GSDLAB TECHNICAL REPORT

Configurator Semantics of the CDL language

Yingfei Xiong

GSDLAB–TR 2011–06–05 Jun 2011

Generative Software Development Laboratory
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

WWW page: http://gsd.uwaterloo.ca/

Configurator Semantics of the CDL language

Yingfei Xiong

Jun 2011

Abstract

This paper presents formal semantics of the Component Description Language (CDL) language. Compared to the CDL semantics proposed by Berger and She [BS10], this version focuses more on the behavior of configurator and is more close to the implementation of a configurator.

1 Introduction

The Component Description Language [VD] is a variability modeling language for embedded languages. Its semantics is first formalized by Berger and She [BS10]. Their semantics mainly focuses on the configuration space, and it is still difficult to understand the behavior of a configurator and implement a configurator based on their semantics. Specifically, there are the following problems if we try to understand the behavior of a configurator from their semantics:

- Their semantics allows cyclic references. Cyclic references mean that there could be arbitary equations in the code, and the configurator has to solve these equations to understand the really allowed value. For example, there could be the following code.

  ```
  cdl_option x {
    flavor data
    calculated (x * x + 1)/2
  }
  ```

  Keyword `calculated` means the value of the feature is calculated from the expression following. However, as the expression cyclically refer to `x` again, we cannot directly calculate the value. Berger and She's semantics considers the expression as a constraint, and by equation solving we can get `x=1` in this case. However, equation solving is impossible for arbitrarily complex equations. Also, `calculated` means that the value of the feature cannot be edited by the user and should be determined by the configurator. If there is an equation have no solution or more than one solution, their semantics gives no rule of determining the value of `x` in those situations. The current eCos configurator also does not support cyclic references, and will freeze in most cases where cyclic references are presented.

- Their semantics does not distinguish the configuration errors from disabled features. In CDL, users could declare two types of constraints on a feature: `active_if` and `requires`. When the feature is selected and a `requires` constraint is violated, the configuration contains an error and the configurator should report this error. When an `active_if` constraint is violated, the declaring feature is automatically disabled (removed from the configuration), and no error is reported. Although the two types of constraints have the same effect in constraining the configuration space, they exhibit quite different behavior in the configurator and should be distinguished in configurator semantics.
- Their semantics does not clearly show which values are user-editable. We have seen that calculated features are not user-editable. There are also many other types of features that are not user-editable. Berger and She's semantics does not provide clear information about this.

This paper presents a new version of CDL semantics. Compared to Berger and She's version [BS10], this version focuses on the configurator behavior and addresses the above issues. In addition, this version presents the full semantics of operators and functions, which was not presented in Berger and She's version [BS10].

2 Synatx

A CDL model consists of a set of packages, where each package is defined by a set of programs in the CDL language. A package can be loaded to a model or unloaded from a model at configuration time. So an important question is whether we model this dynamic behavior of package changes. In this paper we focus on only a fixed set of packages, and leaving dynamic change of packages to future work.

To ease the definition of semantics, we consider an abstract syntax instead of the concrete syntax of CDL. The abstract syntax is obtained from Berger and She's semantics [BS10].

Let *Data* is the union of the set of strings, integers and floats, $String \subset Data$ be the set of strings, and $ID \subset String$ be the set of all possible feature identifiers. A CDL model is a tuple consisting of the following components:

- $Id \subseteq ID$, the set of feature identifiers defined in the model,
- $parent : Id \to Id \bigcup \{\top\}$, a function mapping an identifier to its parent, where $\top$ means the top feature,
- $kind : Id \to \{\texttt{package}, \texttt{component}, \texttt{option}, \texttt{interface}\}$, a function mapping an identifier to its base kind,
- $flavor : Id \to \{\texttt{none}, \texttt{bool}, \texttt{booldata}, \texttt{data}\}$, a function mapping an identifier to its flavor,
- $activeIf : Id \to 2^{Exp(ID)}$, a function mapping an identifier to its `active_if` expressions,

