Coalgebraic Class Specification Language (Reference manual)

Hendrik Tews

Inst. Theor. Informatik, TU Dresden, D-01062 Dresden, Germany.

tews@tcs.inf.tu-dresden.de

July 28, 1998

This report describes the **C**oalgebraic **C**lass **S**ecification **L**anguage CCSL, it's semantic in higher order logic and the CCSL compiler which if part of the LOOP project.

Contents

Forward

This reference manual is organized as follows.

Chapter 1 describes the grammar and the intended semantics of CCSL on an informal level.

Chapter 2 (to be written) gives the semantic of CCSL in higher order logic. That is the output of the compiler is described theory by theory.

Chapter 3 (to be written) describes the user interface of the compiler.

Throughout this reference manual a basic knowledge about object–orientation and a working knowledge about the theorem prover PVS is assumed.

1. Language description

In this chapter we describe the specification language CCSL itself.

The description of the grammar of CCSL will be given in BNF with the following additions: Brackets [...] denote optional components, Braces { ... } denote arbitrary repetition (including zero times). Terminals are set in `UPPERCASE TYPEWRITER`, Non–terminals in *lowercase slanted*. Be careful not to mix up the terminal symbols `[` and `]` with the meta level symbols [and].

1.1. Lexical conventions

1.1.1. Comments

CCSL allows OCAML style comments (enclosed by `(*` and `*)`) and PVS style comments (starting with `%` and reaching until the end of the current line). Note that inside of PVS expressions only PVS style comments are allowed. Nesting of OCAML style comments is handled.

1.1.2. Reserved Words

The list of keywords is

`ADT`	`AND`	`AS`	`ASSERTION`	`ATTRIBUTE`
`BASETYPE`	`BASETYPE`	`BEGIN`	`CARRIER`	`CLASSSPEC`
`CONSTRUCTOR`	`CREATION`	`END`	`ENDPVS`	`IMPORTING`
`IN`	`INHERIT`	`LOCAL`	`METHOD`	`PRIVATE`
`PUBLIC`	`PVS`	`RENAMING`	`SELF`	`SELFVAR`
`TYPE`	`VAR`			

Case is *not* significant for keywords.

1.1.3. Identifiers

Identifiers are strings built up from letters, digits, the underscore _, and the question mark ?. They are required to start with a letter. Case is significant, so `name` and `namE` are different identifiers.

1.1.4. Strings

Strings are enclosed by double quotes `"`. The backslash `\` serves as an escape character. Note that strings can only be used to specify importings. There are no basic expressions of strings.

1.1.5. Numbers

At the moment only natural numbers are handled. They are represented as a string of digits. There is no range restriction.

1.1.6. PVS formulae

Formulae of PVS can be embedded in CCSL between the keywords `pvs` and `endpvs`. There is no syntax nor type checking on such formulae. If they are not correct, they will yield an error when the code is parsed from PVS. It is possible to use a certain set of predefined identifiers in PVS formulas, see section 1.6.5 for more details. The word `ENDPVS` must not occur in an formula regardless of case.

1.2. Value expressions

$$\begin{array}{rcl} \mathit{expr} & ::= & \mathit{number} \\ & | & \mathit{identifier} \end{array}$$

A value is either a (positive) number or an identifier refering to a value parameter inside a specification.

1.3. Importings

$$\mathit{importing} \quad ::= \quad \texttt{IMPORTING}\ \mathit{string}$$

An importing makes the definitions of a PVS source file available in a specification. Depending on where the importing occurs, we distinguish three classes of importings: class–importings, assertion–importings, and creation–importings.

1.4. Type expressions

CCSL supports the usual type expressions except from labeled products. The syntax of type expressions is very close to PVS.

type	::=	*identifier*
	|	`SELF`
	|	`CARRIER`
	|	`[` *type* {`,` *type*} `->` *type* `]`
	|	`[` *type* {`,` *type*} `]`
	|	*identifier argumentlist*
argumentlist	::=	`[` *argument* {`,` *argument*} `]`
argument	::=	*type*
	|	*expr*

1.4.1. Named types

Previously defined types and non–parametric specifications (either ADT or classes) can be referred to by their identifier. Inside an parametric specification an identifier can also refer to the parameters of the specification. The usual scoping rules apply.

