A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System

Leonardo Passos
University of Waterloo
lpassos@gsd.uwaterloo.ca

Marko Novakovic
University of Waterloo
mnovakov@gsd.uwaterloo.ca

Yingfei Xiong
University of Waterloo
yingfei@gsd.uwaterloo.ca

Thorsten Berger
University of Leipzig
tb@informatik.uni-leipzig.de

Krzysztof Czarnecki
University of Waterloo
kczarnec@gsd.uwaterloo.ca

Andrzej Wąsowski
IT University of Copenhagen
wasowski@itu.dk

ABSTRACT

Many variability modeling tasks can be supported by automated analyses of models. Unfortunately, most analyses for Boolean variability models are NP-hard, while analyses for non-Boolean models easily become undecidable. It is thus crucial to exploit the properties of realistic models to construct viable analysis algorithms. Unfortunately, little work exists about non-Boolean models, and no benchmarks are available for such.

We present the non-Boolean aspects of 116 variability models available in the codebase of eCos—a real time embedded operating system. We characterize the types of non-Boolean features in the models, kinds and quantities of non-Boolean constraints in use, and the impact of these characteristics on the hardness of this model from analysis perspective. This way we provide researchers and practitioners with a basis for discussion of relevance of non-Boolean models and their analyses, along with the first ever benchmark for effectiveness of such analyses.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics

General Terms

Measurement

Keywords

Variability Modeling, Feature Models, Decision Models, Automated Model Analysis

1. INTRODUCTION

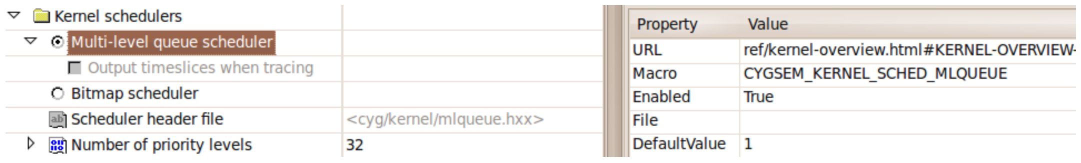
Variability modeling [10, 14] supports feature-oriented software development (FOSD) by enabling (i) understanding and definition of commonalities and variabilities within a product line and (ii) product derivation. Many tasks of variability modeling and management are supported by automated analyses of variability models [2]—among others: diagnosing errors and lesser deficiencies in models, providing metrics about models and their instances, or supporting product derivation. Example analyses include consistency checks, dead feature detection, counting products, interactive guidance during configuration, or fixing models and configurations.

Unfortunately, most analysis algorithms for variability models are NP-hard. This intractability is linked to conciseness of the models, akin to conciseness of logical formulae. For instance, Boolean variability models include Boolean features, or decisions, and propositional Boolean logics constraints over features. A Boolean feature model with n features has $O(2^n)$ possible configurations.

Boolean models are intimately related to Boolean logics [1], thus satisfiability (SAT) checkers are routinely employed in their analyses [1, 6, 9]. Recent evidence [12] suggests that SAT-based analysis of Boolean feature models is easy for realistic models. This observation is in line with the long established understanding in the SAT-community that success relies on exploitation of properties of problem instances that appear in practice [8].

Non-Boolean variability models contain constraints that include—in addition to Boolean formulas and expressions over finite domain variables—expressions over infinite domain variables (integer or float numbers, character strings) together with arithmetic, relational, and string operators. Satisfiability checking of such constraints is undecidable in general. For example, many problems that include integers and reals are undecidable [5]. Thus, for these models it is even more important to exploit the properties of realistic models to construct viable analysis methods. However, little work exists about non-Boolean models, and no benchmarks are available for such.

eCos, an open source real-time operating system for deeply embedded applications [11], provides a set of real-world non-Boolean variability models. The system has been developed originally by Cygnus Solution and RedHat, but later transferred to independent developers who release it under the copyright of the Free Software Foundation. The project supports 14 architectures and 109 cpu types, but can also run inside more than 50 controllers, including flash, Ether-



```
▽ Kernel schedulers
  ▽ ⊙ Multi-level queue scheduler
        Output timeslices when tracing
    ○ Bitmap scheduler
      Scheduler header file          <cyg/kernel/mlqueue.hxx>
  ▷   Number of priority levels      32
```

Property	Value
URL	ref/kernel-overview.html#KERNEL-OVERVIEW-
Macro	CYGSEM_KERNEL_SCHED_MLQUEUE
Enabled	True
File	
DefaultValue	1

Figure 1: ConfigTool: The eCos configurator

net, serial, USB and time-keeping devices. The codebase of eCos contains 116 non-Boolean variability models. Its variability modeling language, The Component Definition Language (CDL), and one of its models have been studied previously [4]. CDL is a textual language that shares many concepts with feature modeling [10] and decision modeling [14]. CDL allows organizing configuration options hierarchically and restricting their possible values and combinations by constraints. Following the feature modeling terminology, we refer to these options as *features*.

