CML2 Language and Tools Description

The Kernel Configuration Menu Language

Eric Steven Raymond

Thyrsus Enterprises (http://www.tuxedo.org/~esr)

esr@thyrsus.com

This paper describes CML2, the Configuration Menu Language designed to support Linux kernel configuration. It is written as a historical narrative because that provides a good frame for describing the design issues in the language.

Why CML2?

“ Every program eventually becomes rococo, and then rubble. ”

Alan Perlis

The CML2 project was launched because in March 2000, in the process of building 2.3.51 kernels, I discovered that the original kbuild configure language had reached the rococo stage; it had become so brittle that any attempt to change or extend it would break everything.

This matters, because the kernel-configuration process has grown excessively complex. The configuration system’s job is to turn the user’s answers to configuration questions into a file of `#define` constructs used to condition features in or out of the C code. As Linux has grown more features and more driver support, the number of menus and prompts one must navigate to choose the appropriate subset of those features has become forbidding even to expert users, and outright impossible for the novice users Linux increasingly seeks to attract.

To properly manage the complexity we have created, we need a configuration interface that can support better chunking and progressive disclosure. Ideally, novices building kernels should see only the choices they *need* to make. As they gain competence and cue the configuration system that they are ready, they

should see more options for tuning and exotic hardware. The full process should be accessible to expert kernel hackers, but not inflicted willy-nilly on everyone else as it now is.

With the brittle old configure language (which we'll call CML1) this kind of redesign would simply not have been possible at all. CML1 had started out life as a set of simple shellscripts, but evolved into a massive hairball featuring three different interpreters, C code customized to generate Tcl/Tk, and source files so difficult to read that (according to the unanimous report of CML1's own maintainers) latter-day CML1 users rarely try to program it by any means more sophisticated than a cut-paste-edit of the menus for existing features.

I had a second reason for getting involved besides wanting to see the configuration process simplified; I like designing little languages more than any other kind of programming. The interesting challenge I saw was to design a language that would (a) capture all the right domain-specific abstractions, but (b) remain flexible enough to allow configuration-system maintainers to experiment with different progressive-disclosure policies.

Design Problems in CML2's Domain

Achieving both a good map of the territory and sufficient flexibiility was not an easy problem. My first straw-man design ("Thesis", for purposes of this paper) was essentially a cleaned-up, stripped-down CML1. The process of hand-translating a few hundred lines of the roughly CML1 corpus into Thesis showed me that a conventional imperative language would not be adequate for this problem.

The cause of the mismatch is that most of the complexity in the CML1 corpus expresses neither actions nor values, but rather visibility constraints. Most of the program's decision points have to do with whether the question that sets a given configuration symbol needs to be asked at all. The logic for deriving a final set of configuration `#define` symbols from the answers to the subset of questions a user actually answers is comparatively trivial. Its most important part is the checking of various constraints on combinations of configuration options.

This description strongly suggests that what was really needed was a language not for describing menu actions but rather for *declaring rules*. This should work with an interpreter that would query the user according to those rules in the process of building a correct solution.

Seen in this light, CML1's rigidity and complexity was partly a direct result of trying to do declarative things with imperative machinery. My second straw-man design, which I'll call "Antithesis" here, was a short-lived attempt to do away with the concept of explicit configuration menus entirely. In Antithesis, all queries would have been driven by a process that started from a set of initial conditions (such as specifying the processor architecture and the user's expertise level) and got to a valid configuration end state by something like theorem-proving.

I say "would have been" and "something like" because Antithesis never got beyond the concept stage. I quickly realized that the ability to group questions into explicit menus and specify the order of menus

was a valuable way to chunk the problem domain, to convey a mental model of the relationships between different configuration options.

Design and Philosophy of CML2

CML2, therefore, has a view of the world that includes both menus and rules. A CML2 system specifies the following things:

- A tree of menus. This tree is explicitly declared by the CML2 programmer. Each menu contains a sequence of questions; each answer to a question sets the value of a configuration symbol. Menus may also contain submenus.
- A set of visibility predicates. At any given time, only a subset of the menu tree is visible. Each node in the tree (each symbol or menu) has a predicate associated with it that controls whether it is visible or suppressed. Input (independent) variables in the predicate may be set from CML2's startup options, or they may be set by the results of previous queries.
- A set of validity constraints. These are predicates that connect two or more configuration symbols. Each constraint is checked every time the user tries to change the value one of its input symbols; if the constraint would not be satisfied by the change, the change is disallowed and the user notified. Other, user-interface-dependent actions to recover from the constraint violation may follow.
- A set of derivations. A derivation is a formula that ties the value of an output symbol to the value of one or more input symbols. Derived symbols may be either be computed for direct use in a final configuration, or used as input variables of constraints or predicates. Whenever a derived symbol is evaluated, the formula behind it is recomputed from the current values of the formula's input symbols (like a cell in a spreadsheet).