1.4.2. Recursive specifications

Inside an specification it is possible to refer to the carrier of the current model (assuming a model has been chosen). For a class specification this is the type on which the coalgebra acts. As usual in object–oriented languages `SELF` is used for this purpose.

In an ADT specification `CARRIER` denotes denotes the carrier set of the initial model[1]. Note that `SELF` and `CARRIER` are keywords, so you can not use them outside a specification.

1.4.3. Products

An expression `[`$T_1, \ldots, T_n$`]` denotes the product type $T_1 \times \cdots \times T_n$. Here n must be greater than or equal to 2. It is not possible to construct the empty product this way[2].

> ***Question:*** *shall we put Unit into the prelude?*

1.4.4. Function types

`[D -> C]` is the type of all functions with domain `D` and codomain `C`. If the domain is a product, it is possible to abbreviate `[[`$T_1, \ldots, T_n$`] ->` T_{n+1}`]` as `[`$T_1, \ldots, T_n$ `->` T_{n+1}`]`.

[1] For ADT specifications initial semantics is always used, see section 2.2
[2] the final object can be specified as an ADT

1.4.5. ADT's and Classes

id[$e_1, \ldots, e_n$] is the parametric specification id instantiated with the expressions listed. The expressions are either type expressions or value expressions. Their number and kind must match the number and kind of parameters of id.

1.5. Type declarations and Type definitions

```
typedef  ::=  TYPE identifier : BASETYPE
           |  TYPE identifier = type
```

Type declarations are used to introduce types to the CCSL compiler in order to avoid type errors. It is the user's responsibility to declare these types to PVS in an appropriate manner.

Type definitions can be used to abbreviate complicated type expressions. Both type declarations and definitions are purely syntactic sugar. They don't have any semantic impact (beside from type checking the CCSL source)

1.6. Class Specifications

```
classspec      ::=  BEGIN identifier [parameterlist] : CLASSSPEC
                    {importing} {classsection}
                    END identifier

parameterlist  ::=  [ parameter {, parameter} ]

parameter      ::=  identifier {, identifier} : TYPE
                 |  identifier {, identifier} : type

classsection   ::=  inheritsection
                 |  [visibility] attributesection [;]
                 |  [visibility] methodsection [;]
                 |  classconstructorsection [;]
                 |  assertionsection
                 |  creationsection

visibility     ::=  PUBLIC
                 |  PRIVATE
```

A class specification starts with a header consisting of the name of the specification and a possible declaration of parameters. The list of parameters is very similar to PVS. A parameter might be either a type parameter or a value parameter. In the latter case it's declaring type expression might contain type variables which refer to previously defined types. Parameters don't allow overloading. In case of equal names the previous parameter is hidden. The identifier in the header (the name of the specification) and after the `END` are required to match.

If the user wants to use types which he defined on his own in PVS, it is necessary to include an importing of the relevant files and to declare those types before as `basetypes`. If the user only wants to supply definitions and lemmas, he might consider augmenting the assertion section with an importing. All importings specified after the keyword `CLASSSPEC` are class–importings.

A class specification must at least contain one inherit–, attribute–, or method section. Besides that, it can contain an arbitrary number of the different sections in any order. Attributes and methods can be declared public or private. Without any visibility flag they are public. In the following section attributes, methods and constructors are commonly refered to as "fields". The corresponding sections consist of semicolon separated lists of fields. For convenience CCSL allows an optional final semicolon.

or should I use $\{\}^+$?

Question: *does anybody insist on the semicola?*

A class specification defines a (parametrized) type of the same name. Depending on compiler flags either loose or final semantic is used (see section 2.1.1 and 2.1.7).

