters of the hybridisation method can be specified in a declarative way and new instances of the main `Logic` class of HETS are generated for each new definition of a hybridised institution. Generating different institutions for different hybridisations is crucial for defining comorphisms from them to first-order logic, which is also implemented in our tool as a generic method, thus enabling proof support for each newly added hybridised institution.

2 Institutions and their hybridisation

Institutions [8] provide a model-theoretic formalization of the concept of logical system. The basic components of an institution are: a notion of *signature*, defining the non-logical symbols in the language, a notion of *logical sentence* over a signature, a notion of *model* giving the

interpretation of the symbols in a signature in some semantic domain and a *satisfaction relation* between the models and the sentences of a signature. This is complemented by a dynamic view on the language: instead of working over an arbitrary but fixed (and implicit) signature, different signatures are related by *signature morphisms*, which induce *translations of sentences* and *reduction of models*. These must be consistent with one another, which means that no change of notation induced by a signature morphism can alter the satisfaction of sentences. This is expressed formally by the so-called *satisfaction condition*. Institutions are a formalization of the above using category theory, thus achieving a high level of abstraction and not making unnecessary assumptions about the components of a logical system.

The hybridisation method [5] was introduced at this abstract level. Given an arbitrary base institution $\mathcal{I}$ as a first parameter, it constructs a hybridised institution $\mathcal{HI}$ whose signatures extend the signatures of $\mathcal{I}$ with nominals (for reconfigurable systems, these correspond to names of configurations) and modalities (names of events causing reconfigurations). Signature morphisms in $\mathcal{I}$ pair signature morphisms in $\mathcal{I}$ with mappings of nominals and of modalities. For a signature in $\mathcal{I}$, the sentences can be $\mathcal{I}$-sentences over the base signature, nominals, modal box- and diamond-sentences over modalities, *retrieve* sentences (meant to hold at a given state), combinations of sentences using Boolean connectors, or quantified sentences. The latter sentences depend on a class $\mathcal{D}$ of signature morphisms of $\mathcal{HI}$ that defines the kind of variables that can be quantified, nominals and/or symbols from $\mathcal{I}$, and forms the second parameter of hybridisation. Models of a $\mathcal{HI}$ signature are Kripke structures such that each possible world is assigned a $\mathcal{I}$-model of the base signature, each nominal is interpreted as one of the possible worlds and each modality as an accessibility relation between these worlds. The third parameter of hybridisation is a set of logic-specific semantic constraints on the accessibility relations and on the interpretation of symbols in possible worlds. For example, the accessibility relation may be reflexive and transitive, as in the modal logic **S4**, or the interpretation of all symbols of a certain kind may be the same in all possible worlds.

Institution comorphisms [11] capture the intuition that an institution is included or encoded into another one. A comorphism from an institution $\mathcal{I}_1$ to an institution I_2 maps $\mathcal{I}_1$-signatures to $\mathcal{I}_2$-signatures along a functor Φ, Σ-sentences in $\mathcal{I}_1$ to $\Phi(\Sigma)$-sentences in $\mathcal{I}_2$ for a $\mathcal{I}_1$-signature Σ and $\mathcal{I}_2$-models of $\Phi(\Sigma)$ to $\mathcal{I}_1$-models of Σ. Again, a satisfaction condition must hold, stating that satisfaction of sentences is not altered by change of logic. Sometimes the cost of encoding an institution $\mathcal{I}_1$ into another one $\mathcal{I}_2$ is that $\mathcal{I}_1$-signatures are mapped to $\mathcal{I}_2$-theories, i.e. not just signatures, but also a set of sentences over them. These theories grow in size with the number of symbols in the original signature.

Given an institution comorphism from an institution $\mathcal{I}$ to the institution of multi-sorted first-order logic FOL^{ms}, [6] introduces a generic method of lifting it to a comorphism from a hybridisation $\mathcal{HI}$ of $\mathcal{I}$ to FOL^{ms}. A hybrid signature Δ get translated to a FOL^{ms}-theory (Σ, E) as follows: first the base signature is translated along the base comorphism, and we obtain a first-order theory. This theory is extended with a new sort for states, its predicates and function symbols get a new argument of sort state, and the sentences of the theory are universally quantified over a variable of sort state that is introduced in all predications and all terms. Domain predicates are introduced for each sort and state, giving the interpretation of that sort in each world. Nominals are constants of sort state and modalities are predicates on states. Moreover, semantic constraints get translated to sentences over this extended signature. The reduction of a (Σ, E)-model to a Δ-model is done by taking as the set of worlds the interpretation of the sort for states, and by keeping the interpretation of nominals and modalities as in the first-order model. The local models are obtained for each world w by taking the reduct along the base comorphism of the first-order model obtained by interpreting

