

The Heterogeneous Tool Set

Till Mossakowski¹, Christian Maeder¹, Klaus Lüttich¹, and Stefan Wölfl²

¹ DFKI Lab Bremen and Department of Computer Science, University of Bremen, Germany

² Department of Computer Science, University of Freiburg, Germany

Abstract. Heterogeneous specification becomes more and more important because complex systems are often specified using multiple viewpoints, involving multiple formalisms. Moreover, a formal software development process may lead to a change of formalism during the development. However, current research in integrated formal methods only deals with ad-hoc integrations of different formalisms.

The heterogeneous tool set (Hets) is a parsing, static analysis and proof management tool combining various such tools for individual specification languages, thus providing a tool for heterogeneous multi-logic specification. Hets is based on a graph of logics and languages (formalized as so-called institutions), their tools, and their translations. This provides a clean semantics of heterogeneous specification, as well as a corresponding proof calculus. For proof management, the calculus of development graphs (known from other large-scale proof management systems) has been adapted to heterogeneous specification. Development graphs provide an overview of the (heterogeneous) specification module hierarchy and the current proof state, and thus may be used for monitoring the overall correctness of a heterogeneous development.

1 Introduction

“... it is at present extremely difficult to integrate in a rigorous way different formal descriptions, and to reason across such descriptions. This situation is very unsatisfactory, and presents one of the biggest obstacles to the use for formal methods in software engineering because, given the complexity of large software systems, it is a fact of life that no single perspective, no single formalization of level of abstraction suffices to represent a system and reason about its behaviour.” [29]

“As can be seen, a plethora of formalisms for the verification of programs, and, in particular, for the verification of concurrent programs has been proposed. ... *there are good reasons to consider all the mentioned formalisms, and to use whichever one best suits the problem.*” [44] (italics in the original)

In the area of formal specification and logics used in computer science, numerous logics are in use:

- logics for specification of datatypes,
- process calculi and logics for the description of concurrent and reactive behaviour,
- logics for specifying security requirements and policies,
- logics for reasoning about space and time,
- description logics for knowledge bases in artificial intelligence and for the semantic web,

- logics capturing the control of name spaces and administrative domains (e.g. the ambient calculus), etc.

Indeed, at present, it is not imaginable that a combination of all these (and other) logics would be feasible or even desirable — even if it existed, the combined formalism would lack manageability, if not become inconsistent. Often, even if a combined logic exists, for efficiency reasons, it is desirable to single out sublogics and study translations between these (cf. e.g. [44]). Moreover, the occasional use of a more complex formalism should not destroy the benefits of *mainly* using a simpler formalism.

This means that for the specification of large systems, heterogeneous multi-logic specifications are needed, since complex problems have different aspects that are best specified in different logics. Moreover, heterogeneous specifications additionally have the benefit that different approaches being developed at different sites can be related, i.e. there is a formal interoperability among languages and tools. In many cases, specialized languages and tools often have their strengths in particular aspects. Using heterogeneous specification, these strengths can be combined with comparably small effort.

Current heterogeneous languages and tools do not meet these requirements. The heterogeneous language UML [7] deliberately has no formal semantics, and hence is not a formal method or logic in the sense of the present work. (However, UML could be integrated in the Heterogeneous Tool Sets as a formalism without semantics, while the different formal semantics that have been developed for UML in the meantime would be represented as logic translations.) Likewise, languages for mathematical knowledge management like OpenMath and OMDoc [21] are deliberately only semi-formal. Service integration approaches like MathWeb [47], Modelware [1] or JETI [27] are either informal, or based on a fixed formalism. Moreover, there are many bi- or trilateral combinations of different formalisms, consider e.g. the integrated formal methods conference series [3, 18, 12, 11]. Integrations of multiple decision procedures and model checkers into theorem provers, like in the PROSPER toolkit [15], provide a more systematic approach. Still, these approaches are uni-lateral in the sense that there is one logic (and one theorem prover, like the HOL prover) which serves as the central integration device, such that the user is forced to use this central logic, even if this may not be needed for a particular application (or the user may prefer to work with a different main logic).

By contrast, the heterogeneous tool set is a both flexible, multi-lateral *and* formal (i.e. based on a mathematical semantics) integration tool. Unlike other tools, it treats logic translations (e.g. codings between logics) as first-class citizens. HETS consists of about 60.000 lines of Haskell code; roughly the half of which is logic independent. The architecture of the heterogeneous tool set is shown in Fig. 1. In the sequel, we will explain the details of this figure.

2 Institutions, Entailment Systems and Logics

Heterogeneous specification is based on individual (homogeneous) logics and logic translations. To be definite, the terms 'logic' and 'logic translation' need to be formalized in a precise mathematical sense. We here use the notions of *institution* [17] and *entailment system* [28], and of *comorphism* [43] between these.