1.6.1. Inheritance

inheritsection	::=	`INHERIT FROM` *ancestor* {`,` *ancestor*}
ancestor	::=	*identifier* [*argumentlist*] [`RENAMING` *renaming* {`AND` *renaming*}]
renaming	::=	*identifier* [`:` *type*] `AS` *identifier*

An inherit clause makes it possible to incorporate all attributes, methods, and assertions[3] of the ancestor specification into the current specification. Multiple inheritance is supported. The ancestor must refer to a class specification (possibly through a type abbreviation), in case the ancestor specification is parametric, actuals must be supplied.

Semantic of MI!

The user can also decide to rename certain fields of the ancestor class. Since overloading is allowed for methods and attributes, the type might be necessary to identify the right field. In case a type is given, it must be the type of the field in this class, ie. after the substitution of the parameters.

Question: *anybody tried to rename a constructor? a constant constructor?*

Resolving name clashes

A name clash occurs, if after renaming there are two fields of the same type in the specification (taking all fields of the specification itself and all inherited fields into account)

[3] in case of compiler switch `-ic` also constructors and creation conditions; see section 3.2

which have identical names. In this case inherited fields are *automatically* renamed and a warning message is generated. The following heuristic is used: If prepending the name of the ancestor class resolves the name clash the method is renamed accordingly. If this fails, digits are appended to the name in an unspecified but deterministic way.

1.6.2. Attributes

$$\begin{array}{lcl} \mathit{attributesection} & ::= & \texttt{ATTRIBUTE}\ \mathit{attribute}\ \{;\ \mathit{attribute}\} \\[1em] \mathit{attribute} & ::= & \mathit{identifier}\ :\ \texttt{SELF -> }\mathit{type} \end{array}$$

An attribute section starts with the keyword `ATTRIBUTE` and contains a semicolon separated list of attribute declarations.

The domain of an attribute must be `Self`. The codomain of an attribute must not contain `Self`.

> ***Question:*** *Why do we not allow attributes* $\texttt{Self} \times T \longrightarrow T'$*?*

CCSL allows to distinguish between two different kinds of attributes: proper and pseudo attributes. Proper attributes correspond to variables of which every object has a private copy[4]. Via variable update actual programming languages provide means to change the value of such attributes.

> ***Question:*** *where is a formal definition of proper?*

Pseudo attributes correspond to computations which yield no successor state of the object like generating a checksum. It makes no sense to update the value of a pseudo attribute. In contrast a (proper) method always delivers a successor state.

The attribute section can be used to declare proper attributes. For any attribute a of type $\texttt{SELF} \longrightarrow T$ an "update method" set_a of type $\texttt{SELF} \times T \longrightarrow \texttt{SELF}$ is generated by the compiler provided that a field with this name does not exist in the current class. The assertions are also augmented to specify the behavior of a as an proper attribute, see section 2.1.5.

Any pseudo attributes (and attributes for which the user doesn't want an update method) must be declared in the method section.

1.6.3. Methods

$$\begin{array}{lcl} \mathit{methodsection} & ::= & \texttt{METHOD}\ \mathit{method}\ \{;\ \mathit{method}\} \\[1em] \mathit{method} & ::= & \mathit{identifier}\ :\ \mathit{type}\ \texttt{->}\ \mathit{type} \end{array}$$

[4]Note: these are different from so called "class variables" of which all object of a given class share one copy

The method section starts with the keyword `METHOD` and contains a semicolon separated list of method declarations. The type of a method underlays certain restrictions.

The domain must be either `SELF` or of the form `SELF` $\times T_1 \times \cdots \times T_n$. The T_i can be either `SELF` or arbitrary type expressions without any occurrence of `SELF`. In other words: Any occurrence of `SELF` in one T_i must not be covered by a type constructor.

The codomain can be an arbitrary type expression with `SELF` occuring only with positive variance.