each sort as its domain in w and each function/predicate symbol as the restriction of its interpretation in (Σ, E) when the extra state argument is always w. Sentence translation is done by adding an universal quantification on a variable w of sort state and then inductively on the structure of the formula, with the base cases of nominals i being translated to $i = w$ and base formulas e being first translated along the base comorphism and then adding w in the resulting first-order sentences as the extra argument of sort state. Details of the interesting cases of box- and diamond formulas and quantification can be found in [6].

3 Hybridisation in HETS

The parameters of the generic hybridisation method are:

(1) the base institution being hybridised, using the name of a known logic in HETS or even of one of its sublogics, written in HETS syntax as `LogicName.SublogicName`,

(2) the kind of symbols allowed to appear in a quantification, which can be `nominal` or a kind of symbols of the base institution, referred to by its HETS name,

(3) the constraints made on the models of the hybridised institution, which can be of two kinds: on the accessibility relations between possible worlds (reflexive, transitive etc.) or on the interpretation of symbols of a certain kind (universes, nominals, or a kind from the base institution) in the possible worlds.

Listing 1 shows how the hybridisation of the extension of the CASL logic with rigid symbols, that will be used in the example in Sec. 4, is specified[1]: first we define `HRigidCASL` as the hybridisation of `RigidCASL` with quantification on rigid constants and nominals. After this definition is analyzed by HETS, it is recorded for further extensions: `HRigidCASLC` adds the constraints that rigid sorts, rigid predicates and rigid total functions share the interpretation and rigid partial functions share the domain of definition in each possible world of a model. HETS generates for each of these two definitions a new instance of the class `Logic`, which will become available for specification in HETS once the newly generated code is compiled. The resulting logic will inherit the syntax of the base institution for declarations of base symbols and for base sentences, and will use generic syntax for declarations of nominals and modalities, and for hybrid sentences. Full details of the syntax are available at [2].

Listing 1 Hybridisation of `RigidCASL`.

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

The process of lifting a comorphism to first-order logic from a base institution to its hybridisation has only two parameters: the HETS name of the comorphism being lifted and the name of the hybridisation of the base institution that will be the source of the lifted comorphism. The latter is needed because a base institution admits more than one hybridisation. Again, HETS analyses this definition and generates source code that must be compiled to make the comorphism available for translation and proofs by translation.

[1] More examples can be found at `https://ontohub.org/forver`.

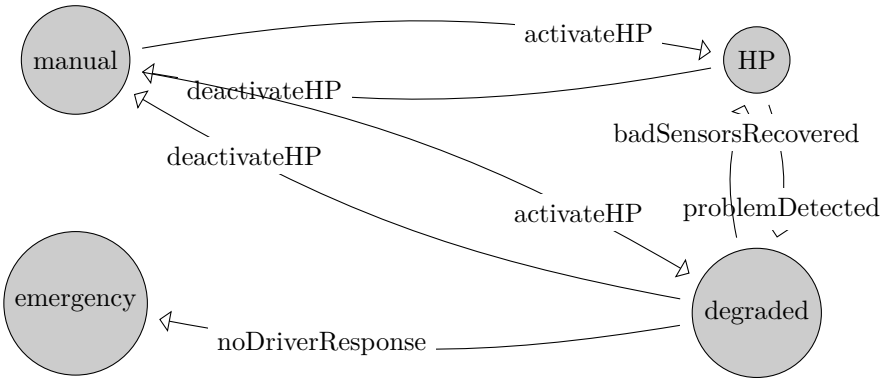

Figure 1 The modes and reconfigurations of the highway pilot.

Listing 2 Lifting the translation to first-order logic of `RigidCASL` to `HRigidCASLC`.

