Towards eCos Autoconfiguration by Static Application Analysis

Horst Schirmeier, Matthias Bahne, Jochen Streicher and Olaf Spinczyk
Computer Science 12 – Embedded System Software
Technische Universität Dortmund
{horst.schirmeier,matthias.bahne,jochen.streicher,olaf.spinczyk}@tu-dortmund.de

***Abstract*—System software product lines such as the embedded real-time operating system *eCos* are a state-of-the-art solution for application scenarios with a very low hardware resource profile. Being highly configurable at compile time, *eCos* can be tailored to the application in terms of inclusion or exclusion of fine-grained operating system features.**

The mandatory manual feature selection is carried out by the developer who knows the functionality essential for the application. This process requires profound knowledge of the feature semantics, is very time-consuming, and, in particular, error-prone – most probably resulting in a resource suboptimal or even dysfunctional operating system variant.

The contribution of this article is an approach to automate the *eCos* configuration process to a major degree, therefore reducing the outlined disadvantages significantly. A thorough examination of configurable *eCos* features exhibits four feature categories of varying complexity: By applying standard *model checking* techniques and *static analysis* of the application source code, we show that for at least two feature categories the configuration decisions can be taken automatically. Complementing the approach with *configlets* – highly *eCos* specific analysis code – a third category is shown to be also coverable.

***Index Terms*—eCos, Automatic Configuration, Static Analysis, Model Checking, CTL, Software Product Lines**

I. Introduction

There exist various reasons for specializing software systems for a specific application scenario or a particular customer. In the embedded systems domain, the primary motivation is to minimize resource consumption: Resources such as energy, memory or CPU cycles are scarce, and keeping the software stack's resource profile low directly impacts the system's hardware costs and therefore its competitiveness on the market. Tailorable system software has been a focus of our research group for several years, in particular embedded operating systems (OS) [14], [15], [18], embedded application product lines [16], and embedded databases [23], [24].

This article describes a case study conducted with the open-source, real-time embedded OS *eCos*[1], a highly configurable software platform, which can be tailored for the specific needs of the application. *eCos* is very widespread, because of it being freely available and its comfortable configuration process: Guided by a GUI configuration tool, the application developer decides from a so-called *feature model* [6], [9] which OS functionality the application mandatorily needs and therefore must be included in the *eCos* variant deployed on the device. Afterwards, an automated product generation process results in an OS library the application can be linked against. This state-of-the-art procedure has proven to be very effective in practice, but also entails a non-negligible problem: In the case of *eCos*, the sheer number of more than 1.500 configurable features[2] makes the configuration process extremely complex and time-consuming.

Our contribution to the automated configuration and tailoring community is an approach to at least partially automate the process by applying standard static analysis and model checking techniques to application source code. We conducted a detailed examination of parts of the *eCos 3.0* OS documentation and API in order to determine which conditions hold within an application model if it needs or does not need certain OS features. These conditions, still formulated in plain English, pose a set of requirements for static analysis and model checking. First experiments with our analysis tool prototype seem promising, but also raise new research questions.

The remainder of this paper is structured as follows: Section II gives a more detailed overview of the motivating context and our approach. Section III describes our manual examination of *eCos* features and examples for the resulting feature categories. Section IV outlines the steps necessary for automatic feature need derivation: application analysis, model checking, and result interpretation. Section V revisits the results from the manual feature examination, and summarizes the current state of our analysis tool prototype and preliminary results with *eCos* test programs. We conclude with a discussion of related work, a summary and an outlook on future work in sections VI and VII.

II. Motivation and Approach

A. eCos is an Infrastructure Software Product Line

Infrastructure software, unlike application software, is in a unidirectional *usage relationship* with other software layers, and typically depicted in a hierarchy level *above* the infrastructure. Database software, middleware or standard C libraries, or operating systems such as *eCos* are typical representatives.