Methods are classified according to the number of `SELF` occuring in the domain and codomain:

Number of `SELF`		
domain	codomain	classification
1	0	unary pseudo attribute
> 1	0	n–ary pseudo attribute
1	> 0	unary method
> 1	> 0	n–ary method

1.6.4. Constructors

classconstructorsection ::= `CONSTRUCTOR` *classconstructor*
{ ; *classconstructor*}

classconstructor ::= *identifier* : `SELF`
| *identifier* : *type* `->` `SELF`

The constructor section of a class starts with the keyword `CONSTRUCTOR` and contains a semicolon separated list of constructor declarations. A constructor denotes either a constant of the state space of the class (first form) or it is a function with codomain `SELF` (second form). In the latter `SELF` is not allowed to occur in the domain of the constructor. Some object–oriented languages allow constructors of a class to take also arguments of the class itself (so called copy constructors in C++ for instance). In the context of CCSL such constructors are regarded as methods.

1.6.5. Assertions

vardeclaration ::= `SELFVAR` *identifier* : `SELF`
| `VAR` *datavardeclaration* {; *datavardeclaration*}

datavardeclaration ::= *identifier* {, *identifier*} : *type*

assertionsection ::= `ASSERTION` {*importing*} {*vardeclaration*}
assertion {*assertion*}

assertion ::= *identifier* : {*vardeclaration*} `PVS_FORMULA`

Assertions can be used to restrict the behavior of the methods and attributes of the class. This section starts with the keyword `ASSERTION` and basically contains a list of named assertions. The optional importing and variable declaration can be used to extend the set of allowed identifiers in the formula. An importing makes the definitions of the named PVS file available. All importings after the keyword `ASSERTION` are assertion-importings.

A variable declaration leads to a all–quantified closure of the formula. The variable declared with `SELFVAR` must be used to refer to the current state. If multiple `SELFVAR` apply to one formula, the last one is in effect. If no `SELFVAR` applies, the variable "x" must be used for the current state inside the formula. The variable declarations after `ASSERTION` or the optional importings apply to all assertions in this assertion section. The others are only local for the specific assertion. For convenience a sequence of variable declarations is allowed. Note that there is no semicolon allowed behind the last variable of the last variable declaration.

> ***Question:*** *I consider dropping the variable declarations of single assertions. Any comments?*

Inside a formula the following identifiers might be used:

- Any identifier from the PVS prelude
- any identifier declared in one of the PVS files which occur in an importing
  - in any assertion section of this class or its ancestor classes
  - of the class or one of its ancestor classes
- the variables declared in the variable declarations
- the methods and attributes of this class and all its ancestors
- the identifier "bisim?" denoting the bisimilarity relation on the current class

1.6.6. Creation conditions

creationsection	::=	`CREATION` {*importing*} {*vardeclaration*} *creationcondition* {*creationcondition*}
creationcondition	::=	*identifier* : {*vardeclaration*} `PVS_FORMULA`

The creation section is very much like the assertion section. The only differences are:

- it is used to restrict the behavior of the elements delivered by a constructor
- a declaration of `SELFVAR` is not allowed in the variable declarations of the creation section

- the importings after the keyword `CREATION` are creation–importings.

- additionally the constructors of this class are valid identifiers[5]

1.7. Datatype Specifications

adtspec	::=	`BEGIN` *identifier* [*parameterlist*] `: ADT`
		{*adtsection*}
		`END` *identifier*
adtsection	::=	*adtconstructorsection* [;]

Datatype specifications can be used to specify (initial) algebraic datatypes like lists in CCSL. Their syntax is very similar to class specifications. The header consists of the same elements, with the same meaning, except that the keyword `ADT` replaces `CLASSSPEC`. In contrast to class specifications the body of datatype specifications can only contain constructor declarations. For convenience a list of constructors can have a final semicolon.

