From specifications to code in Casl

David Aspinall and Donald Sannella

Laboratory for Foundations of Computer Science,
Division of Informatics, University of Edinburgh

Abstract. The status of the Common Framework Initiative (CoFI) and the Common Algebraic Specification Language (Casl) are briefly presented. One important outstanding point concerns the relationship between Casl and programming languages; making a proper connection is obviously central to the use of Casl specifications for software specification and development. Some of the issues involved in making this connection are discussed.

1 Introduction

The *Common Framework Initiative*, abbreviated CoFI, is an open international collaboration which aims to provide a common framework for algebraic specification and development of software by consolidating the results of past research in the area [AKBK99]. CoFI was initiated in 1995 in response to the proliferation of algebraic specification languages. At that time, despite extensive past collaboration between the main research groups involved and a high degree of agreement concerning the basic concepts, the field gave the appearance of being extremely fragmented, and this was seen as a serious obstacle to the dissemination and use of algebraic development techniques. Although many tools supporting the use of algebraic techniques had been developed in the academic community, none of them had gained wide acceptance, at least partly because of their isolated usability, with each tool using a different specification language.

The main activity of CoFI has been the design of a specification language called Casl (the *Common Algebraic Specification Language*), intended for formal specification of functional requirements and modular software design and subsuming many previous specification languages. Development of prototyping and verification tools for Casl leads to them being interoperable, i.e. capable of being used in combination rather than in isolation. Design of Casl proceeded hand-in-hand with work on semantics, methodology and tool support, all of which provided vital feedback regarding language design proposals. For a detailed description of Casl, see [ABK+03] or [CoF01]; a tutorial introduction is [BM01] and the formal semantics is [CoF02]. Section 2 below provides a brief taste of its main features. All Casl design documents are available from the main CoFI web site [CoF].

The main activity in CoFI since its inception has been the design of Casl including its semantics, documentation, and tool support. This work is now essentially finished. The language design is complete and has been approved by IFIP WG1.3 following two rounds of reviewing. The formal semantics of CASL is complete, and documentation including a book containing a user's guide and reference documents are nearing completion. A selection of tools supporting CASL are available, the most prominent of these being the CASL Tool Set CATS [Mos00a] which combines a parser, static checker, LaTeX pretty printer, facilities for printing signatures of specifications and structure graphs of CASL specifications, with an encoding of CASL specifications into second-order logic [Mos03] providing links to various verification and development systems including Isabelle and INKA [AHMS99].

The most focussed collaborative activity nowadays is on tool development, see `http://www.tzi.de/cofi/Tools`. There is also some further work on various topics in algebraic specification in the CASL context; for two recent examples see [MS02] and [BST02b]. However, with the completion of the design activity, there has been a very encouraging level of use of CASL in actual applications. In contrast with most previous algebraic specification languages which are used only by their inventors and their students and collaborators, many present CASL users have had no connection with its design. CASL has begun to be used in industry, and applications beyond software specification include a tentative role in the OMDoc project for the communication and storage of mathematical knowledge [Koh02]. Overall, CASL now appears to be recognized as a *de facto* standard language for algebraic specification.

One important aspect of any specification language is the way that specifications relate to programs. This can be a difficult issue; see [KS98] for a discussion of some serious problems at the specification/program interface that were encountered in work on Extended ML. The approach taken in CASL, which is outlined in Section 3, involves abstracting away from the details of the programming language. This works up to a point, but it leaves certain questions unanswered. In Section 4 we briefly outline a number of issues with the relationship between CASL specifications and programs that deserve attention. Our aim here is to raise questions, not to answer them. The discussion is tentative and we gloss over most of the technical details.

2 A taste of CASL

A CASL *basic specification* denotes a class of many-sorted partial first-order structures: algebras where the functions are partial or total, and where also predicates are allowed. These are classified by signatures which list sort names, partial and total function names, and predicate names, together with profiles of functions and predicates. The sorts are partially ordered by a subsort inclusion relation. Apart from introducing components of signatures, a CASL basic specification includes axioms giving properties of structures that are to be considered as models of the specification. Axioms are in first-order logic built over atomic formulae which include strong and existential equalities, definedness formulae and predicate applications, with generation constraints added as special, non-first-order sentences. Concise syntax is provided for specifications of "datatypes" with constructor and selector functions.

