Checking Conservativity With HETS*

Mihai Codescu[1], Till Mossakowski[2,3], and Christian Maeder[2]

[1] University of Erlangen-Nürnberg, Germany
[2] DFKI GmbH Bremen, Germany
[3] SFB/TR 8 "Spatial Cognition", University of Bremen, Germany

1 Introduction

Conservative extension is an important notion in the theory of formal specification [8]. If we can implement a specification *SP*, we can implement any conservative extension of *SP* as well. Hence, a specification can be shown consistent by starting with a consistent specification and extending it using a number of conservative extension steps. This is important, because during a formal development, it is desirable to guarantee consistency of specifications as soon as possible. Checks for conservative extensions also arise in calculi for proofs in structured specifications [12, 9]. Furthermore, consistency is a special case of conservativity: it is just conservativity over the empty specification. Moreover, using consistency, also *non-consequence* can be checked: an axiom does *not* follow from a specification if the specification augmented by the negation of the axiom is consistent. Finally, [3] puts forward the idea of simplifying the task of checking consistency of large theories by decomposing them with the help of an architectural specification [2]. In order to show that an architectural specification is consistent, it is necessary to show that a number of extensions are conservative (more precisely, the specifications of its generic units need to be conservative extensions of their argument specifications, and those of the non-generic units need to be consistent).

In this paper we present a (sound, but incomplete) algorithm for deciding conservativity of extensions of specifications in CASL [4] as part of the Heterogeneous Tool Set (HETS) [10], available at `http://hets.dfki.de`.

2 Conservative Extensions in CASL

CASL [4] extends many-sorted first-order logic with partial functions and subsorting. It also provides induction sentences, expressing the (free) generation of datatypes.

CASL signatures consist of a set S of sorts with a subsort relation $\leq$ between them together with families $\{PF_{w,s}\}_{w\in S^*,s\in S}$ of partial functions, $\{TF_{w,s}\}_{w\in S^*,s\in S}$ of total functions and $\{P_w\}_{w\in S^*}$ of predicate symbols. Signature morphisms consist of maps taking sort, function and predicate symbols respectively to a symbol of the same kind in the target signature, and they must preserve subsorting, typing of function and predicate symbols and totality of function symbols.

For a signature Σ, terms are formed starting with variables from a sorted set X using applications of function symbols to terms of appropriate sorts, while sentences

* This work has been supported by the BMBF project SHIP.

are partial first-order formulas extended with *sort generation constraints* which are triples (S', F', σ') such that $\sigma' : \Sigma' \to \Sigma$ and S' and F' are respectively sort and function symbols of Σ'. Partial first-order formulas are translated along a signature morphism $\varphi : \Sigma \to \Sigma''$ by replacing symbols as prescribed by φ while sort generation constraints are translated by composing the morphism σ' in their third component with φ.

Models interpret sorts as sets such that subsorts are injected into supersorts, partial/total function symbols as partial/total functions and predicate symbols as relations. Note that sorts are assumed to be interpreted as non-empty sets, unless they are introduced using the keywords **esort** or **etype** (for datatypes).

The satisfaction relation is the expected one for partial first-order sentences. A sort generation constraint (S', F', σ') holds in a model M if the carriers of the reduct of M along σ' of the sorts in S' are generated by function symbols in F'.

A theory Γ is a pair $\langle \Sigma, \Gamma \rangle$ where Σ is a signature and Γ a set of sentences. A theory morphism $\sigma\colon \langle \Sigma, \Gamma \rangle \longrightarrow \langle \Sigma', \Gamma' \rangle$ is a signature morphism $\sigma\colon \Sigma \longrightarrow \Sigma'$ such that the sentences in Γ are mapped by σ to logical consequences of Γ'. A theory morphism $\sigma\colon \langle \Sigma, \Gamma \rangle \longrightarrow \langle \Sigma', \Gamma' \rangle$ is

- *conservative*, denoted $\mathbf{Cons}(\sigma)$, if each $\langle \Sigma, \Gamma \rangle$-model has a σ-expansion to a $\langle \Sigma', \Gamma' \rangle$-model;
- *monomorphic* ($\mathbf{Mono}(\sigma)$), if such an expansion exists uniquely up to isomorphism, and
- *definitional* ($\mathbf{Def}(\sigma)$), if each model has a unique such expansion.

We moreover write $\mathbf{DontKnow}(\sigma)$ and $\mathbf{NotCons}(\sigma)$ when the conservativity of σ can not be determined or does not hold, respectively. The following implications hold: $\mathbf{Def}(\sigma) \implies \mathbf{Mono}(\sigma) \implies \mathbf{Cons}(\sigma)$.