Each datatype specifications defines a corresponding type. Initial semantics is always used (see section 2.2

1.7.1. Datatype constructors

adtconstructorsection	::=	`CONSTRUCTOR` *adtconstructor* {; *adtconstructor*}
adtconstructor	::=	*identifier* `: CARRIER`
	|	*identifier* [*adtaccessors*] `:` *type* `-> CARRIER`
adtaccessors	::=	(*identifier* {, *identifier*})

A constructor is either a constant (in the carrier type of the initial algebra) or a function taking values in the carrier. For each constructor a recognizer is generated by appending a question mark to its name. If a accessor list is given, their length must correspond to the arity of domain. If no accessor list is given the compiler generates one. For a constructor c taking one argument, the accessor is c_acc. If c takes more arguments, the accessors are numbered c_acc1, c_acc2, ...

[5]under the compiler switch "-ic" (inherit constructors): the constructors of this class and all its ancestor classes; see 3.2

1.8. Source Files

declaration	::=	*classspec*
	|	*adtspec*
	|	*typedef*
	|	*local*
local	::=	`BEGIN` `LOCAL` *declaration* {*declaration*}
		`IN` *declaration* {*declaration*}
		`END` `LOCAL`
file	::=	*declaration* {*declaration*} `EOF`

A whole source file is a mixed list of class specification or Adt specifications possibly nested in an local construct. The local construct makes the types declared in the first section available in the second one, but hides them from the declarations following the local construct. The types of the second section are available behind the local construct.

> ***Question:*** *Does anybody has an example for using local? I suggest to drop it.*

2. The Semantic of CCSL

This chapter describes the PVS theories the compiler generates for class specifications and Adt specifications given in CCSL. While some parts are (almost) constant, others depend highly on the input (the fields of the class, the inheritance hierarchy) and the command line options (see Section 3.2) given to the compiler. In the following I will refer to the latter as switches and set them in `typewriter`. So that "…in case of switch `-records` …" should read as "in case the compiler is invoked with command line option `-records` …".

For each class specification and for each Adt specification the compiler generates a series of theories organized in several files as described in Section 3.1. The compiler uses identifiers from the CCSL source to disambiguate the output. In the following angle brackets will refer to items of the CCSL source, so ⟨class⟩ stand for a class specification and ⟨meth⟩ for a method. The context will explain, which classes and methods have to be substituted. In Section 2.1 ⟨class⟩ (in Section 2.2 ⟨adt⟩) will refer to the class (Adt) specification in question.

In describing the theories I will not only need identifiers from the source but also their types. CCSL types map rather trivially to PVS: base types are mapped by their name, for classes depending on switch `-final` either loose (see section 2.1.6) or final (see Section 2.1.7) semantics is used. For Adt's initial semantics is always used.

> ***Question:*** *Separate section for semantics of types?*

Assume the current class ⟨class⟩ (or the current Adt ⟨adt⟩) has a field f. In the following I will need not only its name "f" but also its declared type. It should be clear that where I describe an identifier (say the invariant lemma for f, "inv_f") I obviously refer to the name of f. And when f occurs in a type expression I obviously refer to the (translated) type of f. Where needed I will emphasize this distinction by using `typewriter` whenever I refer to the name and *italic* whenever I refer to the type. I will use the latter orthogonal to the pattern notation ⟨…⟩. Therefore ⟨`meth`⟩ are the names of some methods, where ⟨*meth*⟩ are their types. On other places where I refer to the whole method (as a syntactic category of CCSL) I will write ⟨meth⟩.

> ***Question:*** *Does anybody have ideas for improving the notion and the last paragraph?*

- explain structure of tables

- explain application of coalgebra in `invariant?(c)`

- explain notion for inherited methods and methods directly declared in a class

2.1. Class Specifications

2.1.1. Interface

The interface theory is the basis of the semantic description. It defines the functor ⟨`class`⟩`IFace` corresponding to ⟨class⟩ and the array of constructors. Further it extracts the methods, attributes and constructors from a coalgebra.