The categorization of *eCos* as a software product line (SPL) is a bit more debatable: The SPL community defines these not only by technical aspects like configurability and code

[1] Embedded Configurable Operating System, http://ecos.sourceware.org/

[2] *eCos* release 3.0; platform specific HAL and device driver features not counted in

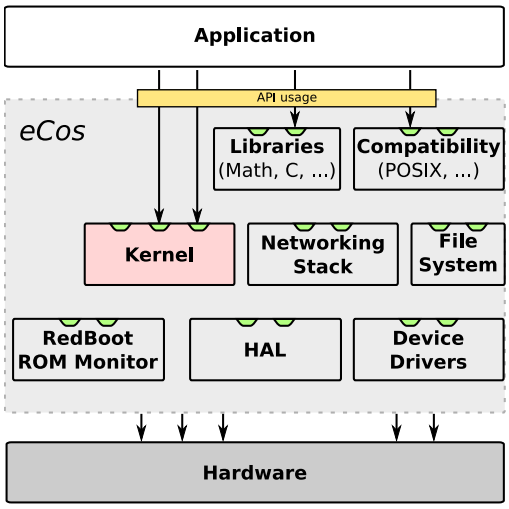


Figure 1: *eCos:* Architectural overview [17]. The unidirectional *API usage* relationship application–OS can be exploited for the detection of *feature need*.

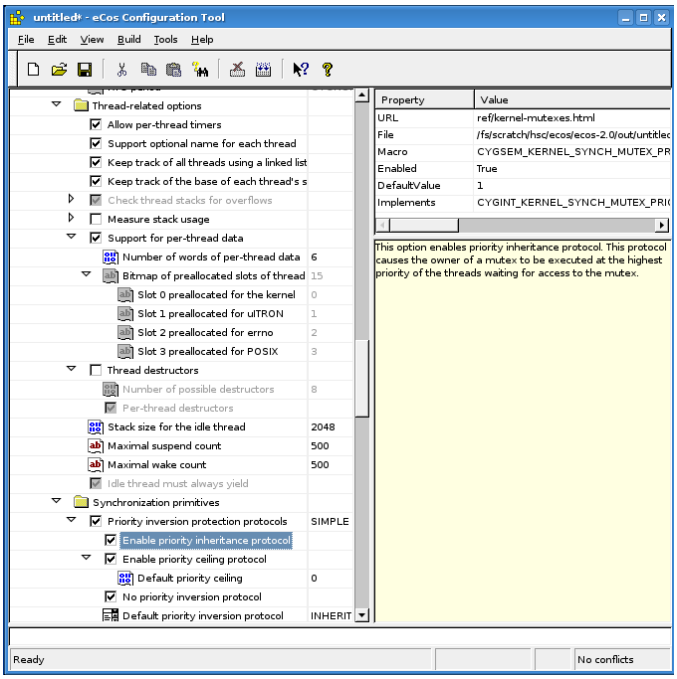


Figure 2: The *eCos Configtool* with a hierarchical display of components and different types of options.

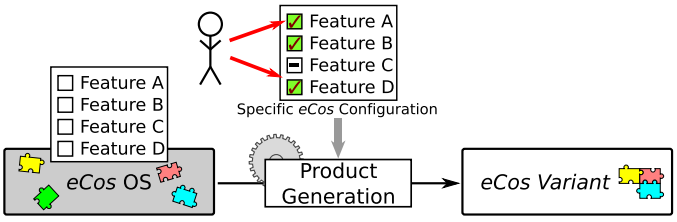


Figure 3: Instantiating an *eCos* variant: User configuration drives the product generation, yielding a tailored variant implementing only the desired features.

reuse among product line variants, but also by the engineering process that accompanies the development [5]. Nevertheless, we believe we can safely consider *eCos* a SPL, because it has all properties relevant to this study:

- **Variability Management:** Similar to the Linux kernel [25], *eCos* supports about a dozen different embedded hardware architectures (with countless subarchitectures), but clearly separates the platform specific variation points from platform independent ones. The configuration tree, separating problem space (configurable features relevant for the application developer) and solution space (the actual implementation behind these features), is organized hierarchically in *subsystems* (*Kernel*, *Hardware Abstraction Layer*, *Standard Libraries*, *I/O*, and other optional subsystems) and several levels of nested *components* (e.g., *Serial Device Drivers*, *ISO C Library String Functions*), cf. Figure 2. On the configuration tree's leaves exist three different types of configuration *options:* binary options (on/off), one-from-many selections (e.g., allowing the selection of one from three different scheduling algorithms), and value options (e.g., a maximum number of concurrent threads, an integer number within an interval). Additionally, *constraints* restrict the configuration space to prevent impossible configurations. [28]
- **Product Configuration:** *eCos* comes with a comfortable GUI configuration tool that allows creating a specific configuration by assembling components and configuring the contained options (cf. Figure 2). Feature *constraints* like dependencies or mutually excluding features are automatically enforced.
- **Product Generation:** The GUI tool is also capable of automatically generating the actual *eCos variant* from the configuration (cf. Figure 3). This process generates a new source code tree: The chosen values for the configuration options are written as C preprocessor macro definitions in generated header files, which in turn have effects on the OS C/C++ code that is finally compiled and linked to the application.