Here is an example of a basic specification:

free types $\mathit{Nat} ::= \mathit{0} \mid \textbf{sort } \mathit{Pos};$
$\quad\quad\quad\quad\mathit{Pos} ::= \mathit{suc}(\mathit{pre} : \mathit{Nat})$
op $\mathit{pre} : \mathit{Nat} \to ? \; \mathit{Nat}$
axioms
$\quad \neg \mathit{def}\ \mathit{pre}(\mathit{0});$
$\quad \forall n : \mathit{Nat} \bullet \mathit{pre}(\mathit{suc}(n)) = n$
pred $\mathit{even}\_\_ : \mathit{Nat}$
var $n : \mathit{Nat}$
$\bullet \quad \mathit{even}\ \mathit{0}$
$\bullet \quad \mathit{even}\ \mathit{suc}(n) \Leftrightarrow \neg \mathit{even}\ n$

The remaining features of CASL do not depend on the details of the features for basic specifications, so this part of the design is orthogonal to the rest. This is reflected in the semantics by the use of a variant of the notion of institution [GB92] called an *institution with symbols* [Mos00b]. (For readers who are unfamiliar with the notion of institution, it corresponds roughly to "logical system appropriate for writing specifications".) The semantics of basic specifications is regarded as defining a particular institution with symbols, and the rest of the semantics is based on an arbitrary institution with symbols. An important consequence of this is that sub-languages and extensions of CASL can be defined by restricting or extending the language of basic specifications without the need to reconsider or change the rest of the language.

CASL provides ways of building complex specifications out of simpler ones — the simplest ones being basic specifications — by means of various specification-building operations. These include translation, hiding, union, and both free and loose forms of extension. A *structured specification* denotes a class of many-sorted partial first-order structures, as with basic specifications. Thus the structure of a specification is *not* reflected in its models: it is used only to present the specification in a modular style. Structured specifications may be named and a named specification may be generic, meaning that it declares some parameters that need to be instantiated when it is used. Instantiation is a matter of providing an appropriate argument specification together with a fitting morphism from the parameter to the argument specification. Generic specifications correspond to what is known in other specification languages as (pushout-style) parametrized specifications.

Here is an example of a generic specification (referring to a specification named PARTIAL_ORDER, which is assumed to declare the sort *Elem* and the predicate $\_\_ \leq \_\_$):

spec LIST_WITH_ORDER [PARTIAL_ORDER] =
$\quad$ **free type** $\mathit{List}[\mathit{Elem}] ::= \mathit{nil} \mid \mathit{cons}(\mathit{hd} :? \mathit{Elem}; \mathit{tl} :? \mathit{List}[\mathit{Elem}])$
then

local
  op $insert : Elem \times List[Elem] \to List[Elem]$
  vars $x, y : Elem; l : List[Elem]$
  axioms $insert(x, nil) = cons(x, nil);$
  $x \leq y \Rightarrow insert(x, cons(y, l)) = cons(x, insert(y, l));$
  $\neg(x \leq y) \Rightarrow insert(x, cons(y, l)) = cons(y, insert(x, l))$
within
  op $order[\_\_ \leq \_\_] : List[Elem] \to List[Elem]$
  vars $x : Elem; l : List[Elem]$
  axioms $order[\_\_ \leq \_\_](nil) = nil;$
  $order[\_\_ \leq \_\_](cons(x, l)) = insert(x, order[\_\_ \leq \_\_](l))$
end

Architectural specifications in CASL are for describing the modular structure of software, in constrast to structured specifications where the structure is only for presentation purposes. An architectural specification consists of a list of unit declarations, indicating the component modules required with specifications for each of them, together with a unit term that describes the way in which these modules are to be combined. Units are normally functions which map structures to structures, where the specification of the unit specifies properties that the argument structure is required to satisfy as well as properties that are guaranteed of the result. These functions are required to be persistent, meaning that the argument structure is preserved intact in the result structure. This corresponds to the fact that a software module must use its imports as supplied without altering them.