Logical theories are usually formulated over some (user-defined) vocabulary, hence it is assumed that an institution provides a notion of *signature*. Especially for modular

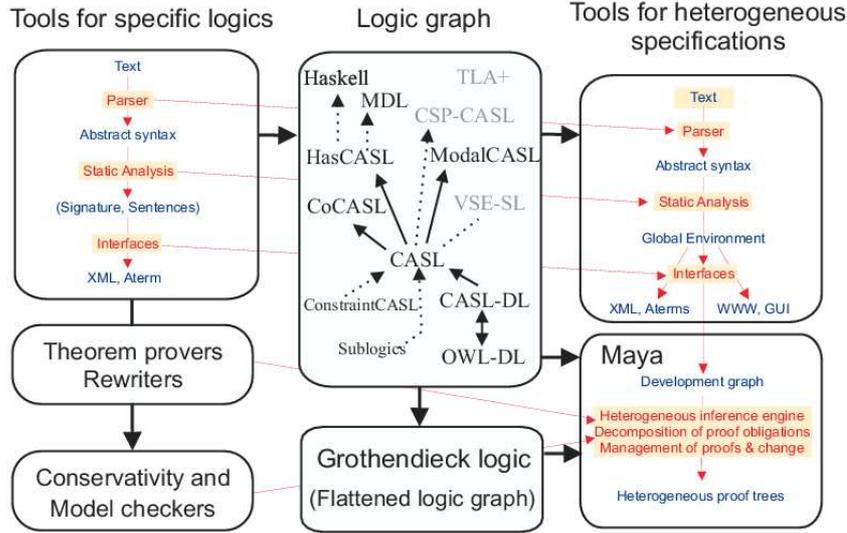


Fig. 1. Architecture of the heterogeneous tool set

specification, it is important to be able to relate signatures, which is done by *signature morphisms*. These can be composed, and hence form a *category of signatures and signature morphisms*. Furthermore, an institution provides notions of *sentences* and *models* (over a given signature Σ). Models and sentences are related by a *satisfaction relation*, which determines when a given sentence holds in a model. An entailment system also provides an *entailment (provability) relation*, which allows to infer sentences (conclusions) from given sets of sentences (premises, or axioms). Finally, it is assumed that each signature morphism leads to translations of sentences and models that preserve satisfaction and entailment. A *institution comorphism* is a translation between two institutions. It maps signatures to signatures, sentences to sentences and models to models, such that satisfaction is preserved (where models are mapped contravariantly, i.e. against the direction of the comorphism).

We refer the reader to the literature [17, 28, 36] for formal details of institutions and comorphisms. Subsequently, we use the terms ‘institution’ and ‘logic’ interchangeably, as well as the terms ‘institution comorphism’ and ‘logic translation’.

3 Implementation of a Logic

How is a single logic implemented in the Heterogeneous Tool Set? This is depicted in the left column of Fig. 1.

The syntactic entities of a logic are represented using types for *signatures* and *signature morphisms* forming a category with functions for identity morphisms and composition of morphisms as well as for extracting domains and codomains. There is also a type of *sentences* as well as a sentence translation function, allowing to translate sentences along a signature morphisms.

```

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

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

```

Fig. 2. The basic ingredients of logics and logic comorphisms

In order to model a more verbose and user-friendly input syntax of the logic we further introduce types for the abstract syntax of *basic specifications* and *symbol maps*.

Each logic has to provide *parsers* taking an input string and yielding an abstract syntax tree of either a basic specification or a symbol map. *Static analysis* takes the abstract syntax of a basic specification to a *theory* being a signature with a set of sentences. Actually, an additional parameter of the analysis, a signature called “local environment”, corresponds to imported parts of a specification and will be initially the empty signature. The static analysis also takes symbol maps (written concise and user-friendly) to signature morphisms (corresponding to mathematical objects, as part of an institution).

A theory, where some sentences are marked as axioms and others as proof goals, can be passed to a (logic-specific) *prover* which computes the entailment relation. A prover returns a proof-status answer (proved, disproved or open), together with a proof tree and further prover-specific information. The proof tree is expected to give at least the information about which axioms have been used in the proof. A *model checker* tries to construct models for a given theory, while a *conservativity checker* can check whether a theory extension is conservative (i.e. does not lead to new theorems).

Each logic is realized in the programming language Haskell [40] by a bunch of types and functions. The Haskell interface for a logic is shown in Fig. 2. We have used a simplified, stripped down version, where e.g. error handling is ignored. For technical reasons a logic is *tagged* with a unique identifier type (`lid`), which is a singleton type the only purpose of which is to determine all other type components of the given logic.

In Haskell jargon, the interface is called a multiparameter type class with functional dependencies [41]. The Haskell interface for logic translations is realised similarly.

4 Logics Available in Hets

The degree of support of different languages by HETS is shown in Fig. 3. In this section we give a short overview which logic is behind these names and we describe the purpose of each logic.

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 [14, 9]. We have implemented the CASL logic in such a way that much of the implementation can be re-used for CASL extensions as well; this is achieved via ‘holes’ (realized via polymorphic variables) in the types for signatures, morphisms, abstract syntax etc. This eases integration of CASL extensions and keeps the effort of integrating CASL extensions quite moderate.