This is a very different view of the world from CML1's conventional imperative one, and will have significant implementations for configuration file maintainers. Notably, the CML1 configuration tree had multiple apices: one top-level config-language file for each port subdirectory. CML2 defines one big menu tree, with portions suppressed during the configuration process.

Some indication of the power of these concepts may be gleaned from the compression ratio of CML1-to-CML2 translation. The 7049 lines of CML1 in the 2.3.99-pre9 kernel tree became fewer than 2400 lines of CML2, and it would have been less if I had not been adding sanity checks and reorganizing it as I went along.

CML2 Language Elements

Syntax

Lexically, a CML program consists tokens: of barewords, whitespace, strings and punctuation (there is one exception associated with the `icon` declaration). A bareword is a token composed of alphanumeric characters and _ (underscore). Whitepace is any mix of spaces, tabs, and linefeeds. A string is delimited by either single or double quotes and may contain whitespace. Everything else is punctuation. Some pairs of punctuation characters stick together; ==, !=, <=, >=. All other punctuation is treated as single-character tokens.

Here are lexical rules, regular expressions describing valid tokens:

```
<symbol>      ::= [A-Z][A-Za-z0-9_]*
<menu-id>     ::= [a-z][a-z0-9_]*
<string>      ::= '[^']*'|&dquot;[^&dquot;]*&dquot;
<decimal>     ::= [0-9]+
<hexadecimal> ::= 0x[A-Fa-f0-9]+
```

base64-data is any number of lines in RFC2045 base64 format, terminated by a newline or comment.

Also, note that there is a lexical-level inclusion facility. The token “source” is interpreted as a request to treat the immediately following token as a filename (string quotes are first stripped if it is a string). Upon encountering this directive, the compiler opens the file (which may be a relative or absolute pathname), and read input from that file until EOF. On that EOF, the current input source is immediately popped back to the including file.

Comments are supported, and run from a # to end-of-line.

Here is a BNF of the grammar. Following it, each language element will be described in detail.

```
;; A few things we need to define up front...
;;
<tritval> ::= 'y' | 'm' | 'n' ;; Yes, no, or module
<name>    ::= <menu-id>|<symbolname> ;; A bareword composed of alphanumerics
<int>     ::= <decimal> | <hexadecimal> ;; Integer literal

;; A CML system consists of a sequence of declarations.
;;
<system> ::= <declaration>*

;; A declaration may be of one of the following types:
;;
<declaration> ::= <source-statement>
```

```
                |   <symbols-declaration>
        |   <menus-declaration>
        |   <helpfile-declaration>

        |   <private-declaration>
        |   <visibility-rule>
        |   <menu-definition>
        |   <radio-definition>
        |   <derive-definition>
        |   <default-definition>
        |   <requirement-definition>

        |   <start-definition>
        |   <prefix-definition>
        |   <banner-definition>

        |   <options-definition>
        |   <condition-declaration>
        |   <warndepend-declaration>
        |   <icon-definition>
        |   <debug-definition>

;; A source statement declares a file to be inserted in place of
;; the source statement in the ruleset.
;;
<source-statement> ::= 'source' <string>

;; A symbols definition creates configuration symbols,
;; and associates prompt strings with them.
;;
<symbols-declaration> ::= 'symbols' {<symbol> <string>}*

;; A menus definition creates menu symbols,
;; and associates banner strings with them.
;;
<menus-declaration> ::= 'menus' {<menu-id> <string>}*

;; A helpfile declaration declares a file to be scanned for help text
;;
<helpfile-declaration> ::= 'helpfile' <word>

;; A private declaration declares that the associated symbols should
;; not be exported in the end-result configuration
;;
<private-declaration> ::= 'private' <symbol>*
```

```
;; A visibility rule associates a visibility predicate with symbols.
;; Optionally, it may declare that the suppressed symbols are constrained
;; in value by the predicate symbols.
;;
<visibility-rule> ::=  'unless' <logical> 'suppress' ['dependent'] <name>*;

;; A menu definition links a sequence or subtree of symbols with a
;; menu identifier.  Subtrees generate implied dependency statements.
;;
<menu-definition> ::= 'menu' <menu-id> <name-or-subtree>*
<name-or-subtree> ::= <name> <suffix>
  |   <name> <suffix> '{' <name-or-subtree>* '}'

<suffix> ::= ;; Empty suffix declares boolean type
 | '?' ;; declares trit type
 | '%' ;; declares decimal type
 | '@' ;; declares hexadecimal type
 | '$' ;; declares string type

;; A radio-menu definition links a choice of symbols with a menu identifier.
;;
<radio-definition> ::= 'choices' <menu-id> <symbol>* 'default' <symbol>

;; A derivation binds a symbol to a formula, so the value of that
;; symbol is always the current value of the formula.
;;
<derive-definition> ::= 'derive' <symbol> 'from' <expr>

;; A default definition sets the value a symbol will have unless it is
;; explicitly set by the user's answer to a question. It may have a
;; range specification attached.
;;
<default-definition> ::= 'default' <symbol> 'from' <expr>
                                  ['range' {<int> | {<int> '-' <int>}}+]

;; A requirement definition constrains the value of one or more symbols
;; by requiring that the given expression be true in any valid configuration.
;;
<requirement-definition> ::=  {'require'|'prohibit'} <logical>

;; We have to declare a start menu, for the beginning of execution
;;
<start-definition> ::= 'start' <menu-id>
```

```
;; A prefix definition sets a string to be prepended to all symbols
;; when they are named in a configuration file.
;;
<prefix-definition> ::= 'prefix' <string>

;; A banner definition sets a string to used in the configurator
;; greeting line.
;;
<banner-definition> ::= 'banner' <menu-id>

;; An option definition sets command-line options for the configurator
;;
<options-definition> ::= 'options' <word>*

;; A condition statement ties a &CML2; control flag to a symbol
;;
<condition-declaration> ::= 'condition' <word> 'on' <configsymbol>

;; A warndepend flags symbols that make dependents dangerous
;;
<warndepend-declaration> ::= 'warndepend' <name>*

;; An icon definition associates data for a graphic icon with the
;; rulebase.
;;
<icon-definition> := 'icon' <base64-data>

;; A debug definition enables debugging output.
;;
<debug-definition> ::= 'debug' <decimal>

;; An expression is a formula
;;
<expr> ::= <expr> '+' <expr>
        | <expr> '-' <expr>
| <expr> '*' <expr>
| <logical>

<logical> ::= <logical> 'or' <logical>
  | <logical> 'and' <logical>
  | <logical> 'implies' <logical>
  | <relational>

<relational> ::= <term> '==' <term>
 | <term> '!=' <term>
```

```
 | <term> '<=' <term>
 | <term> '>=' <term>
 | <term>
 | 'not' <relational>

<term> ::= <term> '|' <term> ;; maximum or sum or union value
       | <term> '&' <term> ;; minimum or multiple or intersection value
       | <term> '$' <term> ;; similarity value
       | <atom>

<atom> ::= <symbol>
        | <tritval>
        | <string>
        | <decimal>
        | <hexadecimal>
| '(' <expr> ')'
```