Name ⟨`class`⟩`Interface`

Parameter `Self` + parameters for ⟨class⟩

Imports

- all class–importings
- for all Adt specifications ⟨adt⟩ which are components of ⟨class⟩ the theory ⟨`adt`⟩
- for all class specifications ⟨comp⟩ which are components of ⟨class⟩ the theory `Loose`⟨`comp`⟩`Class` (loose semantics) or `Final`⟨`class`⟩ (final semantics)
- for all direct ancestor classes ⟨anc⟩ the theory ⟨`anc`⟩`Interface`

Definitions

⟨class⟩`IFace`	TYPE	Functor for ⟨class⟩. This is a product (a PVS record for switch `-records`) of [a] • ⟨anc⟩`IFace` for each direct ancestor of ⟨class⟩ • curry(⟨m⟩) for each method or attribut ⟨m⟩ of ⟨class⟩ In case of switch `-records` the label names are derived from the corresponding fields.
⟨`m`⟩ [b]	curry(⟨m⟩)	rename the projections of ⟨`class`⟩`IFace` with method names
⟨`class`⟩`Constructors`[c]	TYPE	Array of constructors. This is a product of the type of all constructors and (only under switch `-ic`) ⟨`anc`⟩`Constructors` for each direct ancestor ⟨anc⟩ of ⟨class⟩ which has constructors.
⟨`m`⟩`(c)`	⟨m⟩ [d]	Definition of attribute or method ⟨m⟩ of ⟨class⟩ or any of its ancestor classes wrt. a coalgebra c.

Notes

[a] The order of the product components is not specified at the moment. A future version will follow the order of the CCSL source.

[b] These definitions simulate a record and therefore they are only present, if ⟨`class`⟩`IFace` is a (plein) product.

[c] ⟨`class`⟩`Constructors` is only present, if ⟨class⟩ has any constructors.

[d] The full type of ⟨`m`⟩ is (`Self` → ⟨`class`⟩`IFace`) → ⟨m⟩

Purpose This theory enables the user to define a class by supplying definitions[1]

```
c : [Self -> ⟨class⟩IFace] = ...

z : ⟨class⟩Constructors = ( ... )
```

and to write `m(c)( ... )` for calls of any method or attribute which is declared in ⟨class⟩ itself or inherited from some other class.

[1] The CCSL compiler outputs a file ⟨`class`⟩`_user.pvs` containing a blueprint of these declarations.

2.1.2. Interface_invert

This theory inverts all the defintions form ⟨class⟩`Interface`. Auto–rewriting this theory converts projections back to cononical method calls of the form $m_i(c)(\ldots)$.

Name ⟨`class`⟩`Interface_invert`

Parameter `Self` + parameters for ⟨class⟩

Imports ⟨`class`⟩`Interface`

Definitions

⟨`d`⟩ [a]	LEMMA	stating the equality of ⟨`d`⟩$(c)(\ldots)$ with its definition from ⟨`class`⟩`Interface`

Notes

[a] The compiler resolves name clashes between lemma names by appending numbers if necessary.

Purpose The strategy

```
(then*
  (auto-rewrite-theory "⟨class⟩[Self, ...]")
  (assert))
```

rewrites method calls to their standard definition and enables (possibly) other rewrites.

2.1.3. Aux

Name ⟨`class`⟩`Aux`

Parameter `Self` + parameters for ⟨class⟩

Imports

Definitions

Notes

[a]

Purpose

2.1.4. Public Invariants

This theory formalizes the notion of public invariants, i.e. predicates on `Self` which are closed under the application of public methods. It contains the definition of invariant, rewrite lemmas exploiting invariants and three standard results about invariants.

Name ⟨`class`⟩`Invariant`

Parameter `Self` + parameters for ⟨class⟩

Imports

- ⟨`class`⟩`Interface`
- for all direct ancestor classes ⟨anc⟩ the theory ⟨`anc`⟩`Invariant`

Definitions

⟨`class`⟩`Pred`	[a]	predicate lifting wrt. the public part of ⟨class⟩
`invariant?(c)`	(`Self` → bool) → bool[b]	Definition of public invariants
`inv_`⟨`p_meth`⟩	LEMMA	invariant property[c] for all public methods ⟨p_meth⟩ of ⟨class⟩
`inv_true`	LEMMA	the truth predicate on `Self` is an public invariant
`inv_and`	LEMMA	the conjunction of two public invariants is an public invariant
`inv_list`	LEMMA	the finite conjunction of finitely many public invariants is an public invariant

Notes

[a] The type is (`Self` → bool) → (⟨`class`⟩`IFace` → bool), that is ⟨`class`⟩`Pred` delivers a predicate on ⟨`class`⟩`IFace` for any input predicate on `Self`.