Here is a simple example of an architectural specification (referring to ordinary specifications named LIST, CHAR, and NAT, assumed to declare the sorts *Elem* and *List*[*Elem*], *Char*, and *Nat*, respectively):

arch spec CN_LIST =
  units
    C : CHAR ;
    N : NAT ;
    F : ELEM → LIST[ELEM]
  result $F[C$ **fit** $Elem \mapsto Char]$ **and** $F[N$ **fit** $Elem \mapsto Nat]$

More about architectural specifications, including further examples, may be found in [BST02a].

3 Specifications and programs

The primary use of specifications is to describe programs; nevertheless CASL abstracts away from all details of programming languages and programming paradigms, in common with most work on algebraic specification. The connection with programs is indirect, via the use of partial first-order structures or similar mathematical models of program behaviour. We assume that each program P determines a CASL signature $Sig(P)$, and the programming language at hand comes equipped with a semantics which assigns to each program P its denotation as a partial first-order structure, $[\![P]\!] \in Alg(Sig(P))$. Then P is regarded as satisfying a specification SP if $Sig(P) = Sig(SP)$ and $[\![P]\!] \in [\![SP]\!]$, where $Sig(SP)$ and $[\![SP]\!] \subseteq Alg(Sig(SP))$ are given by the semantics of CASL.

The type systems of most programming languages do not match that of CASL, and partial first-order structures are not always suitable for capturing program behaviour. In that case one may simply replace the institution that is used for basic specifications in CASL with another one that is tailored to the programming language at hand.

Example 3.1. A suitable institution for Standard ML (SML) would consist of the following components.

Signatures: These would be SML signatures, or more precisely *environments* as defined in the static semantics of SML [MTHM97] which are their semantic counterparts. Components of signatures are then type constructors, typed function symbols including value constructors, exception constructors, and substructures having signatures. The type system here is that of SML, where functions may be higher-order and/or polymorphic.

Models: Any style of model that is suitable for capturing the behaviour of SML programs could be used. For example, one could take environments as defined in the *dynamic* semantics of SML, where closures are used to represent functions.

Sentences: One choice would be the syntax used for axioms in Extended ML [KST97], which is an extension of the syntax of SML boolean expressions by quantifiers, extensional equality, and a termination predicate.

Satisfaction: If sentences are as in EML and models are as in the dynamic semantics of SML, then satisfaction of a sentence by a model is as defined in the verification semantics of EML [KST97]. □

Here is a variant of the sorting specification shown earlier, given for the SML instantiation of CASL, by adjusting the syntax of basic specifications.

spec LIST_WITH_POLYORDER =
local
  val $insert : (\alpha \times \alpha \to bool) \to \alpha \times \alpha\ list \to \alpha\ list$
  vars $x, y : \alpha; l : \alpha\ list$
  axioms $insert\ leq\ (x, nil) = cons(x, nil);$
    $leq(x, y) \Rightarrow insert\ leq\ (x, cons(y, l)) =$
      $cons(x, insert\ leq\ (y, l));$
    $\neg(leq(x, y)) \Rightarrow insert\ leq\ (x, cons(y, l)) =$
      $cons(y, insert\ leq\ (x, l))$
within
  op $order : (\alpha \times \alpha \to bool) \to \alpha\ list \to \alpha\ list$
  vars $x : \alpha; l : \alpha\ list$

$$\begin{aligned}\textbf{axioms}\ & \mathit{order}\ \mathit{leq}\ (\mathit{nil}) = \mathit{nil};\\ & \mathit{order}\ \mathit{leq}\ (\mathit{cons}(x,l)) = \mathit{insert}\ \mathit{leq}\ (x, \mathit{order}\ \mathit{leq}\ l)\end{aligned}$$

Example 3.2. An institution for a simplified version of Java could be defined by combining and recasting ideas from μJava [NOP00] and JML [LBR01].