Operators have the precedence implied by the above productions. From least to most binding, precedence classes are:

```
1: + -
2: *
3: implies
3: or
4: and
5: not
6: ==, !=, >=, <=, >, <
7: &, |, $
```

Data types and classes

CML2 supports the following data types:

- Booleans. These may have the values ‘y’ or ‘n’
- Tristates or trits. These may have the values ‘y’, ’m’, or ‘n’
- Decimal integers. 32-bit signed integers with decimal I/O formatting.
- Hexadecimal integers. 32-bit signed integers with hexadecimal I/O formatting.
- Strings. Strings are literal data in the ASCII character set encoding.

Support for trits may be disabled at runtime. See the section called *Condition statement* for discussion of the condition/on declaration.

There are four classes of symbols; constant symbols, query symbols, derivation symbols, and frozen symbols.

A constant is one of the boolean/tristate literals `y` or `m` or `n`, or an integer literal, or a string literal.

A query symbol is an ordinary, mutable symbol with a prompt string. Each query must occur exactly once in the menu tree. Query symbols may be set by the user.

A derivation is a symbol bound to an expression. Derivation symbols are immutable, but may vary as the symbols in their formula change value. Derived symbols have no associated prompt string and may not appear in the menu tree.

A frozen symbol is a query symbol which has been immutably bound to a particular value. Once frozen, the value of a symbol may not be changed.

Meaning of the language elements

Source statements

A source statement declares a file to be inserted in place of the source statement in the file, and treated as if the entire contents of that file were present in the current file at the point of their source statement.

Any implementation of CML2 must allow source statements to be nested to a depth of at least 15 levels. The reference implementation has no hard limit.

Symbol declarations

The body of a symbols section consists of pairs of tokens; a configuration symbol and a prompt string.

Rationale: Having these name-to-prompt associations be separate from the dependency rules will help make the text parts of the system easier to localize for different languages. Declaring all query symbols up front means we can do better and faster sanity checks. Some symbols (derivations) are not pre-declared.

Menu declarations

The body of a menus section consists of pairs of tokens; a menu name and a banner string. The effect of each declaration is to declare an empty menu (to be filled in later by a menu definition) and associate a banner string with it.

Any implementation of CML2 must allow menus to be nested to a depth of at least 15 levels. The reference implementation has no hard limit.

Rationale: Having these menu-to-banner associations be separate from the dependency rules will help make the text parts of the system easier to localize for different languages. Declaring all menu names up front means we can do better and faster sanity checks.