CoCASL [38] is a coalgebraic extension of CASL, suited for the specification of process types and reactive system. The central proof method is coinduction.

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

HasCASL [45] 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.

Haskell [40] is a modern, pure and strongly typed functional programming language. It simultaneously is the implementation language of HETS, such that in the future, HETS might be applied to itself.

CspCASL [42] is a combination of CASL with the process algebra CSP.

OWL DL is the Web Ontology Language (OWL) recommended by the World Wide Web Consortium (W3C, <http://www.w3c.org>). It is used for knowledge representation and the Semantic Web [8].

CASL-DL is an extension of a restriction of CASL, realizing a strongly typed variant of OWL DL in CASL syntax.

SPASS [46] is an automatic theorem prover for first-order logic with equality.

Isabelle [39] is an interactive theorem prover for higher-order logic.

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

5 Heterogeneous Specification

Heterogeneous specification is based on some graph of logics and logic translations. The graph of currently supported logics is shown in Fig. 3. However, syntax and semantics of heterogeneous specifications as well as their implementation in HETS is parameterized over an arbitrary such logic graph. Indeed, the HETS modules implementing the logic graph can be compiled independently of the HETS modules implementing heterogeneous specification, and this separation of concerns is essential to keep the tool manageable from a software engineering point of view.

Language	Parser	Static Analysis	Prover
CASL	x	x	-
CoCASL	x	x	-
MODALCASL	x	x	-
HASCASL	x	(x)	-
Haskell	x	x	-
CSP-CASL	(x)	-	-
CASL-DL	x	-	-
OWL DL basic	x	(x)	-
OWL DL structure	x	(x)	-
SPASS	-	-	x
Isabelle	-	-	x

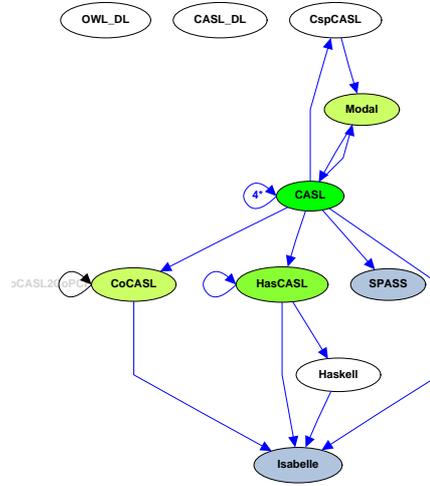


Fig. 3. Current degree of HETS support for the different languages (and x denotes a supported feature, an (x) a prototype), and graph of logics currently supported by HETS. The more an ellipse is filled, the more stable is the implementation of the logic. The arrows denote logic translations (comorphisms). Proof support can be obtained by following a path to a prover-supported logic.

Heterogeneous CASL (HETCASL; see [33]) includes the structuring constructs of CASL, such as union and translation. A key feature of CASL is that syntax and semantics of these constructs are formulated over an arbitrary institution (i.e. also for institutions that are possibly completely different from first-order logic resp. the CASL institution). HETCASL extends this with constructs for the translation of specifications along logic translations.

The syntax of heterogeneous specifications is given (in very simplified form) in Fig. 4. A specification either consists of some basic specification in some logic (which follows the specific syntax of this logic), or an extension of a specification by another one (written *SPEC then SPEC*, or, if the extension only adds theorems that are already implied by the original specification, written *SPEC then %implies SPEC*). A translation of a specification along a signature morphism is written *SPEC with SYMBOL-MAP*, where the symbol map is logic-specific (usually abbreviatory) syntax for a signature morphism. A translation along a logic comorphism is written *SPEC with logic ID*.

A specification library consists of a sequence of definitions. A definition may select the current logic (*logic ID*), which is then used for parsing and analysing the subsequent definitions. It may name a specification, and finally it may also declare a *view* between two specifications. A view is a relation between two specifications, expressing that the first specification (when translated along a signature morphism or a logic comorphism) is implied by the second specification.

It should be stressed that the name ‘HETCASL’ only refers to CASL’s structuring constructs. The individual logics used in connection with HETCASL and HETS can be completely orthogonal to CASL. Actually, the capabilities of HETS go even beyond

```

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

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

LIBRARY = DEFINITION*

```

Fig. 4. 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.

HETCASL, since HETS also supports other module systems. This enables HETS to directly read in e.g. OWL files, which use a structuring mechanism that is completely different from CASL’s. Moreover, support of further structuring languages is planned.

The semantics of HETCASL specifications is given in terms of the so-called *Grothendieck institution* [16, 31]. This institution is basically a flattening, or disjoint union, of the logic graph. A signature in the Grothendieck institution consists of a pair (L, Σ) where L is a logic and Σ is a signature in the logic L . Similarly, a Grothendieck signature morphism $(\rho, \sigma) : (L_1, \Sigma_1) \rightarrow (L_2, \Sigma_2)$ consists of a logic translation $\rho : L_1 \rightarrow L_2$ plus an L_2 -signature morphism $\sigma : \rho(\Sigma_1) \rightarrow \Sigma_2$. Sentences, models, satisfaction and entailment in the Grothendieck institution are defined in a component wise manner.