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

4 Case Study: specification and verification of a highway pilot

We now discuss the example of an autonomous driving system for passenger cars and heavy trucks, called highway pilot (HP). The problem description is adapted from [16]. HP can only be activated when driving on a highway. After activation, the electronic system will keep driving the car on the highway, relying on information from radar and camera sensors. These sensors may exhibit faults or may fail to give a correct interpretation of the surroundings, depending on weather, traffic and road conditions. Unlike bad conditions, faults will not disappear after some time unless the sensors are physically repaired. Faults and bad conditions may be undetected for some time. If a fault or bad conditions are detected when HP is on, the system will enter a so-called `degraded` driving mode, where a safer driving style is adopted, typically including driving slower, and the driver is alerted that he/she should take over driving. If the driver does not deactivate HP mode within a time limit after being alerted, an emergency stop will be performed. If the bad conditions disappear (all unreliable sensors become reliable) before the time limit for HP deactivation is reached and there are no faulty sensors, the system will return to HP mode and stop alerting the driver.

The modes of the systems and the events causing changes of modes are depicted in Fig. 1. In Listing 3 we show how they are specified, together with axioms stating that there are no other modes, that the system can change from the manual to HP and degraded modes and that the only transitions from the modes HP and degraded along the reconfiguration `deactivateHP` are to the manual mode.

Listing 3 Modes.

```
nominals manual, hp, degraded, emergency
modalities activateHP, deactivateHP, problemDetected,
           noDriverResponse, badSensorsRecovered : 2

. manual \/ hp \/ degraded \/ emergency    %(no_other_states)%
. @ manual : <activateHP> (hp \/ degraded) %(manual_to_hp_or_degraded)%
. not (manual \/ emergency)
   => <deactivateHP> manual /\ [deactivateHP] manual %(back_to_manual)%
```

We also keep track of the current state of the HP system, using a different transition system than the one between modes. The system states are loosely specified with the help of observers for the current speed, the current status of the sensors (working, detected bad conditions, detected fault, undetected bad conditions, undetected faults), flags for checking whether the driver alert is on or off and if the driver has turned the HP off and a time counter for checking the time limit in the degraded mode. The transitions between states are given by a non-rigid predicate `step`. In each mode, transitions are possible only from the states that are valid in that mode, as defined by a non-rigid predicate `isValid`. Recall that non-rigid symbols admit different interpretations in different possible worlds. Listing 4 shows the definition of valid states for the mode `hp`: those reached after HP was activated from `manual` mode, those reached after bad sensors recovered in `degraded` mode and those that are reached from a valid state in the `hp` mode and do not satisfy the conditions for reconfigurations.[2]

Listing 4 Highway pilot mode.

```
. @ hp : forallH S' : State . isValid(S') <=> crtDeactivateTime(S') = 0 /\
     (( existsH S : State . @ manual :
        isValid(S) /\ step(S, S') /\ activateHPWorkingCond(S'))
     \/ ( existsH S : State . @ degraded :
           isValid(S) /\ step(S, S') /\ badSensorsRecoveredCond(S'))
     \/ ( existsH S : State . @ hp :
           isValid(S) /\ step(S, S') /\
           not hpDeactivated(S') /\ not problemDetectedCond(S')))
        %(def_isValid_hp)%
```

The other modes are specified in a similar way[3]. We can now verify that the valid states of each mode have the expected properties. The corresponding sentence for the mode `hp`, stating that all valid states have no detected faulty or unreliable sensors is shown in Listing 5. We can prove this conjecture in HETS by translation, using the SPASS prover with a time limit of 70 seconds on a modern machine. Proving that in every state valid in `degraded` mode there is at least one faulty or unreliable sensor takes significantly more time (time limit of 500 seconds with SPASS) and requires the introduction of a lemma. This is typical for proofs in first-order logic, especially in the case of very large theories as the one obtained in this case via translation.

Listing 5 Conjectures.

```
. forallH S : State . (@ hp : isValid(S))
   => not exists s : Sensor .
           status(S, s) = detectedBad \/ status(S,s) = detectedFault
    %(no_detected_problems_hp)% %implied
```

5 Conclusions and future work

By implementing in HETS the hybridisation method and the lifting of translations introduced in [5, 6], we obtain a framework for specification and verification of reconfigurable systems. Proofs are done by translation to first-order logic using the first-order provers already integrated with HETS. Given the large variety of hybrid institutions that can be specified, this is often the only tool support available. An interesting enterprise would be to implement

[2] Note that **forallH** and **existsH** are the universal and existential quantifiers introduced via hybridisation; their semantics does not always subsume that of quantification in the base logic, see [5].