Signatures: A signature would represent the type information for a collection of classes, including class names, types and names of fields and methods in each class, and the subclass relationship between classes.

Models: A natural choice for models is based on execution traces within an abstract version of the Java Virtual Machine [NOP00]. A trace is a sequence of states, each including a representation of the heap (the current collection of objects), as well as a program counter indicating the next method to be executed and a stack containing the actual arguments it will be invoked with.

Sentences: An appropriate choice would be a many-sorted first-order logic which has non side-effecting Java expressions as terms. When specifying object-oriented systems, it is desirable to allow both *class invariants* which express properties of the values of fields in objects, as well as *method pre-post conditions* which express the behaviour of methods. Post conditions need some way to refer to the previous state (possibilities are to use auxiliary variables as in Hoare logic [NOP00], or the $\mathit{Old}(-)$ function of JML [LBR01]), as well as the result value of the method for non void returning methods. It would also be possible to add constructs for specifying exceptional behaviour and termination conditions.

Satisfaction: Roughly, a class invariant is satisfied in a model if it is satisfied in every state in the execution trace, and a method pre-post condition is satisfied if the pre-condition implies the post condition for all pairs of states corresponding to the particular method's call and return points. In practice, we need to be careful about certain intermediate states where class invariants may be temporarily violated; see [LBR01] for ways of describing this, as well as ways of specifying frame conditions which restrict which part of the state a method is allowed to alter. □

Example 3.3. An institution for Haskell-with-CASL is described in [SM02]. It is relates closely to the normal first-order CASL institution, and has been studied in more detail than our sketches for Java and SML above. Here is an overview:

Signatures: These consist of Haskell-style type classes including type constructors, type aliases, and type schemes for operators. There is a way to reduce rich signatures to simpler ones close to ordinary CASL, except that higher-order functions are present.

Models: Models are over basic signatures, and given by intensional Henkin models. Like the institutions outlined above, this choice reflects a more computationally-oriented viewpoint, allowing particular models which capture operational interpretations.

Sentences and satisfaction: Full formulae are similar to those of first-order CASL, but are reduced to a restricted *internal* logic on which satisfaction is defined. □

This institutional approach takes a model-theoretic view and says nothing about how sentences can be *proved* to hold. For this, one would require an associated *entailment system*, see [Mes89].

4 Some unresolved issues

Defining an institution for specifying programs in a given programming language, as in the above examples, provides a link between Casl and the programming language at hand. This gives an adequate framework for analysis of the process of developing programs from specifications by stepwise refinement using Casl architectural specifications, see e.g. [BST02a] and [BST02b].

Still, this seems to be only part of a bigger and more detailed story. Notice that the syntax of programs does not appear anywhere in the institutions outlined in Examples 3.1 and 3.2. One would expect a full account to take into consideration the structure of the programming language, rather than regarding it as a monolithic set of notations for describing a model.

4.1 Combining specifications and programs

We have made the point that specification structure is in general unrelated to program structure, and it often makes sense to use a completely different structure for an initial requirements specification than is used for the eventual implementation. If we are to connect the two formally, however, it is useful to have a path between them. This is what architectural specifications in Casl are intended to provide, as a mechanism for specifying implementation structure.

Architectural specifications in Casl can make use of certain *model building operators* for defining units. These are defined analogously to the specification building operators available for structured specifications. They including renaming, hiding, amalgamation, and the definition of generic units. The semantics of the model building operators is defined for an arbitrary institution with symbols; but once a specific programming language is chosen, it remains to decide how the model building operators can be realised [BST02a]. It may be that none, some, or all of them can be defined directly within the programming language.

Example 4.1. (Continuing Example 3.1) In SML, an architectural unit is a structure (possibly containing substructures) and a generic unit corresponds to a functor. The ways that units are combined in Casl correspond with the ways that structures and functors are combined in SML; it is possible to define renaming, hiding and amalgamation within the language itself. □

Example 4.2. (Continuing Example 3.2) In Java, an architectural unit is perhaps best identified as a set of related classes belonging to the same package. Java has visibility modifiers and interfaces which control access to these units, but there is no dedicated program-level syntax for combining pieces in this higher level of organization. Instead, the operations for constructing architectural units in Casl must be simulated by *meta*-operations on Java program syntax. Moreover, there is nothing corresponding to a generic unit: we must treat generic units as meta-level macros which are expanded at each instantiation. □

Even if the Casl model building operators are definable within the programming language, it may be preferable to use source-level transformations to realise them. This choice will be part of explaining how to instantiate Casl to a particular programming language.