This work zooms into the non-Boolean aspects of the 116 CDL models in eCos, extending the prior work [4], by characterizing the types of non-Boolean features available, kinds and quantities of non-Boolean constraints in use, and the impact of these characteristics on the hardness of this model from analysis perspective.

We believe that this work provides researchers and practitioners with the badly needed basis for discussion of relevance of non-Boolean models and their analyses, along with the first ever benchmark[1] for effectiveness of such analyses.

We proceed by presenting the CDL language briefly in Sect. 2. The experimental part of the paper follows directly after. Sect. 3 outlines the method of the experiment (characterization of non-Boolean features and constraints). Sect. 4 summarizes the results. Sect. 5 discusses threats to validity. We finish with a brief survey of related work (Sect. 6) and a conclusion (Sect. 7).

2. OVERVIEW OF CDL

We now briefly summarize the main concepts of CDL, their semantics, and available tool support.

2.1 Configuration and Tooling

CDL is a domain-specific language for modeling legal configurations in a software project. It is accompanied by ConfigTool—a GUI-based configurator that supports users in creating a legal configuration of a given model. The configurator propagates user choices using a custom inference engine.

The main units of functionality are packages. They are archives that bundle code and variability models. Packages are either hardware-specific (part of hardware abstraction layer for an architecture) or contain hardware-independent application and system software. Figure 1 presents a screenshot of the ConfigTool in a state, when a user has enabled a specific kernel scheduler within the Kernel schedulers package, and has set the number of priority levels to 32.

Given one of the 116 hardware architectures (called targets), and one of nine predefined collections (called template, e.g. *default*, *min*, *all*) of hardware-independent packages, the configurator loads a set of packages and aggregates all variability models into a single one. Additional packages with application and system software can be loaded subsequently.

[1]Available at http://gsd.uwaterloo.ca/FOSD11

The process of configuration adheres to the reconfiguration paradigm: the user starts with a default configuration and modifies it stepwise to reach a specific state. After each step, the configurator checks constraints and reports potential conflicts. Finally, the configuration is used to derive a customized instance of eCos—a library of the OS to be linked with boot and application code. For this purpose, the configurator generates C macros that control the conditional compilation of C code.

Figure 3 shows a use of such macros in code. The names of macros correspond to the choices of Fig. 1. By default, one preprocessor macro is created per feature. This can be customized to create several, or no macro at all.

2.2 Feature Representation

CDL is a domain-specific language providing keywords for various *kinds* of features: Packages, Components, Options, and Interfaces. We explain the key CDL concepts using the example in Fig. 2, referring to it by line numbers. These kinds relate largely to implementation artifacts in eCos.

In a CDL variability model, Packages are containers for features (not shown in Fig. 2); Components are nested features grouping other features (l. 1); and Options are atomic configuration options appearing as leaves (l. 8). The display property (l.2) gives the string used by the configurator to show the feature (cf. Fig. 1). Interfaces are invisible in the configurator and used to impose cardinality constraints on other features, for example to realize feature groups (or, xor, mutex), known from feature modeling (l. 21). Features can declare—using the implements property (l. 4)—to implement an interface. The value of an interface is the number of features currently in the configuration implementing it. The interface in l. 21 requires

```
1   cdl_component CYGSEM_KERNEL_SCHED_MLQUEUE {
2    display "Multi−level queue scheduler"
3    default_value 1
4    implements KERNEL_SCHEDULER
5    description "The multi−level queue scheduler supports multiple priority
6                 levels and multiple threads at each priority level..."
7
8    cdl_option TRACE_TIMESLICE {
9     display "Output timeslices when tracing"
10    active_if USE_TRACING
11    requires !DEBUG_TRACE_ASSERT_SIMPLE
12    ...
13   }
14  }
15  cdl_option KERNEL_SCHED_BITMAP {
16      display "Bitmap scheduler"
17      implements KERNEL_SCHEDULER
18      ...
19  }
20
21  cdl_interface KERNEL_SCHEDULER {
22      display "Number of schedulers in this configuration"
23      requires 1 == KERNEL_SCHEDULER
24  }
25  ...
26  cdl_option AT91_CLOCK_SPEED {
27    display "CPU clock speed"
28    calculated { AT91_CLOCK_OSC_MAIN ∗ AT91_PLL_MULTIPLIER / AT91_PLL_DIVIDER / 2 }
29    legal_values { 0 to 220000000 }
30    flavor data
31  }
```

Figure 2: CDL excerpt from eCos variability model

the value to be 1, thus, implementing an xor-group.

Each feature has a *flavor* determining which value types it admits. Flavor none means the feature is just a place holder. Flavor bool means the feature can be selected or unselected (also referred as *enabled* or *disabled*). An option has flavor bool if not specified otherwise—see l. 8 for an example. Flavor data admits a *data value*: an integer number, a float, or a string (l. 30). Flavor booldata combines bool with data: the feature can be enabled or disabled and it admits a data value if enabled. The flavor instructs the configurator to show a checkbox for bool and a field for data and both for booldata. Radio buttons replace checkboxes for features forming xor-groups, such as two scheduler types in Fig. 1.