Helpfile declarations

A helpfile declaration tells the compiler to mine a given file for help texts. The compiler's assumption is that the file is in the format of a CML1 help file: entries are begun by two lines, the first containing a prompt string and the second beginning with the string `CONFIG_`.

The format of helpfiles may be changed in future releases.

Private declarations

A private declaration sets the *`private`* bit on each symbol of a list of symbol names. Symbols on which this bit is set are not written to the final configuration file.

Rationale: Sometimes you may want to make multiple queries, the results of which are not used directly in the configuration file but become independent variables in the derivation of a symbol that is used. In this kind of case, it is good practice to make the query symbols private.

Visibility rules

A visibility declaration associates a visibility predicate with a set of configuration symbols. The fact that several symbols may occur on the right side of such a rule is just a notational convenience; the rule

```
unless GUARD suppress SYMBOL1 SYMBOL2 SYMBOL3
```

is exactly equivalent to

```
unless GUARD suppress SYMBOL1
unless GUARD suppress SYMBOL2
unless GUARD suppress SYMBOL3
```

Putting a menu on the right side of a visibility rule suppresses that menu and all its children.

Dependence

Optionally, a rule may declare that the suppressed symbols are constrained in value by the predicate symbols. That is, if there is a rule

```
unless GUARD suppress dependent SYMBOL
```

then the value of `SYMBOL` is constrained by the value of `GUARD` in the following way:

```
guard trit bool
--- ---- -----
  y      y,m,n y,n
  m      m,n y,n
  n      n n
```

The reason for this odd, type-dependent logic table is that we want to be able to have boolean option symbols that configure options for modular ancestors. This is why the guard symbol value `m` permits a dependent boolean symbol (but not a dependent modular symbol) to be `y`.

If the guard part is an expression, `SYMBOL` is made dependent on each symbol that occurs in the guard. Such guards may not contain alternations or 'implies'. Thus, if `FOO` and `BAR` and `BAZ` are trit symbols,

```
unless FOO!=n and BAR==m suppress dependent BAZ
```

is equivalent to the following rules:

```
unless FOO!=n and BAR==m suppress BAZ
require BAZ <= FOO and BAZ <= BAR
```

Putting a menu on the right side of a visibility rule with 'dependent' puts the constraint on all the configuration symbols in that menu. Any submenus will inherit the constraint and pass it downward to their submenus.

Dependency works both ways. If a dependent symbol is set `y` or `m`, the value of the ancestor symbol may be forced; see the section called *Symbol Assignment and Side Effects* for discussion.

Rationale: The syntax is unless...suppress rather than if...query because the normal state of a symbol or menu is visible. The dependent construction replaces the dep_tristate and dep_bool constructs in CML1.

Menu definitions

A menu definition associates a sequence of configuration symbols and (sub)menu identifiers with a menu identifier (and its banner string). It is an error for any symbol or menu name to be referenced in more than one menu.

Symbol references in menus may have suffixes which change the default boolean type of the symbol. The suffixes are as follows:

```
?      trit type
%      decimal type
```

```
@      hexadecimal type
$      string type
```

A choices definition associates a choice of boolean configuration symbols with a menu identifier (and its banner string). It declares a default symbol to be set to `y` at the time the menu is instantiated.

In a complete CML2 system, these definitions link all menus together into a single big tree, which is normally traversed depth-first (except that visibility predicates may suppress parts of it).

If the list of symbols has subtrees in it (indicated by curly braces) then the symbol immediately before the opening curly brace is declared a visibility and dependency guard for all symbols within the braces. That is, the menu declaration

```
menu foo
     SYM1 SYM2 {SYM3 SYM4} SYM5
```

not only associates SYM[12345] with foo, it also registers rules equivalent to

```
   unless SYM2 suppress dependent SYM3 SYM4
```

Such subtree declarations may be nested to any depth.

It is perfectly legal for a menu-ID to have no child nodes. In CML2, this is how you embed text in menus, by making it the banner of of a symbol with no children.

Derivations

A derivation binds a symbol to a formula, so the value of that symbol is always the current value of the formula. Symbols may be evaluated either when a menu containing them is instantiated or at the time the final configuration file is written.

The compiler performs type inference to deduce the type of a derived symbol. In particular, derived symbols for which the top-level expression is an arithmetic operator are deduced to be decimal. Derived symbols for which the top level of the expression is a boolean operator are deduced to be bool. Derived symbols for which the top level of the expression is a trit operator are deduced to be trit.

Derived symbols are never set directly by the user and have no associated prompt string.

Defaults

A default definition sets the value a symbol will have until it is explicitly set by the user’s answer to a question. The right-hand side of a default is not limited to being a constant value; it may be any valid expression.

Defaults may be evaluated either when a menu containing them is instantiated or at the time the final configuration file is written.

If a symbol is not explicitly defaulted, it gets the zero value of its type; `n` for bools and trits, `0` for decimal and hexadecimal symbols, and the empty string for strings.