The Grothendieck logic can be implemented as a bunch of *existential* datatypes over the type class `Logic` (see Fig. 5). Usually, existential datatypes are used to realize — in a strongly typed language — heterogeneous lists, where each element may have a different type. We use lists of (components of) logics and translations instead. This leads to an implementation of the Grothendieck institution over a logic graph.

```

data G_sign = forall lid sign morphism sentence basic_spec symbol_map .
  Logic lid sign morphism sentence basic_spec symbol_map =>
  G_sign lid sign

```

Fig. 5. The Haskell implementation of signatures of the Grothendieck logic

6 Parsing and Analysis of Heterogeneous Specifications

We now explain the upper half of the right hand-side of Fig. 1. The heterogeneous parser transforms a string conforming to the syntax in Fig. 4 to an abstract syntax tree and is additionally parameterized over an arbitrary logic graph, using the `Parsec` combinator parser [24]. Logic and translation names are looked up in the logic graph — this is necessary to be able to choose the correct parser for basic specifications. Indeed, the

parser has a state that carries the current logic, and which is updated if an explicit specification of the logic is given, or if a logic translation is encountered (in the latter case, the state is set to the target logic of the translation). With this, it is possible to parse basic specifications by just using the logic-specific parser of the current logic as obtained from the state.

The static analysis is based on the static analysis of basic specifications, and transforms an abstract syntax tree to a development graph (cf. Sect. 7 below). Starting with a node corresponding to the empty theory, it successively extends (using the static analysis of basic specifications) and/or translates (along the intra- and inter-logic translations) the theory, while simultaneously adding nodes and links to the development graph. The static analysis follows a so-called verification semantics [34], which satisfies the following theorem:

Theorem 1. The development graph given by the verification semantics has the same signature and model class (in the Grothendieck institution) as the heterogeneous specification (according to the semantics of structured specifications).

7 Proof Management with Development Graphs

The central device for structured theorem proving and proof management in HETS is the formalism of *development graphs*. Development graphs have been used for large industrial-scale applications with hundreds of specifications [20]. They also support management of change [5]. The graph structure provides a direct visualization of the structure of specifications, and it also allows for managing large specifications with hundreds of sub-specifications. Tools such as MAYA [6] provide a management of proofs, based on the formalism of development graphs. The goal of HETS is to make MAYA heterogeneous.

A development graph (see Fig. 6 for an example) consists of a set of nodes (corresponding to whole structured specifications or parts thereof), and a set of arrows called *definition links*, indicating the dependency of each involved structured specification on its subparts. Each node is associated with a signature and some set of local axioms. The axioms of other nodes are inherited via *definition links*. Definition links are usually drawn as black solid arrows, denoting an import of another specification that is homogeneous (i.e. stays within the same logic). Double arrows indicate imports that are heterogeneous, i.e. the logic changes along the arrow. Technically, this is the case for Grothendieck signature morphisms (ρ, σ) where $\rho \neq id$.

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 only postulates that the local axioms of the source node hold in the target node.

Theorem links are initially displayed in red.¹ The *proof calculus* for development graphs [35, 37, 34] 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

¹ The red colour is only available when displaying the paper on a computer, or with colour printing. For the final version, we will find some transcription of colours into different greys.

proved with this calculus are drawn in green. Local theorem links can be proved by turning them into *local proof goals*. The latter can be discharged using a logic-specific calculus as given by an entailment system (see Sect. 2). 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 the following theorem [34]:

Theorem 2. The proof calculus for heterogeneous development graphs is sound and complete relative to an oracle checking conservative extensions.

8 An Example

In the following we illustrate the use of HETS with an example from the area of ontology engineering. An example involving process algebra is mentioned in the conclusion section.

Suppose we want to build up an ontology of time, i.e. an ontology of temporal entities and the relations between them. Such an ontology could be used as a basis for more complex ontologies including causal concepts, action-theoretical concepts, etc. In this context heterogeneity arises naturally, since first- and higher-order constructs are needed in order to make the relationships between the concepts of the ontology as precise as possible. In a second step, then, we could identify interesting fragments of this ontology and connect them to ontologies presented in more restricted formal frameworks (e.g., OWL DL). In particular, we could feed such fragments into ontology development tools such as Protégé (cf. <http://protege.stanford.edu/>).

Suppose that the ontology of time we are interested in contains the concepts ‘instant’, ‘interval’, ‘event’, etc. Usually, intervals are introduced in terms of instants, that is, intervals are defined as specific sets of points. However, in philosophy there has been a long debate on whether instants or rather events are ontologically prior. In the latter case, instants are to be defined in terms of events, i.e., for example, we define an instant as a meeting point of intervals that immediately equivalence classes by the meeting point of two consecutive events.

