The Heterogeneous Tool Set

Till Mossakowski[1], Christian Maeder[1], and Klaus Lüttich[2]

[1] DFKI Lab Bremen and Department of Computer Science, University of Bremen, Germany
[2] SFB/TR 8 and Department of Computer Science, University of Bremen, 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 specifications, 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.
We illustrate the approach with a sample heterogeneous proof proving the correctness of the composition table of a qualitative spatial calculus. The proof involves two different provers and logics: an automated first-order prover solving the vast majority of the goals, and an interactive higher-order prover used to prove a few bridge lemmas.

1 Introduction

> "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.*" [43] (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/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. [43]). 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 [3] 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 would be represented as logic translations.) Likewise, languages for mathematical knowledge management like OpenMath and OMDoc [18] are deliberately only semi-formal. Service integration approaches like MathWeb [48] 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 [41]. Integrations of multiple decision procedures and model checkers into theorem provers, like in the PROSPER toolkit [9], 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 (HETS) 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. This can be compared with the treatment of *theory morphisms* as first-class citizens, which is a distinctive feature of formalisms like OMDoc [18] and tools like Specware [17] and IMPS [12, 11]. A clear referencing of symbols to their theories can distinguish, for example, the naturals with zero from the naturals without zero, even if they are denoted with the same symbol *Nat*. Theory morphisms can relate the two different theories of naturals. In HETS, both theory morphisms and logic comorphisms are first-class citizens. This means that HETS can also distinguish conjunction in Isabelle/HOL from conjunction in PVS[3] (these actually have two different semantics!) and relate the underlying logics with a comorphism.

[3] At least once a logic for PVS has been added.

The architecture of the heterogeneous tool set is shown in Fig. 2 on page 123. In the sequel, we will explain the details of this figure.

2 Heterogeneous Specifications: the Model-Theoretic View

We take a model-theoretic view on specifications [42]. This means that the notion of logical theory (i.e. collection of axioms) is considered to be only an auxiliary concept, and the meaning of a formal specification (of a program module) is given by

- its signature; listing the names of entities that need to be implemented, typically together with their types, that is, the *syntactic interface* of the module, and
- its class of models, that is, the set of possible *realizations* or implementations of the interface.

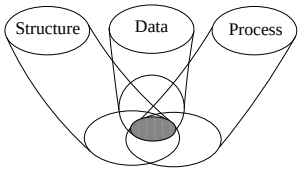

Fig. 1: Multiple viewpoints

This model-theoretic view is even more important when moving from homogeneous to heterogeneous specifications: in general, one cannot expect that different formalisms (say, a specification and a programming language, or a process algebra and a temporal logic) are related by translating theories — it is the *models* that are used to link different formalisms. This point of view is also expressed by the so-called *viewpoint specifications* (see Fig. 1), which use logical theories in different logical formalisms in order to restrict the model class of an overall system from different viewpoints (while a direct specification of the model class of the overall system would become unmanageably complex).

The correct mathematical underpinnings to this are given by the theory of *institutions* [14]. Institutions capture in a very abstract and flexible way the notion of a logical system, by leaving open the details of signatures, models, sentences (axioms) and satisfaction (of sentences in models). The only condition governing the behaviour of institutions is the *satisfaction condition*, stating that *truth is invariant under change of notation* (or enlargement of context):

$$M' \models_{\Sigma'} \sigma(\varphi) \Leftrightarrow M'|_\sigma \models_\Sigma \varphi$$

Here, $\sigma\colon \Sigma \longrightarrow \Sigma'$ is a *signature morphism*, relating different signatures (or module interfaces), $\sigma(\varphi)$ is the translation of the Σ-sentence φ along σ, and $M'|_\sigma$ is the reduction of the Σ'-model M' to a Σ-model.

The importance of the notion of institutions lies in the fact that a whole body of specification theory (concerning structuring of specifications, module concepts, parameterization, implementation, refinement, development, proof calculi) can be developed independently of the underlying institutions — all that is needed is captured by the satisfaction condition.