The optional range part may be used to constrain legal values for decimal or hexadecimal-valued symbol. A range specification consists of any number of either single values or paired upper and lower bounds separated by a dash, interpreted as inclusive ranges. The symbol has a legal value if it either matches a specified single value or is contained in one of the intervals.

Requirements

Requirements as sanity checks

A requirement definition constrains the value of one or more symbols by requiring that the given expression be true in any valid configuration. All constraints involving a given configuration symbol are checked each time that symbol is modified. Every constraint is checked just before the configuration file is written.

It is up to individual CML2 front ends to decide how to handle constraint violations. Here are some possible policies:

- Complain and die. Not recommended, but perhaps appropriate for a batch-mode front end.
- Conservative recovery: Disallow the modification that would violate the constraint. (Thus, earlier answers have priority over later ones.)
- Flag-and-continue: visibly flag all symbols involved in a constraint violation (and unflag them whenever a constraint violation is fixed). Require the user to resolve all constraint violations before the configuration file is saved.
- Backtracking: Present all the menus involved in the constraint. Accept modifications of any of them, but do not allow the modifications to be committed until all constraints are satisfied.

A prohibit definition requires that the attached predicate *not* be true. This is syntactic sugar, added to accommodate the fact that human beings have troouble reasoning about the distribution of negation in complex predicates.

Using requirements to force variables

Requirements have a second role. Certain kinds of requirements can be used to deduce values for variables the user has not yet set; the CML2 interpreter does this automatically.

Every time a symbol is changed, the change is tried on each declared constraint. The constraint is algebraicly simplified by substituting in constant, derived and frozen symbols. If the simplified constraint forces an expression of the form A == B to be true, and either A is a query symbol and B is a constant or the reverse, then the assignment needed to make A == B true is forced.

Thus, given the rules

```
derive SPARC from SPARC32 or SPARC64
require SPARC implies ISA==n and PCMCIA==n and VT==y and VT_CONSOLE==y
and BUSMOUSE==y and SUN_MOUSE==y and SERIAL==y and SERIAL_CONSOLE==y
and SUN_KEYBOARD==y
```

when either SPARC32 or SPARC64 changes to `y`, the nine assignments implied by the right side of the second rule will be performed automatically. If this kind of requirement is triggered by a guard consisting entirely of frozen symbols, all the assigned symbols become frozen.

If A is a boolean or trit symbol and B simplifies to a boolean or trit constant (or vice-versa), assignments may be similarly forced by other relationals (notably A != B, A < B, A > B, A <= B, and A >= B). If forcing the relational to be true implies only one possible value for the symbol involved, then that assignment is forced.

Note that whether a relational forces a unique value may depend on whether trits are enabled or not.

Start declaration

The start definition specifies the name of the root menu of the hierarchy. One such declaration is required per CML2 ruleset.

Prefix declaration

A prefix declaration sets a string to be prepended to each symbol name whenever it is written out to a result configuration file. This prefix is also stripped from symbol names read in in a defconfig file.

Rationale: This was added so the CML2 rule system for the Linux kernel would not have to include the common `CONFIG_` prefix. The alternative of wiring that prefix into the code would compromise CML2's potential usefulness for other applications.

Banner declaration

A banner definition sets the menu id banner string to used in the configurator greeting line. The string attached to the specified menu id should identify the system being configured.

Rationale: As for the prefix string.

Options

Note: The options statement sets options which will be passed to the configurator instance as though they had been specified on the command line of the option. These options are processed before actual command-line options.

Note that switch indicators beginning with - will need to be string-quoted to avoid being broken up by the lexical analyzer.

Rationale: this will typically be used to pre-set the locations of configurator output files.

Condition statement

The condition statement ties a CML2 feature flag to a query symbol; that is, the value of the feature flag is the value of the symbol. The initial value of the flag when a rulebase is read in is simply the associated symbol's default. If there is no symbol associated with the the flag, the flag's value is `n`.

At present only one flag, named "trits", is supported. When this flag is `n`, trit-valued symbols are treated as booleans and may only assume the values `y` and `n`.

This flag may affect the front end's presentation of alternatives for modular symbols. It also affects forcing of ancestor symbols. When the trits flag is on, setting a boolean symbol only forces its trit ancestors to the value `m`; when trits is off, they are forced to `y`. See the section called *Symbol Assignment and Side Effects* for discussion.

Warndepend declaration

The warndepend declaration takes a list of symbol names. All dependents of each symbol have their prompts suffixed with the name of the symbol in parentheses to indicate that they are dependent on it.

Rationale: Added to support the EXPERIMENTAL symbol in the Linux lernel configuration. This declaration is better than tracking experimental status by hand because it guarantees that subsidiary symbols dependent on an experimental feature will always be flagged for the user.

Icon declaration

An icon declaration associates graphic data (encoded in RFC2045 base64) with the rulebase. Front ends may use this data as an identification icon. All front ends are required to accept XPM data here.