Casl provides model building operations for combining units, but has no built-in syntax for constructing basic units. One way to link to a specific programming language would be add syntax from the programming language for basic units (and perhaps also syntax for combining units, e.g. SML functors). Here's a simple example of a unit definition using SML syntax (assuming Partial_Order has been slightly adapted for SML syntax):

```
unit PairOrder : Partial_Order =
  struct
      type Elem = int × int;
      fun leq((x1, x2), (y1, y2)) = (x1 < x2 orelse
                                     (x1 = x2 andalso y1 ≤ y2))
  end
```

This mechanism gives a Casl-based language for writing specifications with pieces of programs inside; such specifications induce proof obligations.

Conversely, we would also like to explain so-called "wide-spectrum" approaches to specification, in which pieces of specification are written inside programs, as exemplified by Extended ML and JML. Although adopting the wide-spectrum approach throughout a formal development may risk early commitment to a particular program structure, it is appealing for programmers because they can be introduced to specification annotations gradually, rather than needing to learn a whole new language.

If we have an institution $\mathcal{I}_{PL}$ for a programming language PL, we can imagine a crude way of adding assertions to the language by attaching sentences ϕ from $\mathcal{I}_{PL}$ to programs P. In reality, we would want to attach sentences more closely to the area of the program they refer to, which depends on the specific language being considered.

The two scenarios we have discussed can be visualized like this:

$$\begin{tikzcd}
\text{wide-spectrum } PL \arrow[r, leftrightarrow, dotted] & CASL(\mathcal{I}_{PL}) + PL \\
PL \arrow[u, hook] \arrow[r, leftrightarrow, dotted] & \mathcal{I}_{PL} \arrow[u, hook]
\end{tikzcd}$$

Here, "wide-spectrum PL" is a programming language PL extended with specification annotations, and $CASL(\mathcal{I}_{PL})$ is a Casl extension for the institution $\mathcal{I}_{PL}$.

To make each side work, we need to consider how to combine the static semantics of the languages to interpret pieces of programs or specifications in their surrounding context. We have two frameworks for specifying, on the boundary of programming and specification, but they are not quite the same.

4.2 Models of programs vs. models of specifications

The activities of specification and programming are quite different. Specification languages put a premium on the ability to express *properties* of functions and data in a convenient and concise way, while in programming languages the concern is with expressing *algorithms* and *data structures*. This "what vs. how" distinction leads to obvious differences in language constructs. More subtle differences arise on the level of models. As explained in Section 3, the Casl approach is to use the same kind of models for programs and for specifications. The following example from [ST96] shows how this might lead to problems.

Example 4.3. Let ϕ_{equiv} be a sentence which asserts that $equiv(n, m) = true$ iff the Turing machines with Gödel numbers n and m compute the same partial function (this is expressible in first-order logic with equality). Now consider the following specification:

local
  op $equiv : Nat \times Nat \to Bool$
  axioms $\phi_{\mathtt{equiv}}$
within
  op $opt : Nat \to Nat$;
  vars $n : Nat$
  axioms $equiv(opt(n), n) = true$

This specifies an optimizing function *opt* transforming TMs to equivalent TMs. (Axioms could be added to require that the output of *opt* is at least as efficient as its input.) If the models in use require functions to be computable, as in Examples 3.1 and 3.2, then this specification will have *no* models because there is no computable function *equiv* satisfying ϕ_{equiv}. Yet there *are* computable functions *opt* having the required property, for instance the identity function on Nat. Thus this specification disallows programs that provide exactly the required functionality.[1] □