Different logical formalisms are related by *institution comorphisms* [13], which are again governed by the satisfaction condition, this time expressing that truth is invariant also under change of notation across different logical formalisms:

$$M' \models^{J}_{\Phi(\Sigma)} \alpha_\Sigma(\varphi) \Leftrightarrow \beta_\Sigma(M') \models^{I}_{\Sigma} \varphi.$$

Here, $\Phi(\Sigma)$ is the translation of signature Σ from institution I to institution J, $\alpha_\Sigma(\varphi)$ is the translation of the Σ-sentence φ to a $\Phi(\Sigma)$-sentence, and $\beta_\Sigma(M')$ is the translation (or perhaps: reduction) of the $\Phi(\Sigma)$-model M' to a Σ-model.

Heterogeneous specification is based on some graph of logics and logic translations, formalized as institutions and comorphisms. The so-called *Grothendieck institution* [10, 24] is a technical device for giving a semantics to heterogeneous specifications. 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 (institution) and Σ is a signature in the logic L. Similarly, a Grothendieck signature morphism $(\rho, \sigma) : (L_1, \Sigma_1) \to (L_2, \Sigma_2)$ consists of a logic translation $\rho = (\Phi, \alpha, \beta) \colon L_1 \longrightarrow L_2$ plus an L_2-signature morphism $\sigma \colon \Phi(\Sigma_1) \longrightarrow \Sigma_2$. Sentences, models and satisfaction in the Grothendieck institution are defined in a component wise manner.

The Grothendieck institution can be understood as a flat combination of all of the involved logics. Here, "flat" means that there is no direct interaction of e.g. logical connectives from different logics that lead to new sentences; instead, just the disjoint union of sentences is taken. However, this does not mean that the logics just coexist without any interaction: they interact through the comorphisms. Comorphisms allow for translating a logical theory into some other logic, and via this translation to interact with theories in that logic (e.g. by expressing some refinement relation).

We refer the reader to the literature [14, 13, 23, 30] for full 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. 2.

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 for translation of sentences along a signature morphisms.

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*.

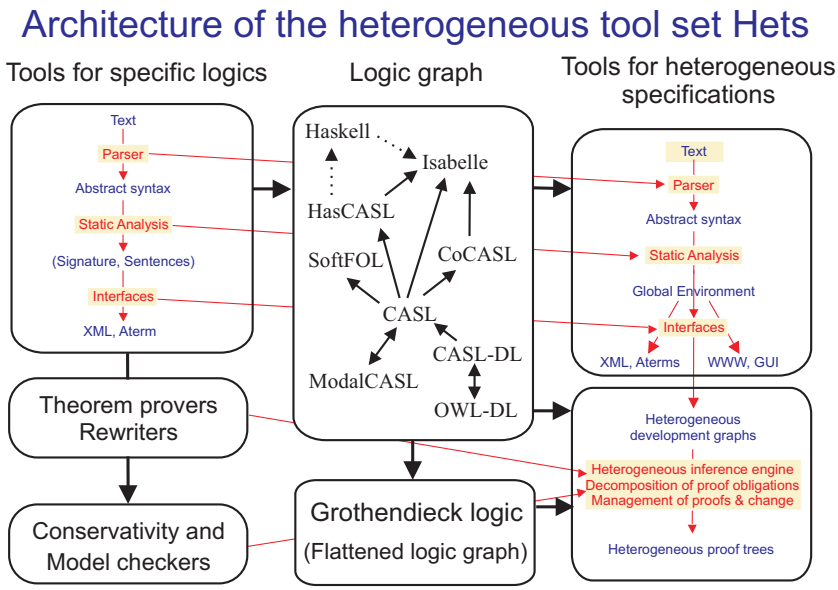

Fig. 2. Architecture of the heterogeneous tool set

```
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]
```

Fig. 3. The basic ingredients of logics

Each logic has to provide *parsers* taking an input string and yielding an abstract syntax tree of either a basic specifications 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).

Models are usually mathematical objects, often infinite, and hence usually not directly represented as syntactical objects. Still, usually it is possible to represent all finite models and some of the infinite models finitely. We assume that there is a syntactically recognizable subset of *constructive* specifications that are guaranteed to have a model, and use these as descriptions for models.[4] We do not require that a constructive specification has exactly one model; this covers cases where a uniqueness property would be achievable only with additional effort (such as recursive function definitions). A *model checker* evaluates whether a formula holds in a given model, or more precisely, in all models of a constructive specification.