The reference front-end implementation uses the image to iconify the configurator when it is minimized while running in X mode. The reference front-end also accepts GIF data.

Debug

This declaration enables debugging output from the compiler (it has no effect on front-end behavior). It takes an integer value and uses it to set the current debug level. It may change or be removed in future releases.

Expressions

All arithmetic is integer. The compiler permits some kinds of type promotion, described below.

For purposes of the relational operators, trit values are strictly ordered with `y` > `m` > `n`.

Boolean logical expressions may be used as parts of integer-valued formulas (e,g in derivations and constraints). The value of true is 1, and of false is zero.

It is a compile-time error to apply the logical operators or/and/implies to trit or numeric values. Also, expressions occuring in guards (in unless/suppress, or require/prohibit declarations) must yield a value of boolean type. The compiler does type propagation in expressions to check these constraints.

The purpose of these restriction is to enable compile-time detection of situations where confusion of trit or numeric with boolean values might induce subtle errors. For the same reason, if the symbol FOO is trit-valued it is a compile-time error to say just “FOO” in an expression, as opposed to “FOO!=n” or some other more explicit relational.

Thus, because the symbol SCSI is trit-valued:

```
unless SCSI suppress A2091_SCSI
```

is illegal and will raise an error. Write an unambiguous test instead:

```
unless SCSI>=m suppress A2091_SCSI
```

The obvious booleans operations (and, or) are supported; they are commutative and associative. An ’implies’ operation is also supported:

```
FOO implies BAR  <=>  not (FOO and (not BAR))
```

It is neither commutative nor associative.

The usual relational tests (==, !=, >=, <=, >, <) are supported. Relationals bind more tightly than boolean operators, so FOO!=n and BAR==m behaves as expected. Additionally, and binds more tightly than or, so that FOO or BAR and BAZ is FOO or (BAR and BAZ).

The following additional ternary-logic operations are available. It is an error to apply these to operands with types other than bool or trit.

Max or union; notation |. Here is a truth table:

```
    y m n
  +----
y | y y y
m | y m m
n | y m n
```

Min or intersection; notation &. Here is a truth table:

```
    y m n
  +----
y | y m n
m | m m n
n | n n n
```

Similarity; notation $. Here is a truth table:

```
    y m n
  +----
y | y n n
m | n m n
n | n n n
```

The operator precedence implied by the BNF above is implemented in the parser.

Symbol Assignment and Side Effects

Setting a symbol's value may have side effects on other symbols in two ways.

First, it may trigger a change in other variables through explicit requirements. See the section called *Using requirements to force variables* for discussion.

Second, each symbol has two implicit lists associated with it: of symbols it depends on (ancestors) and symbols that depend on it (dependents). Whenever a symbol is changed, any side effects are propagated through those lists. Changing the value of the symbol upward (n⇝m, m⇝y) may change the value of ancestors; changing it downard (y⇝m, m⇝n) may affect the value of dependents.

See also the section called *Dependence* for discussion of the two syntactically different ways dependencies can be created, and section V for discussion of the deduction algorithm.

CML2 interpreters are required to implement all-or-nothing side effects; that is, after an assignment, either the assignment and all its side effects have been performed, or (in the event the new values would violate a requirement) none of them are.

The reference implementation achieves this by implementing two-phase commit; the assignment and its side effects can be made tentatively, constraints checked, and then either committed or rolled back.

Side-effect bindings remain linked to the symbol whose value change triggered them, and are backed out whenever that symbol is changed again. Backing out a side effect may expose previous side effects on a symbol. To see how this works, consider the following sequence of actions given the constraints (FOO==y implies BAR==y) and (BAZ==y implies BAR==n):

```
1. User sets FOO=y.  As a side effect, this sets BAR=y
2. User sets BAZ=y.  As a side-effect, this sets BAR=n
3. User sets BAZ=n.  This does not have a direct side-effect on BAR.
   However, since the value BAZ has changed, its side effect BAR=n
   is backed out.  The value of BAR is again y.
```

The reference implementation journals all side-effects and always looks for the most recent binding of a symbol when evaluating it.

Implementation of CML2

Deduction algorithm

CML2 is *not* built around an algorithm for the propositional satisfiability (or SAT) problem , such as GSAT or SATO. Given that CML2 is a constraint-based language, this might at first seem curious. But there are special features of CML2's domain that would make these relatively poor choices and difficult to apply.

The Linux kernel configuration problem for which CML2 was originally invented and tuned involves approximately 1750 symbols. Of these, however, fewer than 400 participate in about the same number of constraints, mostly implied constraints of simple (a <= b) form. However, many of the variables are not booleans but tristates, blowing the equivalent problem back up to about 1600 boolean shadow variables over 400 constraints, and making the translation of the problem into pure boolean-propositional form a significant exercise in itself.