CASL specifications are built starting with basic specifications which are just theories. An extension of specifications is written: SP_1 **then** $\ldots$ **then** SP_n, where at each step $i = 2, \ldots n$, SP_i must extend the signature Σ_{i-1} constructed at the previous step to a correct CASL signature Σ_i.

3 The Conservativity Checker

Unfortunately already for first-order logic, neither the check for conservative, nor monomorphic, nor definitional extension are recursively enumerable, which means that there cannot be a complete (recursively axiomatized) calculus for them.

Let us consider a specification extension SP_1 **then** SP_2. The main idea of the algorithm for checking conservativity of this extension is to separate the definitions of SP_2 into definitions of new sorts and datatypes and definitions of new functions and predicates. Each of the definitions is then analysed individually, and for each of them we obtain a consistency status. The possible answers are not conservative (**NotCons**), no result has been obtained (**DontKnow**), conservative (**Cons**), monomorphic (**Mono**), definitional (**Def**), and they are ordered by their strength as follows:

$$\mathbf{NotCons} < \mathbf{DontKnow} < \mathbf{Cons} < \mathbf{Mono} < \mathbf{Def}$$

The result of the analysis of the conservativity status of the extension is then obtained as the minimum of the results of the analysis of all its definitions w.r.t. the order given above. We denote the conservativity calculus for extensions of specifications thus defined by $\vdash SP_1$ **then** $SP_2 \rhd$ **Status**.

3.1 Checking conservativity for sorts and datatypes

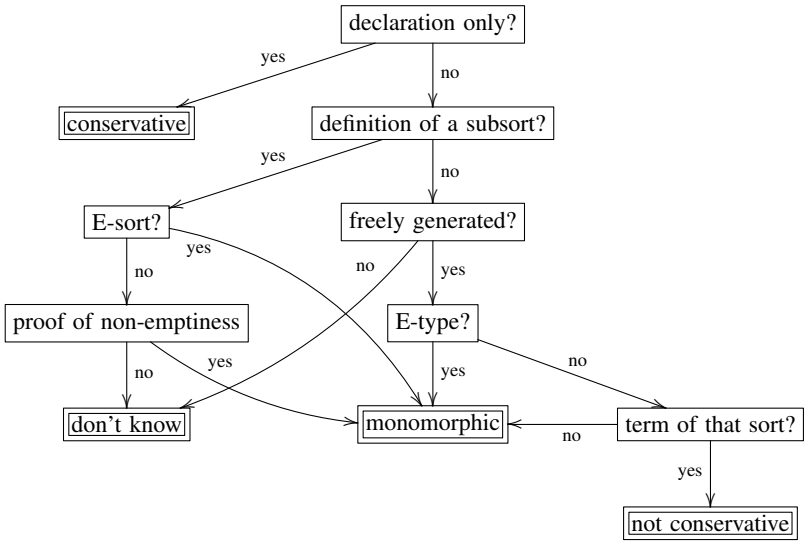

Fig. 1. Check for sorts and datatypes

If SP_2 contains no definitions of new sorts, then the result of analysis is just **Def** (the neutral element w.r.t the minimum operation). Otherwise, if a new sort or subsort has been declared without a definition, it does not depend on the old symbols and can be interpreted in any way, thus its status is **Cons**. If a sort is defined as a subsort of an existing sort with the help of a predicate Φ (as in $t = \{x : s.\Phi(x)\}$), then for a given model M of SP_1 the subsort t can be interpreted as any set isomorphic to the subset M_Φ. The result of analysis is thus **Mono**, provided that t has been declared as a **esort** or if the subset can be proven as non-empty. In the latter case a proof obligation is introduced and if it can not be discharged the status becomes **DontKnow**. If a sort t is defined as a free datatype, the status is **Mono**, provided that t has been declared as a **etype** or the specification ensures existence of a term of sort t. If this is not the case, the status becomes **NotCons**. This is summarised in Fig. 1.

3.2 Checking conservativity for functions and predicates

Figure 2 shows the decision diagram for operation and predicate symbols. For each such symbol, all definitions involving the symbol are collected. Definitions are sentences of the following form

sentence	defined symbol	type of definition
$\forall x_1 : s_1, \ldots x_m : s_m \,.\, f(t_1, \ldots, t_n) = t$	f	function definition
$\forall x_1 : s_1, \ldots x_m : s_m \,.\, \neg def f(t_1, \ldots, t_n)$	f	function definition
$\forall x_1 : s_1, \ldots x_m : s_m \,.\, def f(t_1, \ldots, t_n) \Leftrightarrow \psi$	f	domain axiom
$\forall x_1 : s_1, \ldots x_m : s_m \,.\, p(t_1, \ldots, t_n) \Leftrightarrow \psi$	p	predicate definition

All sentences not associated to sorts are required to have one of these forms (any exception immediately leads to a **DontKnow**). So does any sentence which is a definition of an "old" symbol. The first check for a symbol is that for (syntactic) confluence (or proof of semantic confluence) of all its definitions. For total functions, we need to prove termination of the definitions in order to ensure totality. Sufficient completeness then implies that the function is uniquely defined; otherwise, we only know that the definition has at least one solution. For partial function symbols, the presence of a domain axiom is checked. If there is more than one, we end with **DontKnow**. If there is exactly one, we need to prove that the remaining definitions have a least fixed-point (taken for the function graphs) whose domain is captured by the domain axiom(s). If this proof succeeds, we can proceed as for total functions. If there are no domain axioms, we need to prove termination as well in order to proceed as with total functions. Otherwise, we still have conservativity.