By assuming some weak conditions, the interval-from-instants approach can be shown to be equivalent to the instant-from-intervals approach, and this fact can be reproduced by heterogeneous specifications in HASCASL. To see this, we start from a CASL specification, FLOWOFTIME, characterizing flows of time as dense linear orders without endpoints. In HASCASL we can easily specify different types of intervals (open, closed, bounded, unbounded, etc.). In particular, we can provide a HASCASL specification, ALLENINTERVALS, which defines intervals as so-called *Allen intervals*, i.e. pairs of instants, (x, y) , with $x < y$.

For the instants-from-intervals approach we start with a *first-order* specification of intervals. For example, we could use a version of Allen and Hayes’ first order theory [2, 19], which characterizes intervals in terms of a single binary relation M (read as: ‘meets’ or ‘immediately precedes’). The CASL specification of this theory, ALLENHAYES, contains structural axioms such as $xMy \wedge xMu \wedge zMy \Rightarrow zMu$ (stating that places where intervals meet are unique) and axioms characterizing formal properties of the flow of events rather implicitly.

With these preparations we can state that Allen intervals define a canonical model for Allen and Hayes’ first-order theory. More precisely, each higher-order model of the theory of ALLENINTERVALS defines a first-order model of ALLENHAYES.

To construct, vice versa, points from intervals we use first an auxiliary specification CONSTRUCTPOINTSFO (for reasons that will be discussed in sec. 9), which extends ALLENHAYES by two 4-ary relations Equi and Less (cf. [22, 23]):²

```

logic CASL
spec CONSTRUCTPOINTSFO[ALLENHAYES] = %def
  preds  _ _ Equi _ _ , _ _ Less _ _ : Elem × Elem × Elem × Elem
  ∀x, y, z, u : Elem
  • x y Equi z u ⇔ x M y ∧ z M u ∧ x M u
  • x y Less z u ⇔ x M y ∧ z M u ∧ (∃v : Elem • x M v ∧ v M u)
then %implies
  ∀x, x', y, y', z, u, u', v, v', w : Elem
  • x M y ⇒ x y Equi x y                               %(Equi_refl)%
  • x y Equi z u ⇒ z u Equi x y                       %(Equi_sym)%
  • x y Equi z u ∧ z u Equi v w ⇒ x y Equi v w       %(Equi_trans)%
  • x y Equi x' y' ∧ u v Equi u' v' ⇒ (x y Less u v ⇔ x' y' Less u' v') %(Less_well_def)%
  • x M y ⇒ ¬x y Less x y                             %(Less_irrefl)%
  ...

```

In a second step this specification is applied to introduce points as equivalence classes of interval pairs with respect to the relation Equi (here read as a binary relation between interval pairs). The specification CONSTRUCTPOINTSFROMINTERVALS obtained thereby can be used to state that each model of ALLENHAYES defines a flow of time, that means, we have the following view:

```

logic HasCASL
view FLOWOFTIME_IN_CONSTRUCTPOINTSFROMINTERVALS[ALLENHAYES] :
  FLOWOFTIME to
  { CONSTRUCTPOINTSFROMINTERVALS[ALLENHAYES] then %def
    pred  _ < _ : Inst × Inst
    ∀X, Y : Inst • X < Y ⇔ ∃x, y, u, v : Elem • x M y ∧ u M v
    ∧ X = eqcl(x, y) ∧ Y = eqcl(u, v) ∧ x y Less u v }
  = sort Elem ↦ Inst

```

This small list of specifications (the development graph of them is depicted in Fig. 6) already indicates how heterogeneous specifications can be applied for the development of ontologies.

9 Theorem Proving with HETS

After parsing and static analysis of an heterogeneous specification (see Sect. 6), HETS constructs a heterogeneous development graph. This graph can be inspected, e.g. theories of nodes or signature morphisms of links can be displayed. Using the calculus mentioned in Sect. 7, the proof obligations in the graph can be (in most cases automatically) reduced to local proof goals at the individual nodes. Logics in HETS may provide a sublogic analysis which helps to get a fine-grained overview of the provers that can be applied to a particular proof goal.

The graphical user interface (GUI) for calling a prover is shown in Fig. 7. The list on the right shows all goal names prefixed with the proof status in square brackets. A

² For an explanation of the syntax of these examples, see the CASL user manual [10] and reference manual [14].