[b] The full type of invariant? is (`Self` → ⟨`class`⟩`IFace`) → (`Self` → bool) → bool.

[c] The invariant property states, that if P is a public invariant for coalgebra c and p_meth a public method, then P holds on all successor states of p_meth$(c)(x)$ for all $x \in$ `Self`

Purpose Assuming a predicate P, a coalgebra c, and a state of an object $x \in$ Self

```
P_is_inv : Lemma invariant?(c)(P)
```

```
x_in_P : Lemma P(x)
```

states that for all object states in P (namely x) it holds that whatever public method is applied, the resulting state(s) remain(s) in P.

The lemmas inv_⟨p_meth⟩ can be used to establish a predicate on a successor state. This is easier than unraveling the definition of public invariants.

2.1.5. Semantic

2.1.6. Loose

2.1.7. Final

2.2. ADT specifications

3. The CCSL compiler

This chapter introduces the CCSL Compiler and its user interface. The compiler itself is at the moment not publicly available. But you can use it through a Web interface at URL ...

3.1. Invocation

The compiler accepts several options and one source file as command line arguments in arbitrary order. On default it reads one (CCSL) source file and produces several PVS files in the current directory:

`ccsl`	$*$	Semantic of the prelude
`adt`		Semantic for each Adt specification `adt`
`class_basic`		Semantic for each class specification `class`
`class_user`	$*$	simple pattern for consistency proofs for class specification `class`

Files marked with $*$ are only generated if not yet present in the current directory The compiler generates PVS source as well as proof scripts.[1] If a file is already present the compiler tries to rename it by appending a tilde $\sim$.

Bug!

predicates.pvs

3.2. Compiler switches

The compiler accepts the following command line options. They are sorted for relevance of a LOOP user. Options only of interest for compiler developers come last.

`-v` Verbose. Print compiler version and other messages.

`-j` Java. Parse Java source instead of CCSL source.

`-final` Use final semantics for components and generate additional theories for final semantics, see section 2.1.1 and 2.1.7.

`-ic` Inherit constructors, see section 2.1.1 and 2.1.5.

`-l` Generate assertions without introduction of the coalgebra, see section 2.1.5.

[1] Implementing robust proofs is not a main concern in the project, therefore the compiler generates proofs only for some lemmas

`-records` Use a PVS record to define the interface functor instead of a (unlabeled) product, see section 2.1.1. Note that you can only use `-records` if the class has no overloaded fields, because record labels can not be overloaded in PVS.

`-class spec` Generate only files for the named specifications. This option can be given several times.

`-debug` Run the parser in debug mode. This option is at the moment the only support for finding syntax errors in the source. Each token which is recognized by the parser (including line ends) produces a line "`SHIFT token`" on standard error. Most of the reductions done by the parser produce a line "`REDUCE rule`". Note that the grammar given in Appendix A is not LALR. The compiler uses a different but equivalent grammar, which can be found on the Web. Add URL here

`-syntax` Prints part of internal syntax tree on standard output after parsing the source.

`-c` Print all PVS source on standard output instead generating files. Do not generate proofs. Bug!

`-d dir` Output files in `dir` instead of the current directory.

`-symbol` Prints the global symbol table on standard output after parsing.

`-check` Only perform syntax and variance check. Do not generate any PVS output.

`-prf` Only generate proof scripts.

`-edit-proof lemma` Only generate the named proof on standard output.