This example demonstrates that there is a potential problem with the use of "concrete" models like those in the operational semantics of Standard ML. The problem does not disappear if one uses more "abstract" models, as in denotational semantics and Example 3.3. Such models impose restrictions on function spaces in order to interpret fixed point equations. Further restrictions are imposed in a desire to reflect more accurately the constraints that the programming

[1] One way out might be to use a *predicate* for *equiv* instead of a function, extending models to interpret predicates in such a way that predicates are not required to be undecidable.

language imposes, for example functions in a polymorphic language like SML might be required to be *parametric*, i.e. behave uniformly for all type instances [BFSS90]. Imposing such restrictions, whatever they are, gives rise to examples like the one above. Whether or not such an example illustrates a problem that needs to be solved is a different question. There is also an effect on the meaning of quantification: $\forall f : \tau \to \tau \,.\, \phi$ is more likely to hold if the quantifier ranges over only the parametric functions [Wad89,PA93].

A different problem arises from the use of the same *signatures* for programs and specifications. For specification purposes, one might want richer signatures with auxiliary components (that are not meant to be implemented) for use in specifying the main components (that *are* meant to be implemented). An example is the use of predicates specifying class invariants in JML [LBR01]. The addition of such auxiliary components does not seem to present substantive problems, but it is not catered for by the simple view of the relationship between specifications and programs presented in Section 3.

4.3 Relationships between levels and between languages

A classical topic in semantics is compiler correctness, see e.g. [BL69,Mor73], where the following diagram plays a central role:

$$\begin{tikzcd} PL \arrow[r, "c"] \arrow[d, "\llbracket\cdot\rrbracket"'] & PL' \arrow[d, "\llbracket\cdot\rrbracket'"] \\ M \arrow[r, leftrightarrow, "e"'] & M' \end{tikzcd}$$

Here, PL is the source language, PL' is the target language, c is a compiler, $\llbracket\cdot\rrbracket$ and $\llbracket\cdot\rrbracket'$ are the semantics of PL and PL' respectively, and e is some kind of encoding or simulation relating the results delivered by $\llbracket\cdot\rrbracket$ and $\llbracket\cdot\rrbracket'$. The compiler c is said to be correct if the diagram commutes, i.e. if for any $P \in PL$ the result of interpreting P is properly related by e to the result of interpreting $c(P) \in PL'$.

Given institutions for PL and PL', we may recast this to require compilation to preserve satisfaction of properties. One formulation would be to require that for all programs $P \in PL$ and sentences ϕ over $Sig(P)$,

$$\llbracket P\rrbracket \models \phi \qquad \Longrightarrow \qquad \llbracket \widehat{P}\rrbracket' \models \widehat{\phi}$$

where $\widehat{P}$ is the result of compiling P and $\widehat{\phi}$ is a translation of the property ϕ into the terms of $\widehat{P}$. If the set of sentences over $Sig(P)$ is closed under negation then this is equivalent to what is obtained if we replace $\Rightarrow$ by $\Leftrightarrow$, and then this is a so-called *institution encoding* [Tar00] (cf. [GR02] where the name "forward institution morphism" is used for the same concept).

An issue of great practical importance is "interlanguage working" in which programs in one programming language can work directly with libraries and components written in another. One way to achieve this is via a low-level intermediate language which is the common target of compilers for the programming languages in question. Microsoft's .NET platform is an example, with the CIL common intermediate language [MG01]. In such a framework, faced with an architectural specification of the form:

arch spec ASP =
  units
    U1 : *SP1* ;
    U2 : *SP2*
  result ... *U1* ... *U2* ...

one might consider implementing $U1$ in one language and $U2$ in a different language. If $U1$ satisfies $SP1$ and $U2$ satisfies $SP2$, then it would be possible to combine $\widehat{U1}$ and $\widehat{U2}$ as indicated in ASP to obtain a program in the common intermediate language, with (under appropriate conditions) $\widehat{U1}$ satisfying $\widehat{SP1}$ and $\widehat{U2}$ satisfying $\widehat{SP2}$.

5 Conclusion

We have considered various issues in connecting programming languages to Casl, including the possibility of connecting several languages at once to allow inter-language implementations.