- *requires* : $Id \to 2^{Exp(ID)}$, a function mapping an identifier to its `requires` expressions,
- *calculated* : $Id \to Exp(ID) \bigcup \{\bot\}$, a function mapping an identifier to its `calculated` expression, or $\bot$ if no `calculated` is defined,
- *legalValues* : $Id \to LExp(ID) \bigcup \{\bot\}$, a function mapping an identifier to its `legal_values` expression, , or $\bot$ if no `legal_values` is defined,
- *implements* : $Id \to 2^{Id}$, a function mapping an identifier to the interfaces it implements, and
- *version* : $Id \to String$, a function mapping an a package to its version (denoted by a string), or $\bot$ is the feature is not a package

where *Exp(ID)* is the set of goal expressions defined by the following grammar:

$$
\begin{array}{lll}
e := & id & \text{where } id \in ID \\
\mid & const & \text{where } const \in Data \\
\mid & e \oplus e & \text{where } \oplus \in \{\texttt{||}, \texttt{\&\&}, \texttt{implies}, \texttt{eqv}, \texttt{xor}\} \\
\mid & e \otimes e & \text{where } \otimes \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{<<}, \texttt{>>}, \texttt{\^{}}, \texttt{\&}, \texttt{.}\} \\
\mid & e \oslash e & \text{where } \oslash \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\} \\
\mid & \texttt{!}e & \\
\mid & \tilde{}e & \\
\mid & e?e : e & \\
\mid & \texttt{get\_data}(id) & \text{where } id \in ID \\
\mid & \texttt{is\_active}(id) & \text{where } id \in ID \\
\mid & \texttt{is\_enabled}(id) & \text{where } id \in ID \\
\mid & \texttt{is\_loaded}(id) & \text{where } id \in ID \\
\mid & \texttt{is\_substr}(e, e) & \\
\mid & \texttt{is\_xsubstr}(e, e) & \\
\mid & \texttt{version\_cmp}(v_1, v_2) & \text{where } v_1, v_2 \in String
\end{array}
$$

and *LExp(ID)* is the set of list expressions defined by the following grammar:

$$
\begin{array}{lll}
le & := & le\_unit\ le \mid le\_unit \\
le\_unit & := & e \mid e\ \texttt{to}\ e \qquad \text{where } e \in Exp(ID)
\end{array}
$$

.

We write $a.f$ for $f(a)$ to reduce the number of parentheses.

There are also a few well-formedness rules over the syntax of the model. For any feature $x \in Id$, the following constraints must be satisfied. The explanation of these constraints can be found in [BS10].

- $x.\mathit{flavor} = \texttt{none} \longrightarrow x.\mathit{calculated} = \bot$
- $(x.\mathit{calculated} \neq \bot \vee x.\mathit{flavor} = \texttt{bool}) \longrightarrow x.\mathit{legalValues} = \bot$
- $x.\mathit{kind} = \texttt{interface} \longrightarrow (x.\mathit{flavor} \neq \texttt{none} \wedge x.\mathit{calculated} = \bot)$
- $x.\mathit{kind} \neq \texttt{package} \longrightarrow x.\mathit{version} = \bot$
- $\exists y, y.\mathit{parent} = x \longrightarrow x.\mathit{kind} \neq \texttt{option}$

Also, for the whole feature model, the following two constraints should be satisfied.

- The *parent* relationship should form a tree, with $\top$ as the root.
- References in *Exp(ID)* and *LExp(ID)* should form no cycles (either directly or indirectly).

3 Semantics

Let *Bool* be set $\{0, 1\}$. Given a CDL model (*Id*, *parent*, *kind*, *flavor*, *activeIf*, *requires*, *calculated*, *legalValues*, *implements*, *version*), configurator semantics explains the model to two sets: *Var* and *Constraint*, where *Var* is a set of variables typed over *Bool* or *Data*, and *Constraint* is a set of constraints in Tcl language. A configuration of the model is an assignment to *Var* conforming to the types of the variables. A correct configuration is an assignment over which all constraints evaluate to 1. The variable set also present the values that is user-editable. The configurator should present the variables as editable fields in the user interface. Constraint set *Constraint* also gives the granularity of error reporting. If a constraint in *Constraint* evaluates to 0, the configurator should report an error.