Proof theory, more specifically, derivability of sentences from other sentences, is captured by the notion of *entailment system* [23]. In the HETS interface for logics, this is realized as follows. 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 finder* 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 [35] by a set of types and functions, see Fig. 3, where we present a simplified, stripped down version, where e.g. error handling is ignored. For technical reasons a logic is *tagged* with a unique identifier type (`lid`), which is a singleton type the only purpose of which is to determine all other type components of the given logic. In Haskell jargon, the interface is called a multiparameter type class with functional dependencies [36]. The Haskell interface for logic translations is realised similarly.

4 Logics Available in Hets

In this section we give a short overview of the logics available in HETS.

Propositional is classical propositional logic, with the zChaff SAT solver [15] connected to it.

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 [8, 6]. 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.

[4] If necessary, one can always extend the logic with new sentences leading to constructive specifications.

CoCASL [33] is a coalgebraic extension of CASL, suited for the specification of process types and reactive systems. 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 [44] 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 to be embedded in the specification.

Haskell [35] 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.

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 [5].

CASL-DL [20] is an extension of a restriction of CASL, realizing a strongly typed variant of OWL DL in CASL syntax.

SoftFOL [21] offers three automated theorem proving (ATP) systems for first-order logic with equality: (1) SPASS [45]; (2) Vampire [39]; and (3) MathServ Broker[5] [47]. These together comprise some of the most advanced theorem provers for first-order logic.

Isabelle [34] is an interactive theorem prover for higher-order logic, and (jointly with others) marks the frontier of current research in interactive higher-order provers.

Propositional, SoftFOL 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. 2. 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.

Heterogeneous CASL (HETCASL; see [26]) 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.