Complete SAT algorithms like SATO are very time-consuming (the problem is NP-complete), enough to cause unacceptable lag in an interactive configurator on a problem this size. Incomplete SAT algorithms like the stochastic GSAT technique are not guaranteed to yield an answer even though one may exist. But

there is a more fundamental problem: we do not actually want a “model” in the SAT sense; where variables are underconstrained we want to leave them as don’t-cares (e.g. not set them.)

(Good references on the SAT problem are available on the Web. SATO (http://www.cs.uiowa.edu/~hzhang/sato.html) is the best-of-breed among complete SAT algorithms and the GSAT Page (http://www.owlnet.rice.edu/~tdanner/gsat/) includes a Python implementation, and some good discussion of the algorithm’s limitations. Michael Littman has also assembled a mini-survey of the literature (http://www.cs.duke.edu/~mlittman/topics/sat.html).)

CML2 instead uses a relatively simple custom algorithm independently invented by the author, and more closely related to the resolution method of the original Davis-Putnam elimination algorithm than to the splitting-rule approach of SATO and other modern SAT techniques. The CML2 algorithm exploits facts about the domain. One is the fact that we don’t actually need to find a full model. Another is the fact that many of the constraints (about 3/4 in the Linux-kernel problem) are simple chains of the form x1 <= x2 <= x3...<= xn created by suppress dependent declarations or sub-menu bracketing with {}.

When the value of a symbol is changed, CML2 tries to find variables it can force. It does this by simplifying the value of all frozen, “chilled”, and tentatively set variables out of constraints. In any conjunction that this simplification leaves, the code looks for relationals with a constant on one side and a mutable symbol on the other. When these relationals constrain the symbol to a single value, that value is forced and the symbol is marked “chilled”.

The assignment of the forced symbol is itself done using the same algorithm. Redundant assignments are ignored; an attempt to set a chilled symbol means the ruleset has inconsistent constraints and raises a fatal error.

Implementation

The reference implementation of CML2 is written in Python. This choice has an obvious virtue and a couple of non-obvious ones.

The obvious virtue is that Python is a a truly high-level language that does memory management automatically, eliminating the single most common source of bugs in languages without this property. However, several other non-obvious virtues are equally important. Here are some of them:

- “Batteries are included.” While Python does not have as rich an extension-module base as its main competitor Perl, rather more capability is bundled with the stock Python interpreter. One built-in facility that is particularly important for CML2 is Python’s “pickle” or object-serialization support. A CML2 rulebase is a pickled object.
- Python, unlike other scripting languages, can be (effectively) compiled to pure C using the **freeze** facility. The translation is not pretty, and produces rather large C programs from even small Python sources, but it does meet the problem of portability head-on. Kernels could be shipped with a

precompiled rulebase and a frozen C version of the CML2 interpreter to avoid the requirement for Python.

- Another non-obvious virtue is the way that Python supports conditional runtime loading of support modules. We can use this to detect and recover from situations in which the library support does not exist to provide Tcl-based or curses-based front end.

Note: the CML2 implementation uses two modules that are not part of stock Python. One is an enhanced version of shlex.py slated to enter the stock environment in Python 1.6. The other is John Aycock's SPARK toolkit, used for expression parsing. Copies of `shlex.py` and `spark.py` are shipped with the CML2 implementation.

Running the CML2 tools

The CML2 implementation consists of two Python programs: **cmlcompile** and **cmlconfigure**. The compiler, **cmlcompile**, generates a pickled representation of a rulebase from one or more files of CML2 rules. The interpreter, **cmlconfigure**, reads in a rulebase and uses it to inform a configuration dialogue with the user.

The separation between front and back ends serves two purposes. One: Front ends don't need to know about and are not affected by the details of the compiler implementation. Two: The CML2 rulebase doesn't have to be recompiled every time a front end is run.

The end result is a pair of configuration files, the `defconfig` and the `macrofile`. These are in formats inherited from CML1 (see the section called *CML2 configuration file formats* for complete specification). The `defconfig` consists of a set of variable definitions in shell syntax; it can be re-read by a future instance of cmlconfigure to set defaults. The macrofile is a list of C-style `#define` macros intended to be included in kernel C code.

The compiler, **cmlcompiler**, requires one or more CML2 rules files as arguments. It has two options:

`-o filename`

Set the file to which the compiled rulebase is written. By default, if no -o option is given, the rulebase goes to `rules.out`.

`-d`

Enable debugging output to `stdout`.

The interpreter, **cmlconfigure**, takes at most one filename argument: the rulebase to be interpreted. If no argument is specified, it reads from `rules.out`. The interpreter recognizes the following options:

`-h outfile`

Set the location to which **cmlconfigure** should write its file of C macros. There is no default; if there is no `-h`, option no macro file is written.

`-iconfigfile`

Read in a configuration. The file is expected to be in the same defconfig format written by **cmlconfigure**. Values (including `n`) are set as though selected interactively by the user.