[3] The complete specification of the HP system is available under `https://ontohub.org/forver/hp.dol`.

a generic parameterized prover for hybrid logics, possibly following the ideas of [1], and to make it available in HETS for each generated hybridised institution. The results of [15] hold for a restricted form of hybridisation, without quantifications on nominals and with no constraints on models and therefore cannot be applied in our setting.

References

1 D. Găină. Birkhoff style calculi for hybrid logics. *Formal Asp. Comput.*, 29(5):805–832, 2017. doi:10.1007/s00165-016-0414-y.

2 M. Codescu and R. Diaconescu. Hspec language definition. URL: http://imar.ro/~diacon/forver/Hdef.pdf.

3 M. Codescu and R. Diaconescu. The H system. http://imar.ro/~diacon/forver/forver.html.

4 R. Diaconescu. Grothendieck Institutions. *Applied Categorical Structures*, 10(4):383–402, 2002. doi:10.1023/A:1016330812768.

5 R. Diaconescu. Quasi-varieties and initial semantics for hybridized institutions. *J. Log. Comput.*, 26(3):855–891, 2016. doi:10.1093/logcom/ext016.

6 R. Diaconescu and A. Madeira. Encoding hybridized institutions into first-order logic. *Mathematical Structures in Computer Science*, 26(5):745–788, 2016. doi:10.1017/S0960129514000383.

7 R. Diaconescu and P. S. Stefaneas. Ultraproducts and possible worlds semantics in institutions. *Theor. Comput. Sci.*, 379(1-2):210–230, 2007. doi:10.1016/j.tcs.2007.02.068.

8 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. doi:10.1145/147508.147524.

9 A. Madeira, R. Neves, L. Soares Barbosa, and M. A. Martins. A method for rigorous design of reconfigurable systems. *Sci. Comput. Program.*, 132:50–76, 2016. doi:10.1016/j.scico.2016.05.001.

10 M. A. Martins, A. Madeira, R. Diaconescu, and L. Soares Barbosa. Hybridization of Institutions. In *CALCO*, volume 6859 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2011. doi:10.1007/978-3-642-22944-2_20.

11 J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Proceedings, Logic Colloquium, 1987*, pages 275–329. North-Holland, 1989.

12 T. Mossakowski. Comorphism-based Grothendieck logics. In K. Diks and W. Rytter, editors, *Mathematical foundations of computer science*, volume 2420 of *Lecture Notes in Computer Science*, pages 593–604. Springer Verlag, London, 2002. doi:10.1007/3-540-45687-2_49.

13 T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editors, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer, Heidelberg, 2007. doi:10.1007/978-3-540-71209-1_40.

14 R. Neves, A. Madeira, M. A. Martins, and L. Soares Barbosa. Hybridisation at Work. In R. Heckel and S. Milius, editors, *CALCO 2013*, volume 8089 of *Lecture Notes in Computer Science*, pages 340–345. Springer, 2013. doi:10.1007/978-3-642-40206-7_28.

15 R. Neves, A. Madeira, M. A. Martins, and L. Soares Barbosa. Proof theory for hybrid(ised) logics. *Sci. Comput. Program.*, 126:73–93, 2016. doi:10.1016/j.scico.2016.03.001.

16 M. Nyberg. Safety analysis of autonomous driving using semi-Markov processes. In Stein Haugen, Anne Barros, Coen van Gulijk, Trond Kongsvik, and Jan Erik Vinnem, editors, *Safety and Reliability–Safe Societies in a Changing World*, pages 781–788. CRC Press, 2018. doi:10.1201/9781351174664-97.

A Implementation

In this appendix we give an overview of how the hybridisation method was implemented in HETS. We have made some simplifications and changes of the actual names used in the HETS source code to ease understanding.

A.1 Adding a new logic in HETS

HETS has an abstract interface for logics, in the form of a Haskell multiparameter type class with functional dependencies, called `Logic`. The parameters are types for the constituents of a logic: its identifier, its signatures and signature morphisms, its symbols of signatures with their kinds, its low-level, human readable syntax in the form of basic specifications for theories, lists of symbols for convenient use during structuring and symbol maps for signature morphisms, its sublogics and its proof trees. Being a type class, `Logic` provides a list of methods that must be provided for each particular choice of the parameter types in order to obtain a new instance of the type class. These include, among others, composition of signature morphisms, parsers, printers, static analysis of basic specifications, symbol lists and symbol maps, sentence translation along signature morphisms, various operations on signatures and signature morphisms. The functional dependency between the type logic identifier and all other types is used to determine the missing types and thus the correct instance of a function in the type class. This means that all methods in the class `Logic` take as first argument the logic identifier, and this determines the logic uniquely.