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

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

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

LIBRARY = DEFINITION*
```

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.

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 kind of refinement 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. Indeed, using the heterogeneous language constructs (including the possibility to add new logic translations involving e.g. behavioural quotient constructions) it is possible to capture a large variety of different refinement notions just by heterogeneous views as explained above.

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 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 Grothendieck logic (see Sect. 2), which is the semantic basis of HETCASL, can be implemented as a bunch of *existential* datatypes over the type class `Logic`. 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.

6 Parsing and Analysis of Heterogeneous Specifications

Based on the type class `Logic`, a number of logics and various comorphisms among these have been implemented for HETS. We now come to the logic-independent modules in HETS, which can be found in the right half of Fig. 2. These modules comprise roughly one third of HETS' 100.000 lines of Haskell code.

The heterogeneous parser transforms a string conforming to the syntax in Fig. 4 to an abstract syntax tree, using the `Parsec` combinator parser [19]. 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.

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 [16]. They also support management of change. 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.

A development graph (see Fig. 7 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.

Complementary to definition links, which *define* the theories of related nodes, *theorem links* serve for *postulating* relations between different theories. Theorem links

are the central data structure to represent proof obligations arising in formal developments. Theorem links can be *global* (drawn as solid arrows) or *local* (drawn as dashed arrows): a global theorem link postulates that all axioms of the source node (including the inherited ones) hold in the target node, while a local theorem link only postulates that the local axioms of the source node hold in the target node.

Both definition and theorem links can be *homogeneous*, i.e. stay within the same logic, or *heterogeneous*, i.e. the logic changes along the arrow. Technically, this is the case for Grothendieck signature morphisms (ρ, σ) where $\rho \neq id$. This case is indicated with double arrows.

Theorem links are initially displayed in red in the tool. (In Fig. 7, they are displayed using thin lines and non-filled arrow heads.) The *proof calculus* for development graphs [28, 31, 27] is given by rules that allow for proving global theorem links by decomposing them into simpler (local and global) ones. Theorem links that have been proved with this calculus are drawn in green. Local theorem links can be proved by turning them into *local proof goals*. The latter can be discharged using a logic-specific calculus as given by an entailment system (see Sect. 3). Open local proof goals are indicated by marking the corresponding node in the development graph as red; if all local implications are proved, the node is turned into green. This implementation ultimately is based on a theorem [27] stating soundness and relative completeness of the proof calculus for heterogeneous development graphs.

While the semantics of theorem links is explained in entirely model-theoretic terms, theorem links can ultimately be reduced to local proof obligations (and conservativity checks) of a proof-theoretic nature, amenable to machine implementation. Note however, that this approach is quite different from that of logical frameworks. Suppose that we have a global theorem link $\sigma\colon N_1 \longrightarrow N_2$ between two nodes N_1 and N_2 in a development graph. Note that the logics of N_1 and N_2 may be different. The logical framework approach assumes that the theories of N_1 and N_2 are encoded into some logic that is fixed once and forall. By contrast, in HETS we can rather flexibly find a logic that is a "common upper bound" of the logics of both N_1 and N_2 and that moreover has best possible tool support. This freedom allows us to exploit specialized tools. This is also complemented by a sublogic analysis, which is required for each of the logics in HETS, and which allows for an even more fine-grained determination of available tools.

8 An Example

In the domain of qualitative constraint reasoning, a subfield of AI which has evolved in the past 25 years, a large number of calculi for efficient reasoning about spatial and temporal entities have been developed. A prominent example of that kind are the various region connection calculi [38]. In the region connection calculus RCC8, which also has become a GIS standard, it is possible to express relations between regions (= regular closed sets) in a metric space. The set of RCC8 base relations consists

of the relations DC ("DisConnected"), EC ("Externally Connected"), PO ("Partially Overlap"), TPP ("Tangential Proper Part"), NTPP ("Non-Tangential Proper Part"), the converses of the latter two relations (TPPi and NTPPi, resp.) and EQ ("EQals") (see Fig. 5 for a pictorial representation). The RCC5 calculus is similar, but does not distinguish between tangential and non-tangential parts; it hence has only 5 basic relations.

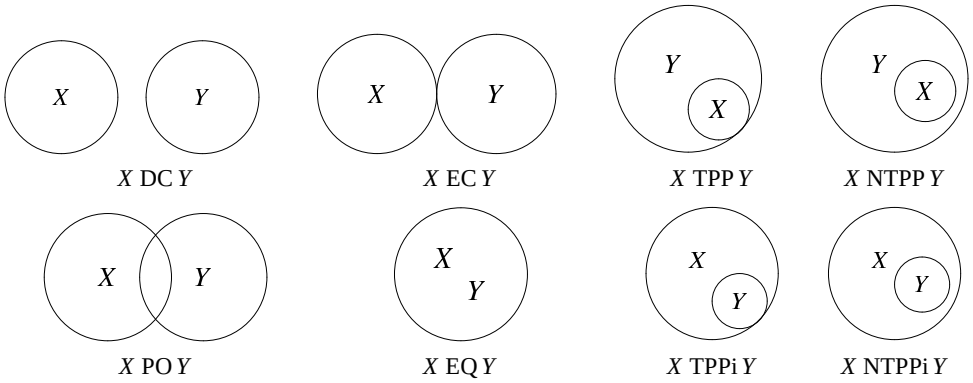

Fig. 5. The RCC-8 relations

For efficiency and feasibility reasons, qualitative spatial and temporal reasoning is not directly done in a (typically infinite) metric space, but rather at the abstract level of a (finite) relation algebra, for example, using the so-called path consistency algorithm. The heart of this approach is the composition table, which captures composition of relations at the abstract and finitary level of the relation algebra.

Composition tables need to be set up only once and for all. Still, this process is error-prone, and we already have found errors in published composition tables. Hence, formal verification of composition tables (w.r.t. their semantic interpretation) is an important task. In [46], we present a heterogeneous verification of the RCC8 composition table w.r.t. the interpretation in metric spaces. This verification goal can be split into two subgoals:

1. Verification that closed discs in a metric (cf. node `RCC_FO` in Fig. 7) satisfy some of Bennett's connectedness axioms [4] (cf. node `MetricSpace` in Fig. 7). `RCC_FO` consists of very *few* (actually, 4) theorems, so-called *bridge lemmas*. Since `MetricSpace` is a higher-order theory, they need to be translated to higher-order logic, and can then be proved using the *interactive* theorem prover Isabelle.
2. Verification that Bennett's connectedness axioms imply the standard RCC axioms (cf. nodes `ExtRCCByRCC5Rels` and `ExtRCCByRCC8Rels` in Fig. 7). The latter are *many* (actually, 95) first-order theorems, and can be proved using the *automated* theorem proving system SPASS.

