EasyChair Preprint

№ 1490

A component-based formal language workbench

Peter Mosses

September 12, 2019

A Component-Based Formal Language Workbench

Peter D. Mosses
Delft University of Technology, Delft, The Netherlands
p.d.mosses@tudelft.nl

The CBS framework supports component-based specification of programming languages. It aims to significantly reduce the effort of formal language specification, and thereby encourage language developers to exploit formal semantics more widely. CBS provides an extensive library of reusable language specification components, facilitating co-evolution of languages and their specifications.

After introducing CBS and its formal definition, this short paper reports work in progress on generating an IDE for CBS from the definition. It also considers the possibility of supporting component-based language specification in other formal language workbenches.

1 Introduction

Developers of major programming languages always give *formal* specifications of *syntax*. For *semantics*, however, they usually resort to *informal explanations*. They sometimes define the formal semantics of sublanguages, but scaling up to full languages is usually regarded as a huge effort, and not worthwhile.

To encourage language developers to specify formal semantics of their full languages, it is essential to reduce the effort required – not only for an initial specification, but also for co-evolution of language specifications with the specified languages. The CBS framework aims to do just that, by providing an extensible library of *reusable language specification components*. The semantics of the components is defined once and for all, so simply translating a programming language to compositions of components specifies the language semantics. And specifying such translations can be significantly less effort than specifying language semantics directly.

Crucially, the definition of each component in the CBS library can be validated independently: adding new components to the library cannot invalidate bisimulation equivalences of previous components. After a component has been validated and released, its defined behaviour cannot be changed, so each occurrence of a particular component name in CBS specifications refers to the same definition. When a specified language evolves, its translation to components always has to change accordingly, as the components themselves cannot change.

Use of CBS is supported by an IDE for editing and validating specifications of components and languages. Validation is currently based on testing prototype implementations generated from specifications. The IDE is itself generated from a formal definition of the syntax and static analysis of CBS, which is specified in declarative meta-languages supported by the Spoofax language workbench [22]. CBS and its IDE have been developed by the PLanCompS project.[1]

Here, after recalling the main features of the CBS framework (§2), we give a progress report on the formal definition of CBS (§3). We explain how an IDE for CBS is generated from the definition (§4), and how prototype implementations of programming languages are generated from their specifications in CBS (§5). We also compare CBS with some other language specification frameworks regarding the possibility of defining libraries of reusable components (§6).

[1] Programming Language Components and Specifications, `http://plancomps.org`.

Preliminary Report. Final version to appear in:
F-IDE 2019

2 Component-based specification of programming languages

We start by briefly recalling the main features of the CBS framework for component-based specification. For more detailed expositions, see [5, 7, 10, 15].

A CBS for a programming language is a specification of an inductively-defined translation function, mapping well-formed program phrases to terms formed from so-called *fundamental programming constructs* ('funcons'). The specification includes a *grammar* for the (concrete and abstract) syntax of the language, and a *translation equation* for each alternative of the grammar. The translation of programs to funcon terms, together with the semantics of funcons, determines the semantics of the programs:

The funcon definitions are reusable components of language specifications.

Funcons. A funcon represents a common and reusable computational concept, such as variable assignment or looping. It usually corresponds to a simple ingredient of constructs commonly found in mainstream programming languages. For example, an assignment in a program might be an expression, returning either the target variable or the assigned value; the funcon for assigning a value to a variable simply has that effect, and does not return either of its arguments. Loops in programs can often be terminated abruptly by break statements; the funcons for loops simply propagate abrupt termination of their bodies, and other funcons are used to handle abrupt termination. Funcons for integer operations return the unbounded mathematical results, leaving it to other funcons to enforce boundedness. Etc.

CBS uses a modular variant of structural operational semantics (MSOS) [13] to define funcons. Elision of implicitly propagated semantic entities (environments, stores, etc.) ensures conciseness as well as modularity. Funcons are defined independently of any particular programming language, and of each other. Adding new funcons to the CBS library does not require any changes at all to previous definitions.

The intended interpretation of funcon definitions in CBS is based on their translation to MSOS, and thereby as value-computation transition systems [9]. The translation from a precursor of CBS (MSDF) to MSOS was originally defined in Prolog [10]; funcon definitions are currently translated declaratively to monadic Haskell code [5], where the monads involved correspond directly to the entities used in MSOS. The definition of the translation is validated empirically by using generated code to execute funcon terms (manually-written unit tests, and translations of test programs in languages specified in CBS).