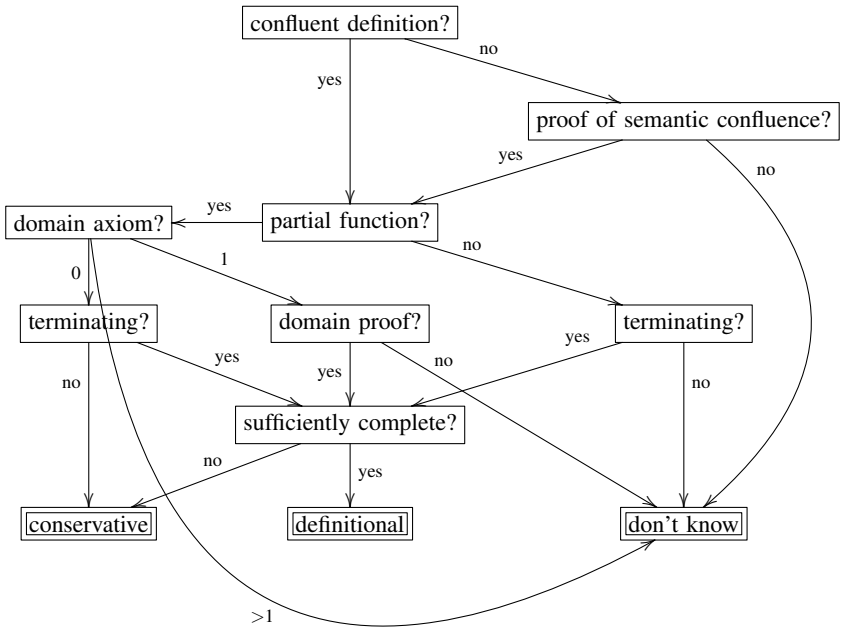

Fig. 2. Check for definitions of functions and predicates

The following result has been proved in [6].

Theorem 1 (Soundness). *If* $\vdash SP_1$ **then** $SP_2 \rhd$ **Status** *then* **Status**$(\iota : SP_1 \longrightarrow SP_2)$.

4 Examples

As an example, we present a simple but illustrative HETS library of natural numbers and operations on them. More examples, illustrating all branches of the diagrams, are available at `https://hets.dfki.de/Hets-lib/Conservativity/examples.het`.

spec NAT = **%mono**
free type $Nat ::= 0 \mid suc(Nat)$

spec NAT_COMP =
NAT **then** **%def**
free
{**pred** $\_\_<\_\_ : Nat \times Nat$
$\forall\, x, y : Nat$
$\bullet\ 0 < suc\ (x)$
$\bullet\ x < y \Rightarrow suc\ (x) < suc\ (y)$
}

spec POS =
NAT **then** **%mono**
sort $Pos = \{p : Nat \bullet \neg\, p = 0\}$

spec NAT_ADD =
NAT **then** **%def**
op $\_\_+\_\_ : Nat \times Nat \to Nat$
$\forall\, x, y : Nat$
$\bullet\ x + 0 = x$
$\bullet\ x + suc\ (y) = suc(x + y)$

spec NAT_PRED =
NAT **then** **%def**
op $pre : Nat \to ? Nat$
$\forall\, x : Nat$
$\bullet\ \neg\, def\ (pre(0))$
$\bullet\ pre(suc(x)) = x$

spec NAT_DIFF =
NAT_PRED **and** NAT_COMP **then** **%def**
op $\_\_-\_\_ : Nat \times Nat \to ? Nat$
$\forall\, x, y : Nat$
$\bullet\ def\ (x - y) \Leftrightarrow y < x \vee y = x$
$\bullet\ def\ (x - 0) \Rightarrow x - 0 = x$
$\bullet\ def\ (x - suc\ (y)) \Rightarrow x - suc\ (y) = pre(x - y)$

HETS analyses the specifications and produces a development graph [9]. Checking that NAT is monomorphic using HETS is done by right-clicking the node of NAT (its

yellow colour denotes that a conservativity proof obligation has been introduced) and selecting "Check conservativity", with the meaning that conservativity is checked over the empty signature. This also implies that NAT is consistent. HETS provides an interface for checking consistency of a theory directly, using a conservativity checker, by selecting "Check consistency" in the context menu of the node. This is illustrated in Fig. 3. In all other cases, the conservativity checker can be invoked by right-clicking the corresponding link (which is marked with **Mono?** or **Def?**) and selecting "Check conservativity".