```
view RCC_FO_IN_METRICSPACE :
    RCC_FO to
    {EXTMETRICSPACEBYCLOSEDBALLS[METRICSPACE]
     then %def
          pred __C__ : ClosedBall × ClosedBall;
               nonempty(x : ClosedBall) ⇔ x C x
          ∀ x, y : ClosedBall
          • x C y ⇔ ∃ s : S • rep x s ∧ rep y s                    %(C_def)%
    } =
    QReg ↦ ClosedBall
end
```

Fig. 6. Specification of a heterogeneous refinement expressing correctness of the RCC8 composition table

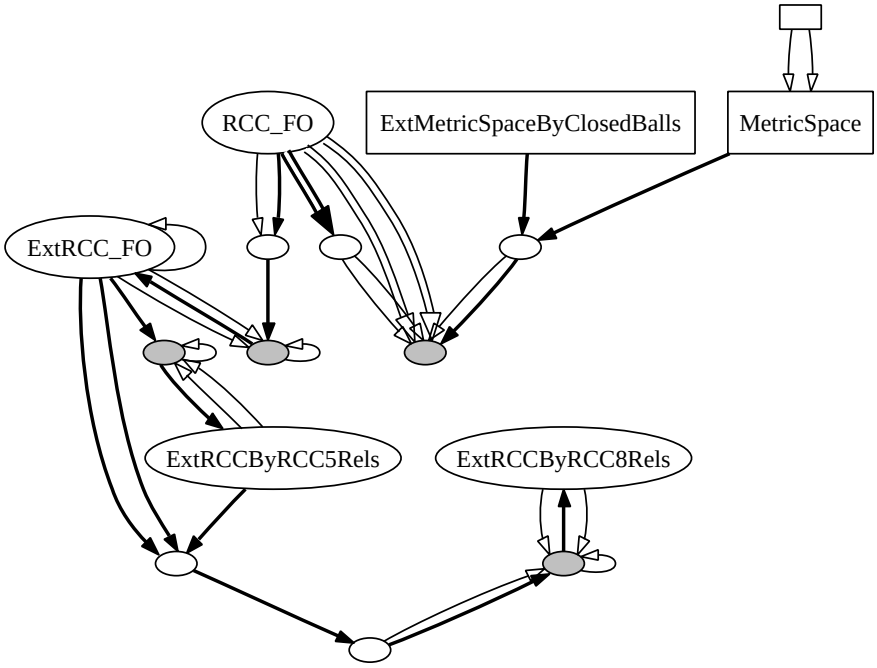

Fig. 7. Development graph for correctness proof of RCC8 composition table in CASL and HASCASL

9 Theorem Proving with HETS

Fig. 6 contains the heterogeneous refinement expressing the correctness of the RCC8 composition table. After parsing and static analysis of an heterogeneous specification (see Sect. 6), HETS constructs a heterogeneous development graph, see Fig. 7. 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. Nodes with local proof goals are marked with a grey color in Fig. 7, while in the tool, red is used. The thick edges in the development graph are definition links and the thin ones are theorem links. A double arrow denotes a heterogeneous link (e.g.

between RCC_FO and the extension of EXTMETRICSPACEBYCLOSEDBALLS). Unnamed nodes show intermediate structuring of specifications and box-shaped nodes are imported from a different specification library, while the round nodes are theories specified locally.

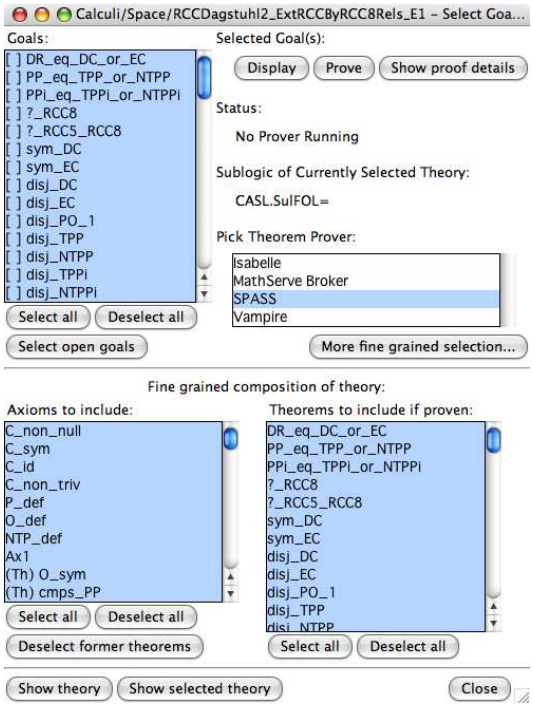

Fig. 8. Hets Goal and Prover Interface

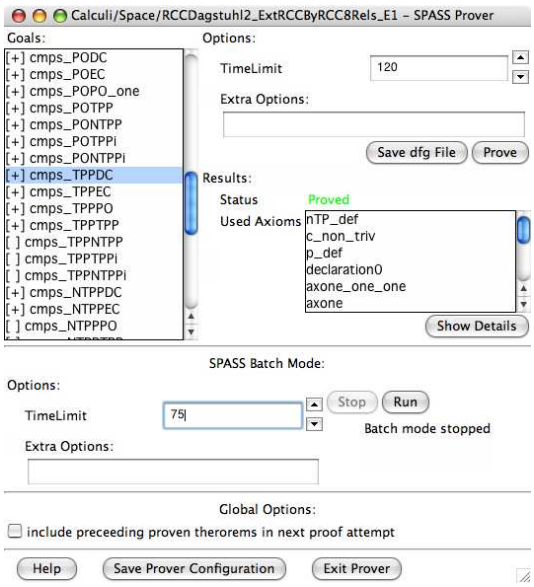

Fig. 9. Interface of the SPASS prover

The graphical user interface (GUI) for calling a prover is shown in Fig. 8. The list on the left shows all goal names prefixed with the proof status in square brackets. A proved goal is indicated by a '+', a '-' indicates a disproved goal 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. However, the shortest path need not always be the best one. Therefore, the button 'More fine grained selection...' lets you pick a specific path of comorphisms in the logic graph that leads into a prover 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 [7, 27]:

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