Syntax. CBS includes a variant of BNF context-free grammars, extended with standard notation for regular expressions; the interpretation of such grammars as datatypes of ASTs is well established. Concise grammars for expression syntax are usually highly ambiguous; CBS allows disambiguation (associativity, relative priority, etc., as in SDF3 [20], [19, Ch. 2]) so the same grammar can be used to specify both abstract and concrete syntax. Lexical syntax can be context-free (e.g., for nested comments).

Translation functions. The equations used in CBS to specify translations from program ASTs to funcons look like semantic equations in denotational semantics: the left side is an application of a translation function to an AST pattern, with meta-variables matching sub-trees; the right side is a funcon term containing applications of translation functions to meta-variables. The meta-variables range over specified

sorts of sub-trees. Syntactic desugaring equations relate pairs of (composite) AST patterns. The equations are interpreted inductively as functions on ASTs, as usual.

Figure 1 illustrates the notation used in CBS for specifying funcons, syntax, and translation functions.

- Top left: *Exp* is declared as the stem of meta-variables ranging over `exp` phrases.
- Bottom left: The rules for the translation function `rval` map ASTs of `exp` phrases '*Exp1* && *Exp2*' to funcon terms formed from `if-else`, the translations of *Exp1* and *Exp2*, and `false`.
- Top right: The signature of the funcon `if-true-else` implies that its first argument is to be pre-evaluated; the two rules specify its subsequent reduction to one of its two unevaluated arguments. Aliases such as `if-else` support formal abbreviation in funcon terms.
- Bottom right: A SIMPLE program has been parsed and translated to a funcon term.

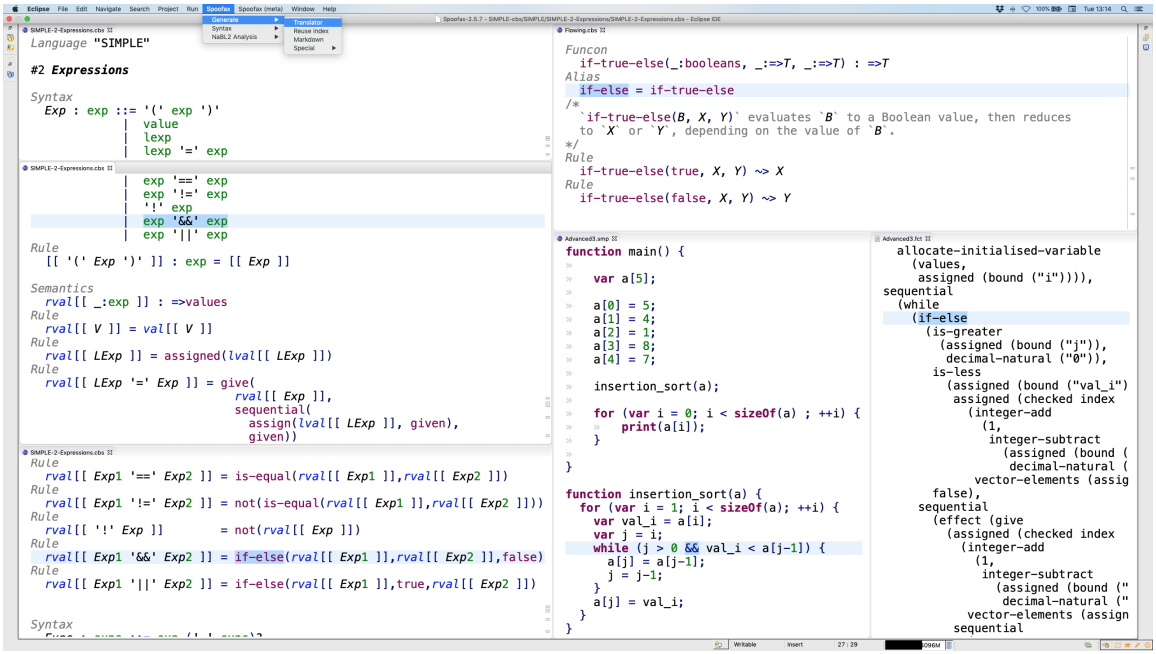

Figure 1: Using the CBS IDE on SIMPLE [17, Languages-beta]

3 Definition of the CBS meta-language

This section briefly explains how the syntax and static analysis of CBS are defined [14]; the meta-languages used are explained in the Spoofax documentation [22].