Once we have characterised a setting for working with specifications and programs, there is more to do before we have a framework for formal development. Perhaps the most important question is: what do we actually want to do with our specifications?

There are various possibilities. We can use specifications as a basis for testing [Gau95]; they might be used to generate code for run-time error checking [WLAG93,LLP+00], or they may be used as a basis for additional static analysis [DLNS98].

The traditional hope is to be able to prove properties about specifications, and to prove that implementations satisfy specifications. A common approach for this is to connect the specification language to an already existing theorem prover. There is a range of strategies here. The extremes are represented by a *shallow embedding* which formalizes just the semantics of the represented language directly within a theorem prover's logic, and a *deep embedding*, which formalizes the syntax of the language being represented, together with a meaning function which describes its semantics within the theorem prover's logic [RAM+92]. Shallow embeddings have the advantage of direct reasoning within the theorem prover's logic, but require a degree of compatibility between the logic and the language being formalized. A deep embedding, by contrast, is more flexible, but typically more difficult to use, so that proof principles for the represented language may have to be derived. Deep embeddings may also allow formalization of meta-properties of the language being represented, although whether this is useful or not depends on the application. Among the existing work in connecting theorem provers to languages, the work on CATS [Mos00a] and

HasCasl [SM02] use shallow embeddings, whereas the work on μJava [NOP00] and ASL+$_{\text{FPC}}$ [Asp97] use deep embeddings; in the case of μJava the object of the formalization is to prove meta-properties about Java, rather than reason about Java programs.

References

[ABK+03] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella and A. Tarlecki. Casl: The common algebraic specification language. *Theoretical Computer Science*, 2003. To appear.

[AHMS99] S. Autexier, D. Hutter, H. Mantel and A. Schairer. System description: INKA 5.0 – a logic voyager. *Proc. 16th Intl. Conf. on Automated Deduction*. Springer LNAI 1632, 207–211, 1999.

[AKBK99] E. Astesiano, B. Krieg-Brückner and H.-J. Kreowski, editors. *Algebraic Foundations of Systems Specification*. Springer, 1999.

[Asp97] D. Aspinall. *Type Systems for Modular Programming and Specification*. PhD thesis, University of Edinburgh, 1997.

[BFSS90] E. Bainbridge, P. Freyd, A. Scedrov and P. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

[BL69] R. Burstall and P. Landin. Programs and their proofs: An algebraic approach. B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, 17–43. Edinburgh University Press, 1969.

[BM01] M. Bidoit and P. Mosses. A gentle introduction to Casl. Tutorial, WADT/CoFI Workshop at ETAPS 2001, Genova. Available from `http://www.lsv.ens-cachan.fr/~bidoit/CASL/`, 2001.

[BST02a] M. Bidoit, D. Sannella and A. Tarlecki. Architectural specifications in Casl. *Formal Aspects of Computing*, 2002. To appear.

[BST02b] M. Bidoit, D. Sannella and A. Tarlecki. Global development via local observational construction steps. *Proc. 27th Intl. Symp. on Mathematical Foundations of Computer Science*, Warsaw. Springer LNCS, 2002. To appear.

[CoF] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.brics.dk/Projects/CoFI/`.

[CoF01] CoFI Language Design Task Group. Casl – The CoFI Algebraic Specification Language – Summary, version 1.0.1. `Documents/CASL/v1.0.1/Summary`, in [CoF], 2001.

[CoF02] CoFI Semantics Task Group. Casl – The CoFI Algebraic Specification Language – Semantics, version 1.0. `Documents/CASL/Semantics`, in [CoF], 2002.

[DLNS98] D. Detlefs, K.R.M. Leino, G. Nelson and J. Saxe. Extended static checking. Technical Report #159, Compaq SRC, Palo Alto, USA, 1998.

[Gau95] M.-C. Gaudel. Testing can be formal, too. *Intl. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'95)*, Aarhus. Springer LNCS 915, 82–96, 1995.

[GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Assoc. for Computing Machinery*, 39:95–146, 1992.