To simplify the definitions, we define the semantics in two steps. First, let us consider features that is not calculated and is not an interface. In CDL, each feature has a boolean value and a data value. Depending on the flavor of the features and whether the feature is calculated, some of values are user-editable and some of the values are derived. The user-editable values are mapped to variables. We define the variable set as a union of denotation $Var[\![x]\!]$ over each feature x. We first give the definition where $x.calculated = \bot \wedge x.kind \neq \texttt{interface}$.

$$Var = \bigcup_{x \in Id} Var[\![x]\!]$$

$$Var[\![x]\!] = \begin{cases} \{\} & x.flavor = \texttt{none} \\ \{\texttt{x\_bool} : Bool\} & x.flavor = \texttt{bool} \\ \{\texttt{x\_data} : Data\} & x.flavor = \texttt{data} \\ \{\texttt{x\_bool} : Bool, \texttt{x\_data} : Data\} & x.flavor = \texttt{booldata} \end{cases}$$

To access the two values of a feature, we define another two denotations. Denotation $enabled[\![x]\!]$ returns a Tcl expression for accessing the Boolean value of a feature, and we first give its definition over feature x where $x.calculated = \bot \wedge x.kind \neq \texttt{interface}$.

$$enabled[\![x]\!] = \begin{cases} 1 & x.flavor = \texttt{none} \vee x.flavor = \texttt{data} \\ \texttt{x\_bool} & x.flavor = \texttt{bool} \vee x.flavor = \texttt{booldata} \end{cases}$$

Similarly, denotation $data[\![x]\!]$ returns the data value of feature x. We give its definition where $x.calculated = \bot \wedge x.kind \neq \texttt{interface}$ as below.

$$data[\![x]\!] = \begin{cases} 1 & x.flavor = \texttt{none} \vee x.flavor = \texttt{bool} \\ \texttt{x\_data} & x.flavor = \texttt{data} \vee x.flavor = \texttt{booldata} \end{cases}$$

A feature in CDL can also be active or inactive. When a feature is inactive, its is disabled on the configurator interface and the user cannot change its variable(s). This behavior is modeled as denotation $active[\![x]\!]$.

$$active[\![x]\!] = pActive[\![x]\!] \;\texttt{\&\&}\; eActive[\![x]\!]$$

$$pActive[\![x]\!] = \begin{cases} 1 & x.parent = \top \\ effective[\![x.parent]\!] & x.parent \in Id \end{cases}$$

$$eActive[\![x]\!] = \begin{cases} 1 & x.activeIf = \bot \\ expr[\![x.activeIf]\!] & x.activeIf \neq \bot \end{cases}$$

Denotation $\mathit{effective}[\![]\!]$ is used to determine whether a feature is included in a configuration or not. When a feature is not in the configuration, it is not considered in the code generation, and we will always get zero when we try to access its value through CDL expressions. A feature is included in a configuration when it is both enabled and active.

$$\mathit{effective}[\![x]\!] = \mathit{active}[\![x]\!]\ \texttt{\&\&}\ \mathit{enabled}[\![x]\!]$$

Denotation $\mathit{expr}[\![]\!]$ converts a CDL expression into a Tcl expression. It is defined below.