The data value is dynamically typed. In the eCos configurator, if the user inputs a signed long literal written in decimal, octal or hexadecimal, it is interpreted as an integer. If the number contains a radix point, it is interpreted as a float. Other input is considered as a string. Booleans are denoted by integers: 0 means *false* and 1 means *true*. These types are dynamically converted when needed. For example, an addition of the empty string to the number 2 results in 2, because the empty string is implicitly converted into 0.

2.3 Feature constraints

CDL offers several mechanisms to introduce *feature constraints*, that is, constraints among features. These mechanisms include feature properties, like active_if and default_value; feature nesting (hierarchy); and interfaces. Following [4], we classify them as (1) *configuration constraints*, which restrict combinations and values of features; (2) *visibility conditions*, which control visibility of features in the configurator; and (3) *defaults*, providing default values for users.

We now briefly explain each of the mechanisms.

Active_if represents both a visibility and configuration constraint. If unsatisfied, the feature and all its children are immediately *inactive* (grayed-out and not changeable in the configurator). For example, the option defined in l. 8 (Fig. 2) is inactive in Fig. 1 since USE_TRACING is disabled (not shown) and, thus, active_if in l. 10 is unsatisfied.

Requires represents a configuration constraint (l. 11). The requires condition must hold if the feature is active and enabled. In contrast to active_if, the constraint can be temporarily violated in the configurator (though a conflict is reported) such that the corresponding feature, its children, and dependent features remain editable. The configurator's inference engine generates proposals to fix these violations.

Legal_values is a configuration constraint and restricts the possible values of a feature. This property declares ranges (l. 29) or enumerations (explained later).

Calculated is a configuration constraint and restricts a feature's value to an expression (l. 28), which is re-evaluated by the configurator after each configuration step. Users cannot edit calculated features in the configurator.

Default_values declare default values for the configurator (l. 3). They can be overridden by the user at any time.

Interfaces impose configuration constraints, as described previously.

Feature hierarchy imposes visibility and configuration constraints. When a parent feature is inactive, all its children are inactive.

The first five mechanisms take expressions built from features identifiers, literals, and the following operators and built-in functions (we explain them in parentheses):

```
#ifndef CYGSEM_KERNEL_SCHED_MLQUEUE
  #error POSIX pthreads need MLQ scheduler
#endif
...
// the HAL CDL and the HAL startup code.
fmcn = AT91_CLOCK_SPEED * 1.5 / 1000000 + 0.999999; // We must round up!
```

Figure 3: C code excerpt using CDL options

- Boolean: ! (not), && (and), || (or), implies;
- Relational: == (=), != (≠), <, <= (≤), >, >= (≥);
- Arithmetic: +, -, *, / (division), % (modulo);
- Bit-wise: << (left shift), >> (right shift), & (and), | (or), ^ (xor);
- String: . (concatenation), is_substr (substring check);
- Conditional: *a* ? *b* : *c*;
- Built-in functions: bool (cast into boolean value); is_active and is_enabled (check whether a feature is, respectively, active or enabled), and some more, which do not occur in any of the studied models.

2.4 Semantics

CDL has complex semantics; however, different analyses rely on abstractions of the complete semantics. A commonly considered abstraction is *configuration semantics*, which is the set of legal configurations, each being an assignment of values to features that satisfies the constraints of the variability model. In CDL, a configuration can be understood as a function assigning each feature a so-called *effective value*. This is the value that is passed to code, when the feature's macro is used (Fig. 3).

The configuration semantics is insufficient for analyses supporting intelligent configuration. The reason is that the CDL configurator shows whether a given feature is enabled or disabled, active or inactive, and its data value. The two states and the data value define the feature's effective value.

Thus, we provide the *configurator semantics* for CDL, which explicitly relates the user input variables, i.e., the enabled states and data values of features, and provides the active state and effective values as derived ones. The semantics is given as a translation from a CDL model to a set of semantics constraints over the enabled state and data value variables. For brevity, this section presents a simplified version of the semantics; we refer to [16] for details.

2.4.1 Variables

The semantic constraints are defined over variables representing enabled states and data values of features. There can be zero, one, or two of such variables per feature, depending on its flavor (see Table 1; "–" means no variable created). Variable n_enable ranges over {0, 1} and n_data ranges over integers, floats, and strings.

The semantics assumes that both the enabled state and data value are available for each feature, regardless of its

Table 1: Variables created for feature n

Flavor	Boolean value	Data value
none	–	–
bool	n_enabled	–
booldata	n_enabled	n_data
data	–	n_data

flavor. When there is no variable for a value, the value is 1. We use the following notation to access the values: $\rho(n)$ returns the enabled state and $\theta(n)$ returns the data value. When there is a variable created for a value, the alias represents the variable; otherwise it represents the value 1.