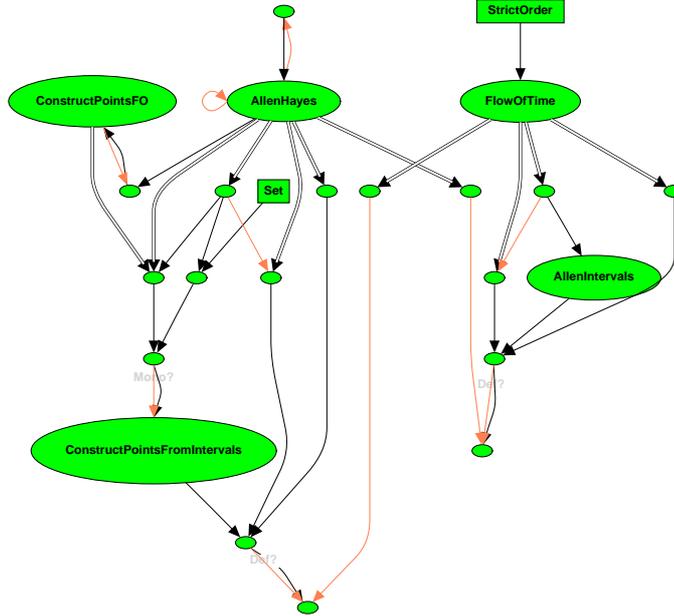


Fig. 6. The development graph of Allen, Hayes, and Ladkin’s theory of intervals. The right-hand side presents the intervals-from-instant approach, the left-hand side the instant-from-interval approach. Simple black arrows denote definition links (imports of theories), double arrows denote heterogeneous imports. Red arrows denote open theorem links (open proof obligations corresponding to interpretations of theories).

proved goal is indicated by a '+', a '-' indicates a disproved goal and a space denotes an open goal. Within this list, one can select those goals that should be inspected or proved. A button 'Display' shows the selected goals in the syntax of this theory’s logic.

The list 'Pick Theorem Prover:' lets you choose one of the connected provers. By pressing 'Prove' the selected prover is launched and the theory along with the selected goals is translated via the shortest possible path of comorphisms into the prover’s logic. The button 'More fine grained selection...' lets you pick a path of comorphisms in the logic graph that leads into a proved supported logic. It is assumed that all comorphisms are model-expansive, which means that borrowing of entailment systems along the composite comorphism $\rho = (\Phi, \alpha, \beta)$ is sound and complete [13, 34]:

Theorem 3.

$$(\Sigma, \Gamma) \models_{\Sigma}^I \varphi \text{ iff } (\Phi(\Sigma), \alpha(\Gamma)) \models^J \alpha_{\Sigma}(\varphi).$$

That is, if entailment \vdash captures semantic consequence \models , we can re-use the prover along the (composite) comorphism.

For the automatic theorem prover SPASS [46] a new window is opened which controls the prover calls (Fig. 8). Isabelle [39], a semi automatic theorem prover, is started with ProofGeneral [4] in a separate Emacs.

The 'Close' button allows for integrating the status of the goals’ list back into the development graph. If all goals have been proved, this theory’s node turns from red into green.

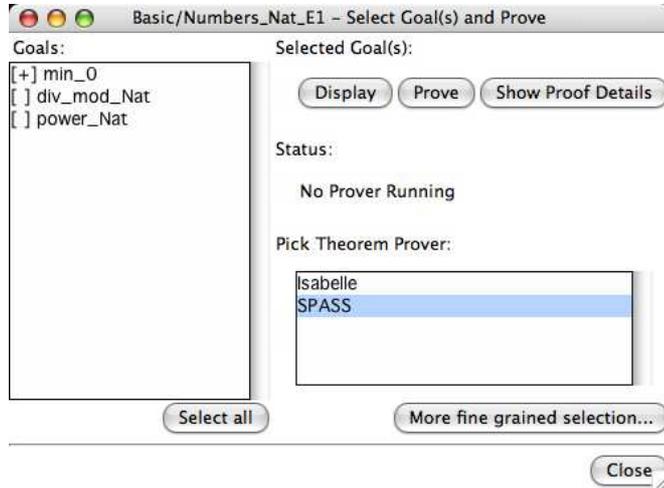


Fig. 7. Hets Goal and Prover Interface

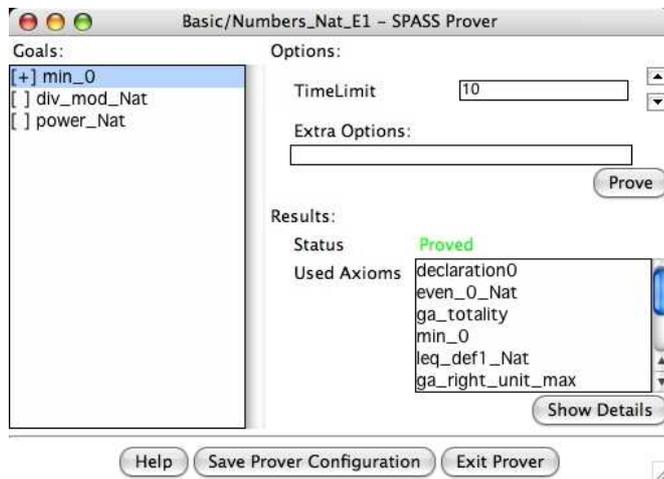


Fig. 8. Interface of the SPASS prover

For the example presented in sect. 8, after reducing the development graph in Fig. 6 using the development graph calculus, we successfully used SPASS for proving the CASL proof obligations in the nodes beyond the nodes ‘ConstructPointsFO’ and ‘AllenHayes’. To prove the proof obligations in the other nodes in the lower part of Fig. 6 the higher-order proof assistance system Isabelle was applied. The most interesting point here is that we used a first-order specification, namely CONSTRUCTPOINTSFO, to prove as much as possible by the automatic reasoner SPASS (thus minimizing the number of proof obligations to be proven by a semi-automatic reasoner). In fact, all the higher-order proof obligations arising from the view FLOWOFTIME_IN_CONSTRUCTPOINTSFROMINTERVALS (such as the well-definedness of the binary relation $<$ as well as its formal properties) can essentially be established in first order.