That is, if the entailment $\vdash$ generated by the prover captures semantic consequence $\models$, we can re-use the prover along the (composite) comorphism. In the terminology of [1], $(\Sigma, \Gamma) \models^I_\Sigma \varphi$ in institution I captures the *what to prove*, while its translation to institution J captures the *how to prove*.

Additionally, this interface offers to select in detail the axioms and proven theorems which are included in the theory for the next proof attempt. Among the axioms theorems imported from other specifications are marked with the prefix '(Th)'. This is particularly useful for large theories with problematic theorems that blow up the search space of ATP systems. A detailed discussion of using ATPs for CASL can be found in [21].

If an ATP is selected, a new window is opened, which controls the prover calls (Fig. 9). Here we use the connection to SPASS [45], for the other ATPs listed (MathServ Broker and Vampire) see [21]. Isabelle [34], a semi automatic theorem prover, is started with ProofGeneral [2] in a separate Emacs from the GUI.

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.

For the example presented in Sect. 8 we successfully used SPASS for proving the CASL proof obligations in the unnamed grey nodes between the nodes "RCC_FO" and "ExtRCC_FO" and below "ExtRCC_FO". To discharge the proof obligations in the node below "RCC_FO" with incoming heterogeneous theorem links on the right of the center of Fig. 7 the higher-order proof assistance system Isabelle was applied. The most interesting point here is that we used a first-order specification, namely RCC_FO, to prove as much as possible by the ATP SPASS (thus minimizing the number of proof obligations to be proven by a semi-automatic reasoner).

10 Conclusion

The Heterogeneous Tool Set is available at `http://www.dfki.de/sks/hets`; some specification libraries and example specifications (including those of this paper) under `http://www.cofi.info/Libraries`. A user guide is also available there. Brief introductions into HETS are given in [32] and [6].

There is related work about generic parsers, user interfaces, theorem provers etc. [34, 2]. 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.