For example, for the feature in l. 26 of Fig. 2, we create one variable AT91_CLOCK_SPEED_data because its flavor is data; then the enabled state and data value are defined as follows:

$$\rho(\text{AT91\_CLOCK\_SPEED}) \equiv 1$$
$$\theta(\text{AT91\_CLOCK\_SPEED}) \equiv \text{AT91\_CLOCK\_SPEED\_data}$$

Every feature also has an associated variable n_active, stating the active state of the feature. When a feature n is active, the value of n_active is 1, otherwise it is 0. The semantics determines this value uniquely from the variables in Table 1; thus, the active state variables are derived and not stored.

2.4.2 *Semantic constraints*

All feature constraints described in Sect. 2.3 except default_values are translated into semantic constraints. This section lists and explains each semantic constraint produced.

The effective value of a feature is both exposed to C code and used when a feature is referenced in feature constraints. We define the effective value $\sigma(n)$ as follows.

$$\sigma(n) = \rho(n) \wedge \text{n\_active} \ ? \ \theta(n) : 0 \qquad (1)$$

A feature only returns a value $\sigma(n) \neq 0$ if it is active and has been enabled by the user.

A feature is active only when all its active_if constraints are satisfied. Thus, we create the following constraint for each feature n:

$$\texttt{n\_active} \rightarrow \bigwedge_{c \in \text{active\_if(n)}} replace(c) \qquad (2)$$

where *replace* replaces all features reference f in c by $\sigma(f)$.

For example, the active_if in l. 10 (Fig. 2) gives rise to the following constraint, where the reference to USE_TRACING is replaced by its effective value:

$$\text{TRACE\_TIMESLICE\_active} \rightarrow \sigma(\text{USE\_TRACING}) = 1$$

Second, when feature n is enabled and active, all its requires constraints must be satisfied.

$$\text{n\_active} \wedge \rho(n) \rightarrow \bigwedge_{r \in \text{requires(n)}} replace(r) \qquad (3)$$

Third, when a feature is calculated, its value is determined by the expression computing it.

$$\texttt{n\_active} \rightarrow \sigma(n) = replace(\text{calculated}(n)) \qquad (4)$$

Fourth, when a parent is inactive or disabled, all its children are inactive. Let feature p be the parent of feature n. We have then the following constraint.

$$\texttt{n\_active} \rightarrow \texttt{p\_active} \wedge \rho(p) \qquad (5)$$

Also, the active state of a feature is completely determined by its parent and its `active_if` constraints.

$$\texttt{n\_active} \leftarrow \texttt{p\_active} \wedge \rho(p) \wedge \bigwedge_{c \in \text{active\_if(n)}} replace(c) \qquad (6)$$

Finally, legal_values and interfaces are special cases of the above ones. Legal_values can be treated as a requires expression, which constrains the feature value to a range. Interfaces can be treated as calculated features. Their values are the numbers of active and enabled features implementing them.

```
cdl_component LIBM_COMPATIBILITY {
  legal_values {"POSIX" "IEEE" "XOPEN" "SVID"}

  cdl_option LIBM_COMPAT_DEFAULT {
    calculated {
    (LIBM_COMPATIBILITY == "POSIX") ? "CYGNUM_LIBM_COMPAT_POSIX" :
    (LIBM_COMPATIBILITY == "IEEE") ? "CYGNUM_LIBM_COMPAT_IEEE" :
    (LIBM_COMPATIBILITY == "XOPEN") ? "CYGNUM_LIBM_COMPAT_XOPEN" :
    (LIBM_COMPATIBILITY == "SVID") ? "CYGNUM_LIBM_COMPAT_SVID" :
    "<undefined>" }
    flavor data
  }
  ...
  cdl_option UITRON_ISR_ACTION_QUEUESIZE {
     legal_values {4 8 16 32 64 128 256}
  }
}
...
cdl_option CYGBLD_LINKER_SCRIPT {
  calculated {"src/arm.ld"}
  flavor data
}
```

Figure 4: Range definition using calculated and legal_values constructs.

As stated, active state and effective value can be derived from the other variables. Further, the data value of a calculated feature is determined by the expression calculating it. We exploit this observation, to inline the expressions defining the derived variables in semantic constraints, in order to reduce the total number of variables (see [16] for details).

3. METHODOLOGY

Our goal is to characterize the non-Boolean part of constraints in all 116 eCos models, relative to their Boolean content. We analyze both the feature constraints, as stated in the CDL syntax (Sect. 2.3), and the semantics constraints, as defined in Sect. 2.4. We consider the feature constraints, as they allow us to characterize concretely and objectively the non-Boolean aspect that modelers see. We also consider the semantic constraints, as we want to characterize what is exposed to automated reasoners for analyses. Our scope is analyses requiring configurator semantics (cf. Sect. 2), such as those to support intelligent configuration.

Our approach is as follows. (1) At the syntactic level, we first characterize the model sizes and the data types of features. Since many non-Boolean features have restricted value domains, we also analyze these restrictions. We further classify feature constraints as purely Boolean, non-Boolean, or mixed and give occurrence frequencies for non-Boolean operators. We provide summary statistics for all 116 models, reporting minimal, maximal, and median values, and qualitative data, such as sample constraints. (2) We provide similar type of data for the semantic level. We present the number of variables created, along with the characterization of the non-Boolean content of the semantic constraints.