[GR02] J. Goguen and G. Roşu. Institution morphisms. *Formal Aspects of Computing*, 2002. To appear.

[Koh02] M. Kohlhase. OMDoc: An open markup format for mathematical documents, version 1.1. Available from `http://www.mathweb.org/omdoc/index.html`, 2002.

[KS98] S. Kahrs and D. Sannella. Reflections on the design of a specification language. *Proc. Intl. Colloq. on Fundamental Approaches to Software Engineering. European Joint Conferences on Theory and Practice of Software (ETAPS'98)*, Lisbon. Springer LNCS 1382, 154–170, 1998.

[KST97] S. Kahrs, D. Sannella and A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[LBR01] G. Leavens, A. Baker and C. Ruby. Preliminary design of JML. Technical Report TR #98-06p, Department of Computer Science, Iowa State University, 2001.

[LLP+00] G. Leavens, K.R.M. Leino, E. Poll, C. Ruby and B. Jacobs. JML: notations and tools supporting detailed design in Java. *OOPSLA 2000 Companion*, Minneapolis, 105–106, 2000.

[Mes89] J. Meseguer. General logics. *Logic Colloquium '87*, 275–329. North Holland, 1989.

[MG01] E. Meijer and J. Gough. A technical overview of the commmon language infrastructure. Available from `http://research.microsoft.com/~emeijer/Papers/CLR.pdf`, 2001(?).

[Mor73] F. Morris. Advice on structuring compilers and proving them correct. *Proc. 3rd ACM Symp. on Principles of Programming Languages*, 144–152. ACM Press, 1973.

[Mos00a] T. Mossakowski. CASL: From semantics to tools. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000), European Joint Conferences on Theory and Practice of Software*, Berlin. Springer LNCS 1785, 93–108, 2000.

[Mos00b] T. Mossakowski. Specification in an arbitrary institution with symbols. *Recent Trends in Algebraic Development Techniques: Selected Papers from WADT'99*, Bonas. Springer LNCS 1827, 252–270, 2000.

[Mos03] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 2003. To appear.

[MS02] P. Machado and D. Sannella. Unit testing for CASL architectural specifications. *Proc. 27th Intl. Symp. on Mathematical Foundations of Computer Science*, Warsaw. Springer LNCS, 2002. To appear.

[MTHM97] R. Milner, M. Tofte, R. Harper and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[NOP00] T. Nipkow, D. von Oheimb and C. Pusch. μJava: Embedding a programming language in a theorem prover. F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Intl. Summer School Marktoberdorf 1999*, 117–144. IOS Press, 2000.

[PA93] G. Plotkin and M. Abadi. A logic for parametric polymorphism. *Proc. of the Intl. Conf. on Typed Lambda Calculi and Applications*, Amsterdam. Springer LNCS 664, 361–375, 1993.

[RAM+92] R. Boulton, A. Gordon, M. Gordon, J. Herbert and J. van Tassel. Experience with embedding hardware description languages in HOL. *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, Nijmegen, 129–156. North-Holland, 1992.

[SM02] L. Schröder and T. Mossakowski. HasCasl: Towards integrated specification and development of haskell programs. *Proc. 9th Intl. Conf. on Algebraic Methodology And Software Technology*, Reunion. Springer LNCS, this volume, 2002.

[ST96] D. Sannella and A. Tarlecki. Mind the gap! Abstract versus concrete models of specifications. *Proc. 21st Intl. Symp. on Mathematical Foundations of Computer Science*, Cracow. Springer LNCS 1113, 114–134, 1996.

[Tar00] A. Tarlecki. Towards heterogeneous specifications. D. Gabbay and M. van Rijke, editors, *Proc. of the Intl. Conf. on Frontiers of Combining Systems (FroCoS'98)*, 337–360. Research Studies Press, 2000.

[Wad89] P. Wadler. Theorems for free! *Proc. of the 4th Intl. Conf. on Functional Programming and Computer Architecture*. ACM Press, 1989.

[WLAG93] R. Wahbe, S. Lucco, T. Anderson and S. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.