```
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. 10. A specification of fair buffers in CASL, CSP-CASL and MODALCASL.

It may appear that HETS just provides a combination of some first-order provers and Isabelle, and the reader may wonder what the advantage of HETS is when compared to an ad-hoc combination of Isabelle and such provers, like [22]. 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. Although this can be compared to embedding the new logic in a HOL prover, our translational approach has the major advantage that several translations may exist in parallel (think of the standard and functional translations of modal logic), and the best one may be chosen depending on the theory at hand.

Future work will integrate more logics and interface more existing theorem proving tools with specific institutions in HETS. In [25], we have presented a heterogeneous specification with more diverse formalisms, namely CSP-CASL [40] (a combination of CASL with the process algebra CSP), and a temporal logic (as part of MODALCASL). An example is shown in Fig. 10. 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).

In [29] we describe how heterogeneous specification and HETS could be used for proving a refinement of a specification in CASL into a Haskell-program. Another challenge is the integration of proof planners into HETS. Finally, there is work in progress about the meta-level specification of institutions and their comorphisms in Twelf [37], which shall lead to correctness proofs for the comorphisms integrated into HETS.

Acknowledgement

This work has been supported by the project MULTIPLE of the *Deutsche Forschungsgemeinschaft* under grant KR 1191/5-2. We thank Katja Abu-Dib, Mihai Codescu,

Carsten Fischer, Jorina Freya Gerken, Rainer Grabbe, Sonja Gröning, Daniel Hausmann, Wiebke Herding, Hendrik Iben, Cui "Ken" Jian, Heng Jiang, Anton Kirilov, Tina Krausser, Martin Kühl, Mingyi Liu, Dominik Lücke, Maciek Makowski, Immanuel Normann, Razvan Pascanu, Daniel Pratsch, Felix Reckers, Markus Roggenbach, Pascal Schmidt, Lutz Schröder, Paolo Torrini, René Wagner, Jian Chun Wang and Thiemo Wiedemeyer for help with the implementation of HETS, and Erwin R. Catesbeiana for testing the consistency checker.

References

1. Jean-Raymond Abrial and Dominique Cansell. Click'n prove: Interactive proofs within set theory. In David A. Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2003.
2. David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, LNCS 1785, pages 38–42. Springer, 2000.
3. Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors. *UML 2004*, LNCS 3273. Springer, 2004.
4. B. Bennett. *Logical Representations for Automated Reasoning about Spatial Relationships*. PhD thesis, School of Computer Studies, The University of Leeds, 1997.
5. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.
6. M. Bidoit and P. D. Mosses. CASL *User Manual*, volume 2900 of *LNCS*. Springer, 2004.
7. M. Cerioli and J. Meseguer. May I borrow your logic? (transporting logical structures along maps). *Theoretical Computer Science*, 173:311–347, 1997.
8. CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.
9. 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.
10. R. Diaconescu. Grothendieck institutions. *Applied categorical structures*, 10:383–402, 2002.
11. William M. Farmer. An infrastructure for intertheory reasoning. In *Automated Deduction - CADE-17*, LNCS 1831, pages 115–131. Springer, 2000.
12. William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, 1993.
13. J. Goguen and G. Roşu. Institution morphisms. *Formal aspects of computing*, 13:274–307, 2002.
14. 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.
15. Marc Herbstritt. zChaff: Modifications and extensions. report00188, Institut für Informatik, Universität Freiburg, July 17 2003. Thu, 17 Jul 2003 17:11:37 GET.
16. 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.
17. Kestrel Development Corporation. Specware 4.1 language manual. `http://www.specware.org/`.
18. Michael Kohlhase. *OMDoc - An Open Markup Format for Mathematical Documents [version 1.2]*. LNCS 4180. Springer, 2006.
19. Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report. UU-CS-2001-35.
20. K. Lüttich, T. Mossakowski, and B. Krieg-Brückner. Ontologies for the Semantic Web in CASL. In José Fiadeiro, editor, *WADT 2004*, LNCS 3423, pages 106–125. Springer, 2005.
21. Klaus Lüttich and Till Mossakowski. Reasoning Support for CASL with Automated Theorem Proving Systems. WADT 2006, Springer LNCS, to appear.
22. Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Inf. Comput*, 204(10):1575–1596, 2006.