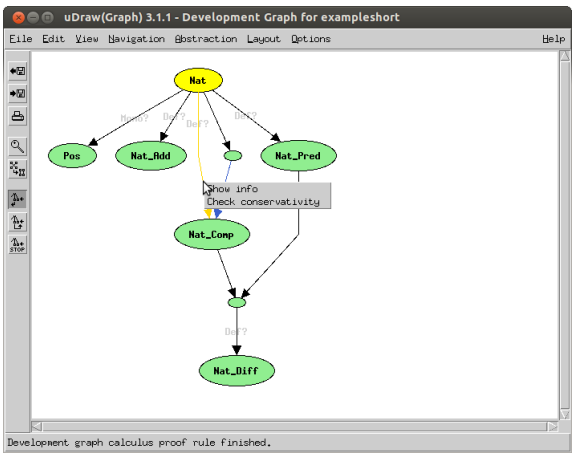

Fig. 3. Checking conservativity with HETS

HETS also uses conservativity to simplify the task of checking consistency of large theories. A node in a development graph is consistent if it has an incoming path with the origin in a node that has been proven to be consistent and such that all the morphisms contained in the path have been proven to be conservative.

We have run our conservativity checker for all the links in the CASL basic libraries [11]. The results are as follows:

don't know	790
conservative	455
conditionally conservative	49
monomorphic	25
conditionally monomorphic	11
definitional	322
sum	**1652**

The conditional variants indicate that some proof obligation has to be shown before conservativity (or monomorphicity) can be guaranteed.

5 Conclusion and Related Work

We have presented a tool (as part of the Heterogeneous Tool Set HETS) for checking conservativity of first-order specifications in CASL. In [5] we have shown the upper ontology DOLCE to be consistent using HETS architectural specifications. This has involved 38 checks of consistency and conservativity, which all have been done with the HETS conservativity checker.

A tool with similar scope is the CASL consistency checker (CCC) [7]. However, it only provides a cumbersome-to-use calculus, whereas our tool uses a decision diagram giving automated results in most cases (proof obligations are generated only in a few cases). The CCC also has rules for structured specifications; in HETS these are realised as so-called development graphs rules [9].

Future work will integrate recent research on SMT solvers, in particular, the "big engine" approach with a generic base theory [1].

References

1. B. Beckert, T. Hoare, R. Hähnle, D. R. Smith, C. Green, S. Ranise, C. Tinelli, T. Ball, and S. K. Rajamani. Intelligent systems and formal methods in software engineering. *IEEE Intelligent Systems*, 21(6):71–81, 2006.
2. M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273, 2002.
3. M. Codescu and T. Mossakowski. Refinement trees: calculi, tools and applications. In A. Corradini and B. Klin, editors, *CALCO 2011*, LNCS 6859, pages 145–160, 2011.
4. CoFI (The Common Framework Initiative). CASL *Reference Manual*. 2960 (IFIP Series). Springer, 2004.
5. O. Kutz and T. Mossakowski. A modular consistency proof for Dolce. In W. Burgard and D. Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence and the Twenty-Third Innovative Applications of Artificial Intelligence Conference*, pages 227–234. AAAI Press; Menlo Park, CA, 2011.
6. M. Liu. Konsistenz-Check von CASL-Spezifikationen. Master's thesis, University of Bremen, 2008.
7. C. Lüth, M. Roggenbach, and L. Schröder. CCC - the CASL consistency checker. In J. L. Fiadeiro, P. D. Mosses, and F. Orejas, editors, *WADT*, LNCS 3423, pages 94–105, 2004.
8. T. S. E. Maibaum. Conservative extensions, interpretations between theories and all that! In M. Bidoit and M. Dauchet, editors, *TAPSOFT 97*, volume 1214 of *LNCS*, pages 40–66. Springer, 1997.
9. 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.
10. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editors, *TACAS 2007*, volume 4424 of *LNCS*, pages 519–522. Springer, 2007.
11. M. Roggenbach, L. Schröder, and T. Mossakowski. Libraries. In P. Mosses, editor, *CASL reference manual*, volume 2960 of *Lecture Notes in Computer Science*. Springer; Berlin; http://www.springer.de, 2004. Early version appeared as CoFI note L-12, http://www.informatik.uni-bremen.de/cofi/old/Notes/L-12/index.html.
12. D. Sannella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. EATCS Monographs on theoretical computer science. Springer, 2012.