Syntax of CBS. We use SDF3 [20] to define the context-free syntax of CBS. SDF3 allows arbitrary context-free grammars; disambiguation is supported by associativity and relative priority specification for context-free syntax, and rejections and follow-restrictions for lexical syntax.

The current SDF3 definition of the syntax of CBS is given in [14, CBS/syntax]. Its formal interpretation is based on the transformation of SDF3 to SDF2 in [22] and the formal definition of SDF2 [21]. The syntax definition has been empirically validated for coverage and disambiguation against a collection of existing specifications written in CBS, using a parser generated from it by Spoofax [22].

Static analysis of CBS. We currently use NaBL2 [4] to define name resolution and type checking for CBS. NaBL2 rules map ASTs to sets of constraints involving scope graphs. The current NaBL2 definition of static analysis for CBS is given in [14, CBS/trans/static-semantics].

The rules for CBS name resolution require each funcon in the library to have a unique definition. A language specification can use any defined funcon (without explicit import). Grammar non-terminals, meta-variable stems, and names of translation functions are local to language specifications; the variables used in a semantic equation or operational rule are local to it, and checked to be source dependent.

However, NaBL2 is not sufficiently expressive to define the intended type checking of funcon terms in CBS. Our NaBL2 rules check that applications of funcons are consistent with signatures regarding number of arguments, and whether arguments may compute undefined results, but not the exact types of computed values (which involves structural subtyping). The rules also check whether the syntax patterns used in specifications of translation functions match the specified grammar alternatives. NaBL2 is to be superseded by a more powerful meta-language, Statix [3], which should support full CBS type checking.

4 Generation of the CBS IDE

The IDE for CBS is implemented using Spoofax [22]. As depicted in Figure 2, Spoofax generates the IDE directly from the definition of CBS: it generates a parser for CBS from its SDF3 grammar (with automatic error recovery, and syntax highlighting); it generates name resolution and arity checking constraints for CBS from the NaBL2 rules. Parsing and static analysis errors in specifications are flagged by Eclipse.

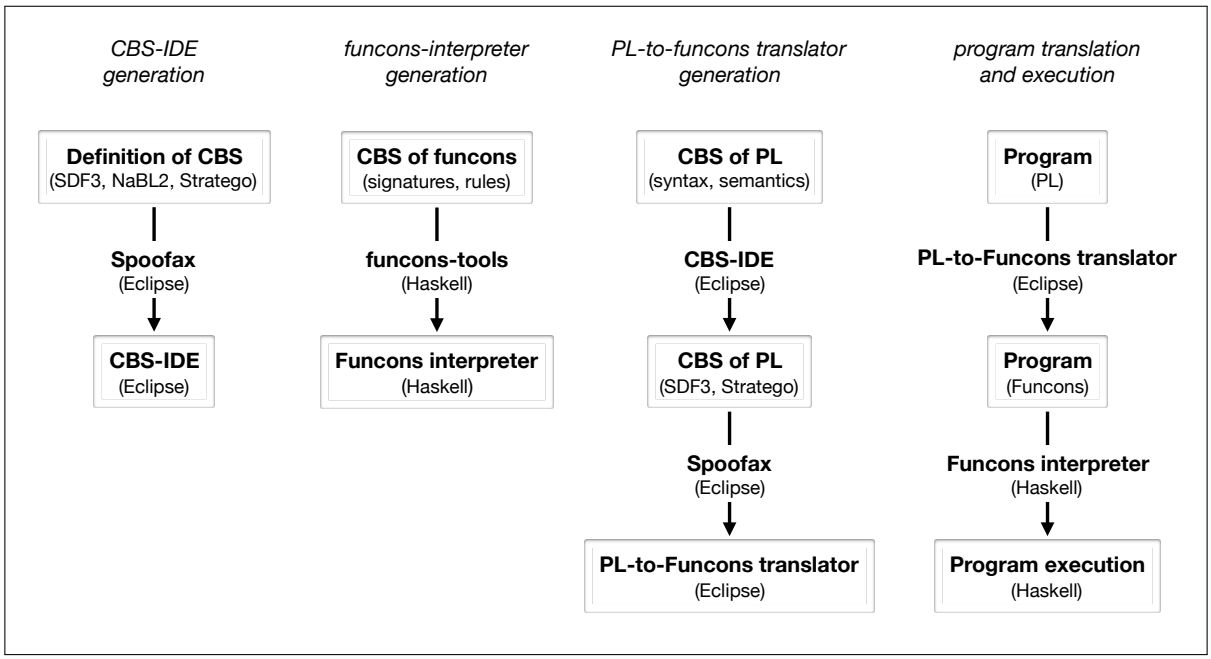

Figure 2: Generation and use of the CBS IDE

Each language specification in CBS is a separate Eclipse project, with shared access to the funcons library, which is stored in a remote repository. A language specification can be split into sections, and stored in any number of files. Multi-file name resolution generates hyperlinks from names to their definitions.