23. J. Meseguer. General logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989.
24. T. Mossakowski. Comorphism-based Grothendieck logics. In K. Diks and W. Rytter, editors, *MFCS*, LNCS 2420, pages 593–604. Springer, 2002.
25. T. Mossakowski. Foundations of heterogeneous specification. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *WADT 2002*, LNCS Vol. 2755, pages 359–375. Springer, 2003.
26. T. Mossakowski. HetCASL - heterogeneous specification. language summary, 2004.
27. T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Habilitation thesis, University of Bremen, 2005.
28. T. Mossakowski, S. Autexier, and D. Hutter. Development graphs – proof management for structured specifications. *Journal of Logic and Algebraic Programming*, 67(1-2):114–145, 2006.
29. Till Mossakowski. Institutional 2-cells and Grothendieck institutions. In K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning and Computation. Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, LNCS 4060, pages 124–149. Springer, 2006.
30. Till Mossakowski, Joseph Goguen, Razvan Diaconescu, and Andrzej Tarlecki. What is a logic? In Jean-Yves Beziau, editor, *Logica Universalis*, pages 113–133. Birkhäuser, 2005.
31. Till Mossakowski, Piotr Hoffman, Serge Autexier, and Dieter Hutter. CASL logic. In Peter D. Mosses, editor, *CASL Reference Manual*, LNCS 2960, part IV. Springer Verlag, 2004.
32. Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Orna Grumberg and Michael Huth, editors, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer-Verlag Heidelberg, 2007.
33. Till Mossakowski, Lutz Schröder, Markus Roggenbach, and Horst Reichel. Algebraic-co-algebraic specification in CoCASL. *Journal of Logic and Algebraic Programming*, 67(1-2):146–197, 2006.
34. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002.
35. S. Peyton-Jones, editor. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge, 2003. also: J. Funct. Programming **13** (2003).
36. Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Haskell Workshop*. 1997.
37. Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. pages 202–206.
38. D. A. Randell, Z. Cui, and A. G. Cohn. A spatial logic based on regions and connection. In B. Nebel, W. Swartout, and C. Rich, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the 3rd International Conference (KR-92)*, pages 165–176. Morgan Kaufmann, 1992.
39. Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.
40. Markus Roggenbach. Csp-casl - a new integration of process algebra and algebraic specification. *Theor. Comput. Sci.*, 354(1):42–71, 2006.
41. Judi Romijn, Graeme Smith, and Jaco van de Pol, editors. *Integrated Formal Methods, 5th International Conference, IFM 2005, Eindhoven, The Netherlands, November 29 - December 2, 2005, Proceedings*, volume 3771 of *Lecture Notes in Computer Science*. Springer, 2005.
42. Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.
43. Klaus Schneider. *Verification of Reactive Systems*. Springer Verlag, 2004.
44. L. Schröder and T. Mossakowski. HasCASL: Towards integrated specification and development of Haskell programs. In H. Kirchner and C. Reingeissen, editors, *AMAST, 2002*, LNCS 2422, pages 99–116. Springer, 2002.
45. 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*, LNCS 2392, pages 275–279. Springer-Verlag, 2002.
46. Stefan Wölfl, Till Mossakowski, and Lutz Schröder. Qualitative constraint calculi: Heterogeneous verification of composition tables. In *20th International FLAIRS Conference*, 2007.
47. Jürgen Zimmer and Serge Autexier. The MathServe System for Semantic Web Reasoning Services. In U. Furbach and N. Shankar, editors, *3rd IJCAR*, LNCS 4130. Springer, 2006.
48. Jürgen Zimmer and Michael Kohlhase. System description: The mathweb software bus for distributed mathematical reasoning. In Andrei Voronkov, editor, *18th CADE*, LNCS 2392, pages 139–143. Springer, 2002.