`-Iconfigfile`

Read in a configuration. The file is expected to be in the same defconfig format written by cmlconfigure. Values (including `n`) are frozen and will be displayed but not modifiable during the configure run.

`-l`

List. Run in batch mode to generate a menu map (that is, a tree diagram of all the menus and question symbols in the system.)

`-s outfile`

Set the location to which cmlconfigure should write its defconfig file of shell variable settings. Explicit n values are saved. This file will be loadable by cmlconfigure. There is no default; if there is no `-s` option no defconfig file is written.

`-t`

Force tty (line-oriented) mode.

`-c`

Force curses (screen-oriented) mode.

`-x`

Force X (GUI using Tk) mode.

`-d`

Increment the debug flag.

`-D`

Preset a symbol. `-DFOO` sets FOO=`y` at startup. -DFOO=`xxx` may be used to specify a value. The value of a preset symbol is frozen; that is, it will never be queried.

`-S`

Don’t hide elided symbols. When this option is on, suppressions are ignored. Only symbols that have derived a frozen value from -D or constraints are skipped. This may be useful if you know that you want to set configuration symbols deep in the hierarchy and have their requirements propagate upwards, as opposed to the normal sequence in which you refine your way down from the top of the tree with symbols becoming visible only when they are unmasked by previous questions.

`-V`

Print the configurator version and exit.

The environment variable CML2OPTIONS may specify command-line switches (but not arguments). Switches taken from CML2OPTIONS are processed after any compiled in by an "options" directive in the rulebase, but before switches specified on the actual command line.

The environment variable BROWSER may specify a colon-separated list of browser commands, to be used in making URLs in the help widgets live. Each command should have the string "%s" in it where the URL can be substituted. The CML2 front end will try the commands in succession until one succeeds. The default sequence is mozilla, then any netscape already running, then a new netscape instance, then lynx, then w3m.

CML2 configuration file formats

Defconfig format

A `defconfig` file consists of a series of lines in the format of Unix shell variable assignments. Each line is led with a symbol name (prefixed if there is a prefix declaration in the rulebase), continues with a `=`, and finishes with a value. The value may be `y`, `n`, or `m` for boolean or tristate symbols. It may be a quoted string for string literals, or an (unquoted) numeric literal for decimal or hexadecimal symbols.

Unlike CML1’s tools, CML2 writes explicit FOO=n defconfig lines when a boolean or tristate symbol has been set by the user (or as a side effect of user actions).

When a symbol FOO has not been set, CML2 may emit a comment of the form

```
#CONFIG_FOO is not
set.
```

. In CML1, these comments were also emitted for symbols with the value `n`.

Rationale: The change in treatment of symbols with value n is important in order to allow defconfigs to contain partial configurations that suppress features, e.g. such as might be generated by an autoprobe utility checking for the presence of buses or controller types.

In the reference implementation, symbols are written in depth-first traversal order. Derived symbols are written after the query symbols; each derived symbol is written out if any of the symbols in its formula has been set.

Macrofile format

A macrofile is a series of C #define macro lines corresponding to configuration symbols. The macro corresponding to as symbol has the same name as that symbol, prefixed if there is a "prefix" declaration in the rulebase.

If a boolean or tristate symbol has the value `y`, CML2 generates a line defining the corresponding macro to have the value `1`.

If a boolean or tristate symbol has the value `n`, CML2 generates a line undefining the corresponding macro.

If a boolean or tristate symbol has the value `m`, CML2 generates two lines. The first undefines the corresponding macro. The second defines a macro with the same name and the suffix "_MODULE" to be `1`.

Each decimal and hexadecimal symbol generates a line defining the corresponding macro to the value of the numeric literal.

Each string symbol generates a line defining the corresponding macro to the value of the string, as a C string literal in double quotes.

These format conventions are identical to CML1's.

Current limitations of CML2

There are no composition operators for string types. (These would be trivial to implement, but are not yet needed.)

The compiler objects to uses of question symbols in a visibility-guard expression for a symbol occuring *before* the associated question (in preorder). This could be supported, but such forward declarations make it hard to reason about valid configurations.

Change history

0.1 – 24 May 2000 Original CML2 specification.
0.2.0 – 31 May 2000 Removed whenever/sets from the language. Everything is now done with deduction.
0.2.1 – 4 June 2000 Added warndepend.
0.3.0 – 9 June 2000 Deduction algorithm rewritten and documented in section V.
0.4.0 – 20 June 2000: Value stacking is implemented.
0.5.0 – 26 June 2000: Fuller support for ranges. Spec format converted to DocBook.

Acknowledgements

Credit where credit is due, to the intrepid early adopters on the linux-kernel list who helped CML2 get past its teething stage: David Kamholz, Giacomo Catenazzi, Robert Wittams, Randy Dunlap, Urban Widmark, and André Dahlqvist.