To gather the statistics, we created our own infrastructure with custom tools for each part of the process (parsing the models, semantic translation, and semantic and syntactic level analyses). Since CDL is dynamically typed, we created a heuristic-driven data type inference. To get the models in a parser-friendly format, we reused an instrumented version of the configurator (from [4]) that exports models for each architecture using the *all* template.

4. RESULTS

The eCos models all have similar size, about one thousand features each (cf. Table 2, first row). Coming from the same software project, the models overlap significantly. The most similar models in the set differ only by 2 features, while the most distant pair in the set differs by 307 features. In average, models differ by 122 features, so about 90% of features are shared by a typical pair.

4.1 Feature Data Types

Our first objective was to understand the distribution of data types across the features. We distinguished Boolean and non-Boolean features as follows: all features without the data part, or with data part fixed to 1, are considered Boolean; the remaining features of Data and BoolData flavors are considered non-Boolean. We determined the types of non-Boolean features using an automated procedure. First, the procedure identified non-Boolean features that are constants, enumerations, or ranges (as defined shortly) and it also classified Packages and Interfaces, respectively, as strings and numbers; this step determined the type of 55% of all non-Boolean features with full certainty. Next, a type inference, deriving types from constraint expressions, assigned types to 9% of all non-Boolean features, with an estimated certainty of 90%. Inspecting feature names allowed assigning types to 6% of the features, and the remaining features (30%) were classified as strings—CDL's most generic data type. As can be seen in Table 2, there is roughly the same amount of Boolean and non-Boolean features in a typical eCos model. Furthermore, the non-Boolean features divide almost evenly into character string and numeric features.

We further classify non-Boolean feature data types into enumeration types, range types, and constants. In CDL, there is no explicit construct for declaring such types, but a similar effect is achieved by domain restrictions on data values by means of legal_values and calculated constraints. We use the following heuristics to identify these types.

A feature is a *constant* if its calculated or legal_values constraint only admits a single literal value (string or integer). An example is given in Fig. 4. The calculated constraint of CYGBLD_LINKER_SCRIPT (l. 20) binds its value to *"src/arm.ld"*, which cannot be changed during configuration.

If legal_values or calculated constraints define a finite set of (at least two) literals, we classify the type as an *enumeration*. In Fig. 4, the legal_values constraint of LIBM_COMPATIBILITY (l. 2) restricts the domain to: "POSIX", "IEEE", "XOPEN" and "SVID"—and in l. 15 we see a restriction to an enumeration of integers. Similarly, the LIBM_COMPAT_DEFAULT option has a calculated expression (l. 5) guarded by four conditionals, each resulting in a string literal. We have identified similar patterns of calculated constraints to obtain a distribution of enumeration types across the models.

Ranges are easily identified by the range construct of the legal_values constraint. See for instance feature AT91_CLOCK_SPEED in Fig. 2, which admits values from 0 to 220 million (l. 29).

Table 3 summarizes the distribution of these types across non-Boolean features. The top three rows (the left column compartment) show the number of features classified as constants, enumerations and ranges as a percentage of all the non-Boolean features. Constants and enumerations are further categorized by the source of restriction (calculated or legal_values constraints). The right compartment shows the size of restricted domains—domains of constants are always singletons, and the size of an enumeration domain is the number of values it admits. The size of ranges is defined as the difference between the upper and the lower bound. Its median value (65,535) indicates that ranges introduces short integer types. They are also much more common than constants and enumerations. This is interesting as ranges are harder to handle for SAT and CSP solvers than enumerations (given their relatively large domain sizes). The last row in the table shows the percentage of non-Boolean features with no explicit constraints restricting domains.

We are certain that 42% of the features reported as numbers are integers; the rest could be either integers or floats. In general, feature values can be provided by the user in the configurator or set in the model. Figure 5 presents an example of a float literal that is explicitly specified in the model. Feature POWERPC_BOARD_SPEED (l. 1) specifies the clock speed of the MPC8xx development board. This feature is likely a floating point feature, since its default value is 33.330. Due to dynamic typing of CDL expressions, inputting float literal as feature's value instantly promotes constraints involving such a feature to floating point constraints. Thus, it is possible, but unlikely, that the models contain many floating point valued features.