$$
\begin{array}{lcll}
\mathit{expr}[\![id]\!] & = & \mathit{effective}[\![x]\!]?\mathit{data}[\![x]\!] : 0 & \text{if } id \in \mathit{Id} \\
\mathit{expr}[\![id]\!] & = & 0 & \text{if } id \in \mathit{ID} \wedge id \notin \mathit{Id} \\
\mathit{expr}[\![const]\!] & = & const & \text{if } const \in \mathit{Data} \\
\mathit{expr}[\![e_1 \oplus e_2]\!] & = & \mathit{expr}[\![e_1]\!] \oplus \mathit{expr}[\![e_2]\!] & \\
\multicolumn{4}{l}{\quad \text{if } \oplus \in \{\texttt{||}, \texttt{\&\&}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{<<}, \texttt{>>}, \texttt{\^{}}, \texttt{\&}, \texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\}} \\
\mathit{expr}[\![e_1\ \texttt{implies}\ e_2]\!] & = & \texttt{!}\mathit{expr}[\![e_1]\!]\ \texttt{||}\ \mathit{expr}[\![e_2]\!] & \\
\mathit{expr}[\![e_1\ \texttt{eqv}\ e_2]\!] & = & \mathit{expr}[\![e_1]\!]\texttt{\&\&}\mathit{expr}[\![e_2]\!]\ \texttt{||}\ \texttt{!}\mathit{expr}[\![e_1]\!]\texttt{\&\&!}\mathit{expr}[\![e_2]\!] & \\
\mathit{expr}[\![e_1\ \texttt{xor}\ e_2]\!] & = & \texttt{!}\mathit{expr}[\![e_1\ \texttt{eqv}\ e_2]\!] & \\
\mathit{expr}[\![e_1\texttt{.}e_2]\!] & = & \texttt{concat}\ \mathit{expr}[\![e_1]\!]\ \mathit{expr}[\![e2]\!] & \\
\mathit{expr}[\![\texttt{!}e]\!] & = & \texttt{!}\mathit{expr}[\![e]\!] & \\
\mathit{expr}[\![\texttt{\~{}}e]\!] & = & \texttt{\~{}}\mathit{expr}[\![e]\!] & \\
\mathit{expr}[\![e_1\texttt{?}e_2\texttt{:}e_3]\!] & = & \mathit{expr}[\![e_1]\!]\texttt{?}\mathit{expr}[\![e_2]\!]\texttt{:}\mathit{expr}[\![e_3]\!] & \\
\mathit{expr}[\![\texttt{get\_data}(id)]\!] & = & \mathit{data}[\![id]\!] & \text{if } id \in \mathit{Id} \\
\mathit{expr}[\![\texttt{get\_data}(id)]\!] & = & 0 & \text{if } id \in \mathit{ID} \wedge id \notin \mathit{Id} \\
\mathit{expr}[\![\texttt{is\_active}(id)]\!] & = & \mathit{active}[\![id]\!] & \text{if } id \in \mathit{Id} \\
\mathit{expr}[\![\texttt{is\_active}(id)]\!] & = & 0 & \text{if } id \in \mathit{ID} \wedge id \notin \mathit{Id} \\
\mathit{expr}[\![\texttt{is\_enabled}(id)]\!] & = & \mathit{enabled}[\![id]\!] & \text{if } id \in \mathit{Id} \\
\mathit{expr}[\![\texttt{is\_enabled}(id)]\!] & = & 0 & \text{if } id \in \mathit{ID} \wedge id \notin \mathit{Id} \\
\mathit{expr}[\![\texttt{is\_loaded}(id)]\!] & = & 1 & \text{if } id \in \mathit{Id} \\
\mathit{expr}[\![\texttt{is\_loaded}(id)]\!] & = & 0 & \text{if } id \in \mathit{ID} \wedge id \notin \mathit{Id} \\
\mathit{expr}[\![\texttt{is\_substr}(e_1, e_2)]\!] & = & \mathit{substr}(\texttt{concat " "}\ \mathit{expr}[\![e_1]\!]\ \texttt{" "}, \mathit{expr}[\![e_2]\!]) & \\
\mathit{expr}[\![\texttt{is\_xsubstr}(id)]\!] & = & \mathit{substr}(\mathit{expr}[\![e_1]\!], \mathit{expr}[\![e_2]\!]) & \\
\mathit{expr}[\![\texttt{version\_cmp}(v_1, v_2)]\!] & = & \texttt{string compare}\ \mathit{toVer}(v_1)\ \mathit{toVer}(v_2) &
\end{array}
$$

$$
\text{where} \quad \mathit{toVer}(v) = \begin{cases} v.version & v \in \mathit{Id} \\ v & v \notin \mathit{Id} \end{cases}
$$

$$
\mathit{substr}(e_1, e_2) = \texttt{(string first}\ e_2\ e_1\texttt{) >= 0}
$$

We have seen how *Var* is defined and a few auxiliary denotations for accessing different states of features. Now let us see how *Constraint* is defined. Basically, *Constraint* is translated from the `requires` constraints and `legal_values`.

$$\mathit{Constraint} = \bigcup_{x \in \mathit{Id}} (\mathit{reqConstrs}[\![x]\!] \cup \mathit{legalValConstrs}[\![x]\!])$$

where $\mathit{reqConstrs}[\![x]\!]$ and $\mathit{legalValConstrs}[\![x]\!]$ are defined as follows.

$$
\begin{array}{lcl}
\mathit{reqConstrs}[\![x]\!] & = & \{\mathit{effective}[\![x]\!] \rightarrow \mathit{expr}[\![req]\!] \mid req \in x.\mathit{requires}\} \\
\mathit{legalValConstrs}[\![x]\!] & = & \{\mathit{lexpr}[\![x, x.legalVal]\!] \mid x.\mathit{legalValues} \neq \bot\}
\end{array}
$$