To sketch an example, propositional logic has as identifier a singleton type, called `Propositional`. Types must be provided for its constituents: signatures are sets of names (implemented in HETS using the datatype `Id`) of propositional symbols, signature morphisms are maps between these sets, where we also store the source and the target signature for each morphism, and so on.

Listing 6 Signatures in propositional logic.

```
newtype PropSign = PropSign {items :: Set Id}
```

The type class `Logic` contains a method for union of signatures: `signature_union :: lid -> sign -> sign -> Result sign`, where the type `Result a` is a polymorphic type with variable `a` used for dealing with errors (the union of signature may not give a legal signature for each institution). We must provide an implementation of this method for propositional logic, and this will be done as a method `signatureUnion :: PropSign -> PropSign -> Result PropSign`. Then we must provide an instance declaration for the types for components of propositional logic where we say how the methods of the type class are implemented.

Listing 7 Propositional logic.

```
instance Logic -- the type class
         Propositional -- the logic identifier
         ...
         PropSign -- the type of signatures
         PropMorphism
         ...
   where
    ...
    signature_union Propositional = signatureUnion
    ...
```

To sum up, adding a new logic in HETS requires creating a new instance of the `Logic` class, and this is achieved by defining types for the constituents of that logic and by implementing the methods of the `Logic` class for these types.

A.2 Generic implementation of hybridisation in HETS

The first step is to define the generic types for the constituents of a hybridised institution. We used type variables for the parts that come from the base institution. For example, the type of hybrid signatures is presented in Listing 8.

Listing 8 Hybrid signatures.

```
data HSign sig = HSign {
                   baseSig :: sig,
                   noms :: Set Id,
                   mods :: Set Id}
```

where the variable `sig` stands for the base signatures. Then we need to implement the methods of the `Logic` class over these generic types. Typically this will require that the corresponding method in the base institution is involved, and we need to make it accessible. This is achieved by giving as an argument the logic identifier of the base institution and imposing the condition that the type variables that appear in the generic types introduced at the first step will be instantiated with the corresponding types from the base institution. For example the method for union of hybrid signatures presented in Listing 9 will have to make the union of the base signatures. We add the requirement that the argument for the type variable `sig` is the type of signatures of the base institution, as recorded in the Haskell context of the method `sigUnion`. The identifier `baseLid` allows us to properly identify the `Logic` instance of the base institution, and thus `signature_union baseLid` will invoke the implementation of signature union in the base institution, for the base signatures of the hybrid signatures that we want to unite. Then we unite the sets of nominals and modalities, respectively, and return the result.

Listing 9 Union of hybrid signatures.

```
sigUnion :: (Logic baseLid ... sig ...)
          =>  baseLid -> HSign sig -> HSign sig ->  Result (HSign sig)
sigUnion baseLid hsig1 hsig2 = do
 usig <- signature_union baseLid (baseSig hsig1) (baseSig hsig2)
 let uNoms = Set.union (noms hsig1) (noms hsig2)
     uMods = Set.union (mods hsig1) (mods hsig2)
 return $ HSign usig uNoms uMods
```

As a result of these two steps, we obtain generic types for hybrid institutions and generic implementations of the methods in the `Logic` class for these types. Let us assume we want to extend HETS with the hybridisation of propositional logic, with no quantification and no semantic constraints on models. This is written as in Listing 10.

Listing 10 Hybridisation of propositional logic.

```
newhlogic HProp =
 base: Propositional .
end
```

When HETS analyzes this definition, it generates a new instance of the `Logic` class, whose component type for signatures is `HSign PropSign`, and similarly for the other component types of a logic. The instance declaration in Listing 11 states that the implementation of signature union for the new logic `HProp` is given by the method `sigUnion` introduced in

Listing 9. It uses partial application: the methods on both sides of the equal sign take as arguments two hybrid signatures. `HProp` is the unique value of the singleton type `HProp` generated as a logic identifier for the new hybridised logic that we want to define.

Listing 11 Logic instance for hybrid propositional logic.

```
instance Logic
         HProp
         ...
         (HSign PropSign)
         ...
  where
   ...
   signature_union HProp = sigUnion Propositional
   ...
```