5 Generation of language interpreters from CBS

The generated IDE for CBS has a menu to generate various artefacts when editing a language specification written in CBS: a syntax-aware editor to parse programs and translate them to funcons, a list of reused funcons, and Markdown pages with embedded HTML for hyperlinks and highlighting. When a language specification is changed, stale generated files are regenerated. When the generated Markdown pages are uploaded to GitHub Pages, they create a responsive website (e.g., [17]) where language specifications and funcon definitions can be browsed using hyperlinks for navigation, as in Spoofax.

The Haskell package *funcons-tools* [6] can be used as an external tool from the CBS IDE. It generates highly modular Haskell code for funcon interpreters from CBS definitions of funcons [5].

6 Related work

CBS appears to be the only framework that currently provides a library of reusable components of programming language specifications. If the CBS library could be specified in other frameworks, it might encourage users of those frameworks to exploit funcons, and provide further tool support for CBS users.

The K-framework [18] is highly modular, and has been used to specify the semantics of several major languages. The funcons used in the CBS specification of SIMPLE [17, Languages-beta] have already been re-specified in K, allowing the CBS of SIMPLE programs to be run with the K tools [16]. However, the specification of the K configurations was monolithic, and depended on the set of funcons.

Reduction semantics is a popular form of operational semantics, with mature tool support including Redex [12]. Language extensions can be specified smoothly in Redex, but it is unclear how to define an open-ended collection of funcons: reduction semantics requires grammars for evaluation contexts, and the evaluation contexts for a particular funcon seem likely to depend on which other funcons are needed.

XASM [1] is a component-based language for Abstract State Machines [8], supporting the use of Montages [2] for specifying programming languages. However, it appears that the original home page (`xasm.org`) and the subsequent SourceForge project (`xasm.sourceforge.net`) are no longer in use, and that no library of reusable XASM components has ever been released.

The Overture F-IDE [11] supports dialects of the Vienna Development Method (VDM). Those specification languages contain many constructs which, like funcons, correspond closely to constructs of high-level programming languages: assignment statements, while-loops, exception-handling, etc. Thus translating a programming language to a VDM dialect would be similar to specifying its CBS. It could be interesting to investigate defining an extensible library of funcons in VDM, with IDE support in Overture.

7 Conclusion

The syntax and static semantics of CBS have been formally defined [14] using declarative meta-languages: SDF3 and NaBL2. The NaBL2 definition of CBS type checking needs to be reformulated in the new Statix meta-language [3], to specify rigorous subtype checks, after which the IDE generated from the definition of CBS is to be released as an Eclipse plugin.

Current examples of language specifications in CBS include OCaml-Light; scaling up to full OCaml (and other major languages) remains to be demonstrated. Bisimulation properties of funcons can be proved once and for all [9]; the proofs could be included with the definitions in the library.

The CBS development is part of the PLanCompS project. The project was initially supported by an EPSRC grant (2011–16), and it is continuing as an open collaboration; new participants are welcome.

References

[1] M. Anlauff (2000): *XASM – An Extensible, Component-Based ASM Language*. In: *ASM 2000*, *LNCS* 1912, Springer, pp. 69–90, doi:10.1007/3-540-44518-8_6.

[2] M. Anlauff, P.W. Kutter & A. Pierantonio (1999): *Tool Support for Language Design and Prototyping with Montages*. In: *CC'99*, *LNCS* 1575, Springer, pp. 296–299, doi:10.1007/978-3-540-49051-7_22.

[3] H. van Antwerpen, C. Bach Poulsen, A. Rouvoet & E. Visser (2018): *Scopes As Types*. *Proc. ACM Program. Lang.* 2, pp. 114:1–114:30, doi:10.1145/3276484.