Denotation $lexpr[\![]\!]$ converts a list expression into a Tcl expression.

$$\begin{array}{lcl} lexpr[\![x, le\_unit\ le]\!] & = & lexpr[\![x, le\_unit]\!]lexpr[\![x, le]\!] \\ lexpr[\![x, e_1\ \texttt{to}\ e_2]\!] & = & expr[\![x]\!]\ \texttt{>=}\ expr[\![e_1]\!]\ \texttt{\&\&}\ expr[\![x]\!]\ \texttt{<=}\ expr[\![e_2]\!] \\ lexpr[\![x, e]\!] & = & expr[\![x]\!]\ \texttt{=}\ expr[\![e]\!] \end{array}$$

Now we come to the second step of semantics definition and are ready to take into account calculated features and interfaces. These two types of features both have their values determined by other features and are not user-editable. The value of a calculated feature is determined by its `calculated` expression. The value of an interface is the number of implementing features that are effective. To take them into our definitions, we need to complete three previously partial definitions: $Var[\![]\!]$, $enabled[\![]\!]$, and $data[\![]\!]$. But before that, let us defined a denotation for the expression returning the calculated value of a feature. For any feature x where $x.calculated \neq \bot \vee x.kind = \texttt{interface}$, we have the following definition.

$$calculatedVal[\![x]\!] = \begin{cases} expr[\![x.calculated]\!] & x.calculated \neq \bot \\ \Sigma_{\forall y, x \in y.implements} effective[\![y]\!]?1:0 & x.kind = \texttt{interface} \end{cases}$$

Now we are ready to complete our definitions. For calculated features and interfaces, we create no variables.

$$Var[\![x]\!] = \{\} \qquad \text{if } x.calculated \neq \bot \vee x.kind = \texttt{interface}$$

Their boolean values and data values are determined by their calculated values and their flavors. For any feature x where $x.calculated \neq \bot \vee x.kind = \texttt{interface}$, we have the following definitions.

$$enabled[\![x]\!] = \begin{cases} 1 & x.flavor = \texttt{data} \\ calculatedVal[\![x]\!]\texttt{?1:0} & x.flavor = \texttt{bool} \vee x.flavor = \texttt{booldata} \end{cases}$$

$$data[\![x]\!] = \begin{cases} 1 & x.flavor = \texttt{bool} \\ calculatedVal[\![x]\!] & x.flavor = \texttt{data} \vee x.flavor = \texttt{booldata} \end{cases}$$

Putting the two parts together, we have the full semantics of the CDL language.

4 Related Work

The semantics in this paper is used in the analysis of real world CDL constraints [PNX+11]. A simplified description of the semantics is also presented in that paper. To ease understanding, some of the denotations in this paper is presented as variables in that paper. For example, $active[\![n]\!]$ appears as a variable, `n_active`, and a new constraint converted from the definition of $active[\![n]\!]$ is added. So that descriptions gives more variables and constraints, but the essential configuration spaces are the same.

5 Conclusion

This paper has presented the configurator semantics of CDL, additionally modeling the configurator behavior that is not presented in [BS10]. Specifically, the three problems mentioned in the introduction have been addressed as follows.

- Cyclic references are explicitly disallowed on the syntax level.
- Each semantic constraint corresponds to a configuration error. The `active_if` constraints are not directly translated into semantic constraints (but their effect on the configuration space is kept).
- Each semantic variable corresponds to a user-editable value. No user-uneditable variable is created and the user cannot change anything beyond the variables.

Acknowledgment

Thanks to Thorsten Berger and Steven She for their first version of CDL semantics, and to Thorsten Berger for his insightful comments on the draft of this paper.

References

[BS10] Thorsten Berger and Steven She. Formal semantics of the cdl language. `www.informatik.uni-leipzig.de/~berger/cdl_semantics.pdf`, 2010.

[PNX+11] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. A study of non-boolean constraints in variability models of an embedded operating system. In *FOSD'11: 3rd International Workshop on Feature-Oriented Software Development*, 2011.

[VD] B. Veer and J. Dallaway. The eCos component writer's guide. `ecos.sourceware.org/ecos/docs-latest/cdl-guide/cdl-guide.html`.