10 Conclusion

The Heterogeneous Tool Set is available at <http://www.tzi.de/cofi/hets>; some specification libraries and example specifications (including those of this paper) under <http://www.cofi.info/Libraries>. There, a user guide is available as well. A brief introduction into HETS is given in [10].

```

logic CSP-CASL
spec BUFFER =
  data LIST
  channels read, write : Elem
  process let Buf(l : List[Elem]) =
    read?.x → Buf(cons(x, nil))
    □ if l = nil then STOP
    else write!last(l) → Buf(rest(l))
  in Buf(nil)
  with logic → MODALCASL
  then %implies • AGF ∃x : Elem. ⟨write.x⟩ true
end

```

Fig. 9. A specification of fair buffers in CASL, CSP-CASL and MODALCASL.

There is related work about generic parsers, user interfaces, theorem provers etc. [39, 26, 25]. However, these approaches are mostly limited to *genericity*, and do not support real *heterogeneity*, that is the simultaneous use of different formalisms. Technically, genericity often is implemented with generic modules that can be instantiated many times. Here, we deal with a potentially unlimited number of such instantiations, and also with translations between them.

It may appear that HETS just provides a combination of the provers SPASS and Isabelle, and the reader may wonder what the advantage of HETS is when compared to an ad-hoc combination of Isabelle and SPASS. But already now, HETS provides proof support for modal logic (via the translation to CASL, and then further to either SPASS or Isabelle), as well as for CoCASL. Hence, it is quite easy to provide proof support for new logics by just implementing logic translations, which is at least an order of magnitude simpler than integrating a theorem prover.

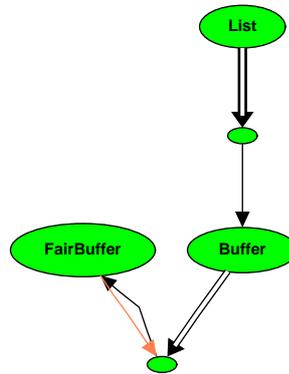


Fig. 10. Development graph for fair buffer specification in CASL, CSP-CASL and MODALCASL

Future work will integrate more logics and interface more existing theorem proving tools with specific institutions in HETS. In [32], we have presented a heterogeneous specification with more diverse formalisms, namely CSP-CASL and a temporal logic (as part of MODALCASL). An example is shown in Fig. 9. CSP-CASL is used to describe the system (a buffer implemented as a list), and some temporal logic is used to state fairness or eventuality properties that go beyond the expressiveness of the process algebra (here, we express the fairness property that the buffer cannot read infinitely often without writing). The corresponding development graph is in Fig. 10. The arrow going downwards from the node `FairBuffer` is a theorem link denoting the proof obligation that is caused by the `%implies` annotation.

In [30] we describe how to use heterogeneous specification and HETS for proving a refinement of a specification in CASL into a Haskell-program.

References

1. Modelware. <http://www.modelware-ist.org/>.
2. J. Allen and P. Hayes. A common-sense theory of time. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 528–531, Los Angeles, CA, USA, 1985.
3. Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors. *Integrated Formal Methods, Proceedings of the 1st International Conference on Integrated Formal Methods, IFM 99, York, UK, 28-29 June 1999*. Springer, 1999.
4. David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000.
5. S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an evolutionary formal software-development using CASL. In C. Choppy and D. Bert, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer-Verlag, 2000.
6. S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The development graph manager MAYA (system description). In H. Kirchner and C. Reingeissen, editors, *Algebraic Methodology and Software Technology, 2002*, volume 2422 of *Lecture Notes in Computer Science*, pages 495–502. Springer-Verlag, 2002.

7. Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors. *UML 2004 - The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings*, volume 3273 of *Lecture Notes in Computer Science*. Springer, 2004.
8. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.
9. M. Bidoit and P. D. Mosses. *CASL User Manual*, volume 2900 of *LNCS*. Springer, 2004.
10. Michel Bidoit and Peter D. Mosses. *CASL User Manual*. LNCS Vol. 2900 (IFIP Series). Springer, 2004. With chapters by Till Mossakowski, Donald Sannella, and Andrzej Tarlecki.
11. Eerke A. Boiten, John Derrick, and Graeme Smith, editors. *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*, volume 2999 of *Lecture Notes in Computer Science*. Springer, 2004.
12. Michael J. Butler, Luigia Petre, and Kaisa Sere, editors. *Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15-18, 2002, Proceedings*, volume 2335 of *Lecture Notes in Computer Science*. Springer, 2002.
13. M. Cerioli and J. Meseguer. May I borrow your logic? (transporting logical structures along maps). *Theoretical Computer Science*, 173:311–347, 1997.
14. CoFI (The Common Framework Initiative). *CASL Reference Manual*. LNCS Vol. 2960 (IFIP Series). Springer, 2004.
15. Louise A. Dennis, Graham Collins, Michael Norrish, Richard J. Boulton, Konrad Slind, and Thomas F. Melham. The prosper toolkit. *STTT*, 4(2):189–210, 2003.
16. R. Diaconescu. Grothendieck institutions. *Applied categorical structures*, 10:383–402, 2002.
17. 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.
18. Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors. *Integrated Formal Methods, Second International Conference, IFM 2000, Dagstuhl Castle, Germany, November 1-3, 2000, Proceedings*, volume 1945 of *Lecture Notes in Computer Science*. Springer, 2000.
19. Patrick J. Hayes and James F. Allen. Short time periods. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-87)*, 1987.
20. Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, and Wolpers Wolpers. Verification support environment (VSE). *High Integrity Systems*, 1(6):523–530, 1996.
21. Michael Kohlbase. OMDOC: An infrastructure for OPENMATH content dictionary information. *Bulletin of the ACM Special Interest Group on Symbolic and Automated Mathematics (SIGSAM)*, 34(2):43–48, 2000.
22. P. Ladkin. Models of axioms for time intervals. In *Proceedings of AAAI-87*, pages 234–239, 1987.
23. Peter B. Ladkin. The completeness of a natural system for reasoning with time intervals. In *Proceedings of IJCAI-87*, pages 462–467, 1987.
24. Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report. UU-CS-2001-35.
25. C. L'uth, H. Tej, Kolyang, and B. Krieg-Brückner. TAS and IsaWin: Tools for transformational program development and theorem proving. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering FASE'99. Joint European Conferences on Theory and Practice of Software ETAPS'99*, number 1577 in LNCS, pages 239–243. Springer Verlag, 1999.
26. C. L'uth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2):167–189, March 1999.
27. Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. jeti: A tool for remote tool integration. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part*

- of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, *Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 557–562. Springer, 2005.
28. J. Meseguer. General logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989.
 29. Jos e Meseguer. Formal interoperability. In *Proceedings of the 1998 Conference on Mathematics in Artificial Intelligence*, Fort Lauderdale, Florida, January 1998. <http://rutcor.rutgers.edu/~amai/Proceedings.html>. Presented also at the 14th IMACS World Congress, Atlanta, Georgia, July 1994.
 30. T. Mossakowski. Institutional 2-cells and Grothendieck institutions. Submitted for publication.
 31. T. Mossakowski. Comorphism-based Grothendieck logics. In K. Diks and W. Rytter, editors, *Mathematical foundations of computer science*, volume 2420 of *LNCS*, pages 593–604. Springer, 2002.
 32. T. Mossakowski. Foundations of heterogeneous specification. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755, pages 359–375. Springer, 2003.
 33. T. Mossakowski. HetCASL - heterogeneous specification. language summary, 2004.
 34. T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Habilitation thesis, University of Bremen, 2005.
 35. T. Mossakowski, S. Autexier, and D. Hutter. Development graphs – proof management for structured specifications. *Journal of Logic and Algebraic Programming*. to appear.
 36. Till Mossakowski, Joseph Goguen, Razvan Diaconescu, and Andrzej Tarlecki. What is a logic? In Jean-Yves Beziau, editor, *Logica Universalis*, pages 113–133. Birkh user, 2005.
 37. Till Mossakowski, Piotr Hoffman, Serge Autexier, and Dieter Hutter. CASL logic. In Peter D. Mosses, editor, *CASL Reference Manual*, volume 2960 of *Lecture Notes in Computer Science*, part IV. Springer Verlag, London, 2004. Edited by T. Mossakowski.
 38. Till Mossakowski, Lutz Schr oder, Markus Roggenbach, and Horst Reichel. Algebraic-co-algebraic specification in CoCASL. *Journal of Logic and Algebraic Programming*. To appear.
 39. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002.
 40. S. Peyton-Jones, editor. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge, 2003. also: *J. Funct. Programming* **13** (2003).
 41. Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Haskell Workshop*. 1997.
 42. Markus Roggenbach. CSP-CASL – A new integration of process algebra and algebraic specification. In A. Nijholt and G. Scollo, editors, *AMiLP-3 – Third AMAST Workshop on Algebraic Methods in Language Processing*. TWLT series, University of Twente, 2003. Long version to appear in *Theoretical Computer Science*.
 43. Grigore Rosu and Joseph Goguen. Composing hidden information modules over inclusive institutions, 2004.
 44. Klaus Schneider. *Verification of Reactive Systems*. Springer Verlag, 2004.
 45. L. Schr oder and T. Mossakowski. HasCASL: Towards integrated specification and development of Haskell programs. In H. Kirchner and C. Reingeissen, editors, *Algebraic Methodology and Software Technology, 2002*, volume 2422 of *Lecture Notes in Computer Science*, pages 99–116. Springer-Verlag, 2002.
 46. 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.
 47. J rgen Zimmer and Michael Kohlhase. System description: The mathweb software bus for distributed mathematical reasoning. In Andrei Voronkov, editor, *18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 139–143. Springer, 2002.