3.3. The Prelude

Some popular types like lists are predefined in a prelude. The prelude is prepended to each source file, such that the definitions in the prelude are available in the source file. The prelude is

---

```
Begin list[ X : Type ] : Adt
    constructor
        null : Carrier;
        cons( car, cdr ) : [X, Carrier] -> Carrier
end list

Begin Lift[ X : Type ] : Adt
    constructor
        bot : Carrier;
```

```
        up( down ) : X -> Carrier
end Lift

Begin Coproduct[ X : Type, Y : Type ] : Adt
    constructor
        in1(out1) : X -> Carrier;
        in2(out2) : Y -> Carrier;
end Coproduct
```

The lists are directly modeled by the internal lists of PVS. The type Lift[X] adds a new element (namely bot) to the type X. It can be used to model partial functions or exceptions. The Coproduct type corresponds to the disjoint union.

Question: *What about sequences, nat, int, real ...?*

For all declarations in the prelude only one file `ccsl.pvs` is generated, see section 3.1.

A. The CCSL Grammar

```
file             ::=  declaration {declaration} EOF

declaration      ::=  classspec
                   |  adtspec
                   |  typedef
                   |  local

local            ::=  BEGIN LOCAL declaration {declaration}
                      IN declaration {declaration}
                      END LOCAL

typedef          ::=  TYPE identifier : BASETYPE
                   |  TYPE identifier = type

classspec        ::=  BEGIN identifier [parameterlist] : CLASSSPEC
                      {importing} {classsection}
                      END identifier

classsection     ::=  inheritsection
                   |  [visibility] attributesection [;]
                   |  [visibility] methodsection [;]
                   |  classconstructorsection [;]
                   |  assertionsection
                   |  creationsection

visibility       ::=  PUBLIC
                   |  PRIVATE

inheritsection   ::=  INHERIT FROM ancestor {, ancestor}

ancestor         ::=  identifier [argumentlist]
                      [RENAMING renaming {AND renaming}]

renaming         ::=  identifier [: type] AS identifier
```

attributesection	::=	`ATTRIBUTE` *attribute* {; *attribute*}
attribute	::=	*identifier* : `SELF ->` *type*
methodsection	::=	`METHOD` *method* {; *method*}
method	::=	*identifier* : *type* `->` *type*
classconstructorsection	::=	`CONSTRUCTOR` *classconstructor* {; *classconstructor*}
classconstructor	::=	*identifier* : `SELF`
	|	*identifier* : *type* `-> SELF`
assertionsection	::=	`ASSERTION` {*importing*} {*vardeclaration*} *assertion* {*assertion*}
assertion	::=	*identifier* : {*vardeclaration*} `PVS_FORMULA`
creationsection	::=	`CREATION` {*importing*} {*vardeclaration*} *creationcondition* {*creationcondition*}
creationcondition	::=	*identifier* : {*vardeclaration*} `PVS_FORMULA`
vardeclaration	::=	`SELFVAR` *identifier* : `SELF`
	|	`VAR` *datavardeclaration* {; *datavardeclaration*}
datavardeclaration	::=	*identifier* {, *identifier*} : *type*
adtspec	::=	`BEGIN` *identifier* [*parameterlist*] : `ADT` {*adtsection*} `END` *identifier*
adtsection	::=	*adtconstructorsection* [;]
adtconstructorsection	::=	`CONSTRUCTOR` *adtconstructor* {; *adtconstructor*}
adtconstructor	::=	*identifier* : `CARRIER`
	|	*identifier* [*adtaccessors*] : *type* `-> CARRIER`
adtaccessors	::=	(*identifier* {, *identifier*})

parameterlist	::=	`[` *parameter* {, *parameter*} `]`
parameter	::=	*identifier* {, *identifier*} : `TYPE`
	|	*identifier* {, *identifier*} : *type*
type	::=	*identifier*
	|	`SELF`
	|	`CARRIER`
	|	`[` *type* {, *type*} `->` *type* `]`
	|	`[` *type* {, *type*} `]`
	|	*identifier argumentlist*
argumentlist	::=	`[` *argument* {, *argument*} `]`
argument	::=	*type*
	|	*expr*
importing	::=	`IMPORTING` *string*
expr	::=	`VALUE`
	|	*identifier*

B. Index