In general, the ability to identify types of data values is a pre-requisite to almost any automatic analysis of non-Boolean feature models. Thus, a side observation of the experiment we did is that variability modeling should preferably be typed to not discourage tool support.

```
1  cdl_option POWERPC_BOARD_SPEED {
2    default_value 33.330
3    flavor data
4  }
```

Figure 5: A feature with a float data type

4.2 Feature Constraints

This section characterizes the feature constraints specified using constraint properties active_if, requires, legal_values, and calculated (Sect. 2.3). Table 4 groups these constraints into:

- *purely Boolean*, containing expressions with only Boolean

Table 2: Number and types of features

	Min	Max	Median
Model size (#features)	1159	1312	1230
Boolean(%)	44	47	46
Non-Boolean(%)	53	56	54
Number (integer or float)(%)	23	26	25
String(%)	28	32	29

Table 3: Restrictions on non-Boolean types

	% of Features			Sizes		
	Min	Max	Med	Min	Max	Med
Constants	5	7	5	1	1	1
Legal values	0.1	0.5	0.2	1	1	1
Calculated	4	7	5	1	1	1
Enumerations	3	12	5	2	29	3
Legal values	2	10	3	2	29	3
Calculated	1	3	2	2	11	2
Ranges	14	19	15	2	9.2e+18	65535
Unrestricted	69	76	75	n/a	n/a	n/a

operators (!, &&, ||, implies), string or integer literals (interpreted as Booleans), and Boolean feature identifiers; operations ==, !=, and ?: are considered Boolean if their operands are purely-Boolean expressions;

- *purely non-Boolean*, containing expressions with non-Boolean operators (relational, arithmetic, bit-wise, string, and conditional), string or integer literals, and non-Boolean feature identifiers; and
- *mixed*, containing both Boolean and non-Boolean operators and operands.

Purely non-Boolean constraints are the most frequent and control different aspects of the system, such as clock (l. 26, Fig. 2) and board speed (l. 1, Fig. 5), sizes (queues, pools, etc), port numbers and many others. Less frequent are mixed and purely Boolean (Table 4). As many as 83% of the expressions are non-Boolean (either purely or mixed).

Table 9 summarizes occurrences of non-Boolean operators in these five types of constraints (ignore the last column for now). As previously, the counts for ==, !=, and ?: disregard occurrences where these operators are used with only purely-Boolean expressions as operands. Requires constraints contained most of the occurrences of relational operators and a few occurrences of arithmetic operators (+, *, %) and substring tests. Active_ifs used relational operators only. Calculated had the most conditionals (?:), arithmetic operators (+, -, *, /), and string concatenations. Legal_values used a few subtractions and conditionals. Overall, the expressions in the four constraint properties used relatively few non-Boolean operators; this suggests that many of these expressions are non-Boolean feature identifiers and literals.

Purely non-Boolean constraints are almost five times more often than purely Boolean constraints. In this sense, if one were to use a SAT-based analysis, around 79% of the constraints would either be disconsidered or would require an approximation mechanism such as the one described in [3].

4.3 Semantic Constraints

In Sect. 2.4 we have outlined how semantic constraints arise from syntax. Since semantic constraints are closer to what automated reasoners expect as input, this characterization gives a more precise view of the hardness of the models. The semantic constraints are defined over enabled state and data value variables. Table 5 summarizes the numbers of variables in the semantic constraints.

Table 6 classifies the kinds of semantic constraints (compare with Table 9). In contrast to feature constrains, the majority of the semantic constraints are purely Boolean. One explanation is that the semantics itself eliminates constraints that are not essential for configurator semantics (for example by inlining calculated expressions).

The last column in Table 9 shows the occurrence numbers of non-Boolean operators in semantic constraints. The most frequently used are ==, <=, and ?:. String concatenations have a maximum occurrence of 1,536, but are only found in a single model. A deeper examination showed that these occurrences resulted from a single calculated expression with six concatenations that semantics translation inlined into 256 other expressions.

In addition to counting the frequency of operators, we manually extracted common structures (patterns) that appear as part of constraints.

Patterns involving equalities are rather simple (enumeration variables compared to enumerands, variable comparisons, literals compared to variables, etc.) and do not include multiplications, nor divisions. Relational expressions containing < were even less complex, since they only contained at most one data variable at a time.

As for string operations, we found that is_substr calls always test string inclusion of a string literal. Substring testing is used as means to check set membership (whether a compilation option is set, whether a library belongs to the set of libraries for linkage, etc.). Concatenation, on the other hand, was only applied to literals.

The most complex patterns found were inequalities defined in terms of <= and >=. Such inequalities were the only to contain multiplication and division. Due to their large number (over 26,000 in total), we reduced our scope to the ones with at least two data variables. From that, we grouped then in two main types of patterns: (i) inequalities with no Boolean terms; (ii) inequalities in which Boolean terms appear as operands.

We identified four inequality patterns with no Boolean terms, as shown in Table 7. In any pattern, $\square$ denotes either <= or >=, a, b, c are constants greater than or equal to zero and p, q, x, y, z are variables. The first pattern is the simplest and most frequent of all (including inequalities with Boolean terms), occurring at least five times in each model. The other patterns with no Boolean terms denote nonlinear inequalities and were restricted to few models.