[4] H. van Antwerpen, P. Néron, A.P. Tolmach et al. (2016): *A Constraint Language for Static Semantic Analysis Based on Scope Graphs*. In: *Proc. PEPM 2016*, ACM, pp. 49–60, doi:10.1145/2847538.2847543.

[5] L.T. van Binsbergen, P.D. Mosses & N. Sculthorpe (2019): *Executable Component-Based Semantics*. *J. Log. Algebr. Meth. Program.* 103, pp. 184–212, doi:10.1016/j.jlamp.2018.12.004.

[6] L.T. van Binsbergen & N. Sculthorpe (2019): *funcons-tools: A Modular Interpreter for Executing Funcons*. Available at `https://hackage.haskell.org/package/funcons-tools`. Hackage package.

[7] L.T. van Binsbergen, N. Sculthorpe & P.D. Mosses (2016): *Tool Support for Component-Based Semantics*. In: *Companion Proc. Modularity 2016*, ACM, pp. 8–11, doi:10.1145/2892664.2893464.

[8] E. Börger (2017): *The Abstract State Machines Method for Modular Design and Analysis of Programming Languages*. *J. Logic Comput.* 27, pp. 417–439, doi:10.1093/logcom/exu077.

[9] M. Churchill & P.D. Mosses (2013): *Modular Bisimulation Theory for Computations and Values*. In: *FOSSACS 2013*, *LNCS* 7794, Springer, pp. 97–112, doi:10.1007/978-3-642-37075-5.

[10] M. Churchill, P.D. Mosses, N. Sculthorpe & P. Torrini (2015): *Reusable Components of Semantic Specifications*. *LNCS Trans. Aspect Oriented Softw. Dev.* 12, pp. 132–179, doi:10.1007/978-3-662-46734-3_4.

[11] L.D. Couto, P. Gorm Larsen, M. Hasanagic et al. (2015): *Towards Enabling Overture as a Platform for Formal Notation IDEs*. In: *F-IDE 2015*, *EPTCS* 187, pp. 14–27, doi:10.4204/EPTCS.187.

[12] C. Klein, J. Clements, C. Dimoulas et al. (2012): *Run Your Research: On the Effectiveness of Lightweight Mechanization*. In: *POPL 2012*, ACM, pp. 285–296, doi:10.1145/2103656.2103691.

[13] P.D. Mosses (2004): *Modular Structural Operational Semantics*. *J. Log. Algebr. Program.* 60-61, pp. 195–228, doi:10.1016/j.jlap.2004.03.008.

[14] P.D. Mosses (2019): *CBS IDE*. Available at `https://plancomps.github.io/CBS-beta/docs/F-IDE-2019/CBS.zip`. Language specification project for Spoofax-2.5.7, unreleased prototype.

[15] P.D. Mosses (2019): *Software Meta-language Engineering and CBS*. *J. Comput. Lang.* 50, pp. 39–48, doi:10.1016/j.jvlc.2018.11.003.

[16] P.D. Mosses & F. Vesely (2014): *FunKons: Component-Based Semantics in K*. In: *WRLA 2014*, *LNCS* 8663, Springer, pp. 213–229, doi:10.1007/978-3-319-12904-4_12.

[17] PLanCompS Project (2019): *CBS: A Framework for Component-Based Specification of Programming Languages*. Available at `https://plancomps.github.io/CBS-beta`. Beta release.

[18] G. Rosu (2017): *K: A Semantic Framework for Programming Languages and Formal Analysis Tools*. In: *Dependable Software Systems Engineering*, IOS Press, pp. 186–206, doi:10.3233/978-1-61499-810-5-186.

[19] L.E. de Souza Amorim (2019): *Declarative Syntax Definition for Modern Language Workbenches*. Ph.D. thesis, Delft University of Technology, doi:10.4233/uuid:43d7992a-7077-47ba-b38f-113f5011d07f.

[20] L.E. de Souza Amorim, E. Visser & G. Wachsmuth (2014): *Developing SDF3*. In: *Parsing@SLE 2014*. Available at `https://www.sleconf.org/2014/parsing-slides/2-sdf3-slides.pdf`.

[21] E. Visser (1997): *Syntax Definition for Language Prototyping*. Ph.D. thesis, University of Amsterdam.

[22] E. Visser, G. Wachsmuth, A.P. Tolmach et al. (2014): *A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs*. In: *Proc. Onward! 2014*, ACM, pp. 95–111, doi:10.1145/2661136.2661149. Spoofax home page: `https://spoofax.readthedocs.io`.