The patterns with Boolean terms have a complex structure. We identified seven such (Table 8). In the table, β terms denote Boolean formulae with no data variable. We obtained these patterns by transforming conditional expressions into multiplications. Consider the example of a semantic constraint in Fig. 6. The conditional expression in l. 7 is transformed as follows (β = TIMER_TC_enabled):

$$\beta ? 32 : 16 \;=\; 32\beta - 16(\beta - 1) \;=\; 32\beta - 16\beta + 16 \;=\; 16(\beta + 1)$$

By representing each unique identifier of the constraint as a variable, one obtains $(xyz)/abc(\beta + 1)pq$. Furthermore, the

Table 4: Feature constraints

	Min	Max	Median
All	916	1269	1015
Purely Boolean	162	184	172
Purely non-Boolean	700	1029	792
Mixed	50	66	55

Table 5: Variables in semantic constraints

	Min	Max	Median
Enabled	498	559	521
Data	399	499	420
Total	897	1017	947

Table 6: Semantic constraints

	Min	Max	Median
All	593	686	616
Purely Boolean	405	423	412
Purely non-Boolean	2	4	2
Mixed	184	275	202

```
(1 ≤
  (
    ((
       (RTC_NUMERATOR_data ∗
        (((OSC_MAIN_data ∗ PLL_MULTIPLIER_data) / PLL_DIVIDER_data)/2)
       )
       / (TIMER_TC_enabled ? 32 : 16)
     )/RTC_DENOMINATOR_data #\label{line:xyz−dabpq}#)/ 1000000000
  )
 )
```

Figure 6: Concrete semantic constraint

term abc is also a constant, so we simply represent it as a, leading to $(xyz)/a(\beta+1)pq$. In addition, since $1/a$ is itself a constant, we reformat the pattern into its final version $b \leq a(xyz)/pq(\beta+c)$ (the last pattern of the table). We did the same transformations for obtaining the other patterns. The conditionals for formulas in rows four and five were not expanded to improve readability.

5. THREATS TO VALIDITY

External. eCos might not be representative of other systems using non-Boolean feature models; however, constraints involving memory sizes, clock speeds, timeouts, etc., are likely to appear in highly configurable embedded software.

Furthermore, we have limited the scope of this investigation to 116 models with the largest (*all*) template enabled in our analysis. This selection of scope leaves out other packages in the eCos ecosystem, for instance the commercial packages. There are features that are not included in default templates and we do not consider them. Expressions in those features might contain operators and constructs that are not covered by our results, and that might potentially influence the reported characteristic.

Internal. There is always a threat that statistics we presented are not correct. To prevent this, we wrote extensive unit tests to check the correctness of the parsing and semantic translation, which is the basis for collecting statistics. Statistics were gathered by either traversing the models (for which we also wrote unit tests) and manual analysis. In the latter case each author independently analyzed the results to assure correctness.

A related threat is that our semantics of the CDL language may be wrong, as the semantics was reverse-engineered from the configurator. To reduce this threat, two of the authors worked independently and produced two versions of semantics [3, 16]. We checked the consistency of the two versions and found that their essential contents converge, though they use quite different styles of definitions.

Another threat to internal validity is the type inference result in Table 2. Due to dynamic typing of CDL, assuming a single type for a feature is impossible, since users can put any value as a data value of a feature. We have approximated the intended types using a custom static analysis of the models. We considered feature names and descriptions, and the way features are used in expressions. However, many features are either not used, their name is not reliable, or their usage in expressions is ambiguous. To be completely sure about the feature types, we would need to conduct interviews with the CDL users and to analyze real-world usage of those features. This is a part of the future work.

6. RELATED WORK

Non-Boolean constraints have been used in feature modeling since the very inception. The FODA report [10] admits non-Boolean variables as attributes of features, but no concrete constraint language is proposed for restricting values of such. Non-Boolean decision variables have also commonly been allowed in decision modeling [14].

Reports on practical use of feature models are rare in research literature [7]. Mendonca [12] argues that consideration of realistic models is crucial for obtaining efficient analysis algorithms for feature models, while Segura and Cortes postulate creating a reference benchmark set [15] for analysis tools. Among others, in [12] Mendonca surveys models available in the literature to gather characteristics of such realistic models. However his focus is solely on Boolean models and SAT-based analysis. A side effect of his work is the creation of the SPLOT repository of models [13]. We expand on his work, by characterizing the very first realistic non-Boolean benchmark.

Recently we have characterized large realistic variability models in the OS domain [4], including the CDL model of eCos. The analysis of the eCos model in [4] has side-stepped the non-Boolean aspects, which we approach more closely in the current work (at the same time expanding the analysis to all 116 models). So far eCos is the only publicly available real-world system with a non-trivial non-Boolean model known to us—it includes both string and numeric constraints. The Linux kernel project also has a variability model with non-Boolean (three valued) features, but these can easily be encoded using pairs of Boolean variables [4].

7. CONCLUSION

We have analyzed a collection of over hundred large realistic variability models containing non-Boolean features and constraints; all models originating in the eCos project. eCos demonstrates that arithmetic and string constraints do appear in real-world variability models—even though there was no strong evidence of such presence, yet. The names and descriptions of the features indicate that similar parameters and constraints are very likely to appear in other highly configurable software systems in the embedded domain.

We have summarized the types of features, and syntactic properties of feature constraints as directly specified by the

Table 7: Inequality patterns with non-Boolean terms

	Pattern	Min	Max	Median	Total
linear	$x \;\square\; y \pm a$	5	8	5	595
nonlinear	$axy \;\square\; b$	0	2	0	6
	$axy/z \;\square\; b$	0	2	0	8
	$axy/pz \leq b$	0	1	0	1

Table 8: Inequality patterns with Boolean terms

Pattern	Min	Max	Median	Total
$\beta x \leq y - a$	2	4	2	243
$\beta x \;\square\; \beta y$	1	1	1	116
$\beta_1 x \leq \beta_2 y - a$	0	2	0	2
$\beta_1 x \leq (\beta_2 ? a : \beta_3) + y - b$	0	2	0	2
$x \leq \beta \wedge bool(y) ? \beta y : a$	1	1	1	116
$(\beta_1 \wedge (a < \beta_2 x))y \;\square\; b$	4	4	4	464
$b \;\square\; axyz/(\beta+c)pq$	0	2	0	6

	requires			active_if			calculated			legal_values			Semantic		
	Min	Max	Med	Min	Max	Med	Min	Max	Med	Min	Max	Med	Min	Max	Med
==	71	88	74	3	8	4	6	51	11	0	2	0	106	423	166
!=	40	43	40	0	5	0	0	2	0	0	4	0	0	0	0
<	0	0	0	3	4	3	0	0	0	0	0	0	6	7	7
<=	4	7	4	0	0	0	0	1	0	0	0	0	208	282	220
>	3	3	3	7	7	7	0	1	0	0	0	0	0	0	0
>=	71	72	71	0	0	0	0	0	0	0	0	0	2	3	2
+	1	2	1	0	0	0	0	5	0	0	0	0	3	51	5
–	0	0	0	0	0	0	0	1	0	1	4	3	1	4	3
*	0	1	0	0	0	0	0	6	0	0	0	0	0	8	0
/	0	0	0	0	0	0	0	11	0	0	0	0	0	18	0
%	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
<<	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
>>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
&	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
|	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
ˆ	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
?:	1	1	1	0	0	0	36	81	42	4	10	4	327	749	430
. (concatenate)	0	0	0	0	0	0	22	34	22	0	0	0	0	1536	0
is_substr	3	13	3	0	0	0	0	7	0	0	0	0	1	5	1
is_active	0	3	0	0	0	0	0	0	0	0	0	0	0	3	0
is_enabled	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Table 9: Non-Boolean operator occurrences in feature and semantic constraints

modelers. We observe that non-Boolean constraints tend to be relatively simple, but a closer look reveals that they are a significant challenge for analysis techniques.

We have analyzed a formal semantic of the models suitable for the configurator tools. The semantic expansion contains a number of nonlinear constraints involving multiplications and division of non-Boolean feature values. Also most of the linear constraints in the model are embedded into complex combinatorial expressions, breaking linearity. Thus models can't be analyzed neither by reduction to constraint programming nor by (integer) linear programming.

In the future we would like to investigate reasoning techniques that address different kinds of constraints identified in this study. In particular, we would like to map out the dependencies between efficiency of the analyses, with various degree of precision (i.e. taking various subsets of constraints into account, depending on their richness).

Acknowledgments We thank Don Batory for encouraging us to do this study and Klaus Schmid and the anonymous reviewers for their comments.

8. REFERENCES

[1] D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, 2005.

[2] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35, September 2010.

[3] T. Berger and S. She. Formal semantics of the CDL language. Technical Note. Available at http://informatik.uni-leipzig.de/~berger/cdl_semantics.pdf.

[4] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. ASE, 2010.

[5] M. Bozga and R. Iosif. On decidability within the arithmetic of addition and divisibility. 2005.

[6] K. Czarnecki and A. Wąsowski. Feature diagrams and logics: There and back again. In *SPLC*, 2007.

[7] A. Hubaux, A. Classen, M. Mendonça, and P. Heymans. A preliminary review on the application of feature diagrams in practice. In *VaMoS*, 2010.

[8] B. A. Huberman and T. Hogg. Phase transitions in artificial intelligence systems. *Artif. Intell.*, 33(2), 1987.

[9] M. Janota. Do SAT solvers make good configurators? In *SPLC (2)*, 2008.

[10] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, 1990.

[11] A. Massa. *Embedded Software Development with eCos*. New Riders, 2002.

[12] M. Mendonca, A. Wąsowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *SPLC*, 2009.

[13] M. Mendonça, M. Branco, and D. D. Cowan. S.p.l.o.t.: software product lines online tools. In *OOPSLA Companion*, 2009.

[14] K. Schmid, R. Rabiser, and P. Grünbacher. A comparison of decision modeling approaches in product lines. In *VaMoS*, 2011.

[15] S. Segura and A. R. Cortés. Benchmarking on the automated analyses of feature models: A preliminary roadmap. In *VaMoS*, 2009.

[16] Y. Xiong. Configurator semantics of the cdl language. Technical Report GSDLAB-TR 2011-06-05, GSD Lab, University of Waterloo, 2011. http://gsd.uwaterloo.ca/GSDLAB-TR2011-06-05.