In this so-called *feature-driven product line derivation* process [27], the application developer has to make all configuration decisions manually in order to tailor the underlying infrastructure product line for his application's needs. With highly configurable, real-world product lines such as *eCos*, the configuration effort – although guided by a GUI tool with automatic inter-feature dependency resolution and textual descriptions of all configuration options – easily exceeds acceptable limits. Moreover, a vast configuration space promotes chances for human mistakes, leading to a resource suboptimal (if too much functionality was configured) or insufficient (essential functionality missing) infrastructure variant. Even worse, this process needs to be iterated once the application evolves. Consequently, (at least partial) automation of the process and tool support is desirable.

B. API Usage Patterns and Model Checking

In earlier work [24] we showed that infrastructure API *usage patterns* within an application allow to deduce feature need or non-need. Ranging from the simple *existence* of certain API calls to complex API *call patterns* and involvement of actual parameter values, different levels of complexity have been observed for real-world infrastructure features. In this case study, we take a closer look at the *eCos* API and its characteristics related to application analysis and *feature relevant* API usage patterns. Motivated by our earlier results, we evaluate the temporal logic CTL (*computation tree logic* [4]) as a language for formulating feature conditions to be checked within an application model.

III. Examining the eCos Operating System

For our approach, the first step towards automatic configuration comprises establishing a relationship between *infrastructure features* on the one hand, and *properties of the application* on the other hand. We conducted a thorough manual examination of the configurable features in the *eCos* kernel components *"Kernel schedulers"*, *"Thread-related options"* and *"Synchronization primitives"* in order to gain experience with real-world OS features and to study the implications for application analysis.

This manual infrastructure feature examination must not be confused with the automated analysis steps described later in this article:

- This examination must be carried out only *once*, whereas its results can be reused in all infrastructure deployments.
- Therefore, an infrastructure expert (e.g., a member of its development team, outfitted with in-depth knowledge of the API semantics) can (and should) be the person conducting this analysis.
- In contrast, the *automated* analysis steps take place later at the application developer's site.

A. Manual examination

The aim of the manual examination of the *eCos 3.0* source code and documentation was to ascertain what exact effect each configurable feature has on actual OS functionality, and especially the OS API. We show that an application, containing *calls* to this API, fulfills certain conditions that indicate its need for OS features. The results, including these *feature conditions*, were diverse and are presented in the following in terms of four feature categories, each accompanied by examples.

1) Simple API call usage: One very common case of API usage observed is the simple *existence* of certain API calls. For example, enabling *"Message box blocking put support"* (located in the *"Synchronization primitives"* component) defines a preprocessor macro, which in turn adds the functions *cyg_mbox_put(...)* and *cyg_mbox_timed_put(...)* to the kernel API. Knowing this direct relationship from feature selection to the API, and being able to assure these API calls are not additionally related to other features, one can pose a simple, definitive condition on the application code:

```
class Cyg_Thread {
/* ... */
#ifdef CYGVAR_KERNEL_THREADS_NAME
private:
    // An optional thread name string,
    // for humans to read
    char *name;
public:
    // function to get the name string
    char *get_name();
#endif
/* ... */
```

Figure 4: An excerpt from an *eCos 3.0* internal kernel header file: The effect of a GUI configuration option on an internal thread abstraction data structure.

> *Iff there exist (reachable) calls to* cyg_mbox_put(...) *or* cyg_mbox_timed_put(...)*, then the feature* "Message box blocking put support" *is needed.*

2) API call patterns and actual parameters: Another pattern can be illustrated by the example of the feature *"Support optional name for each thread"*, located in the *"Thread-related options"* component. This option enables applications to assign a name (a C character string) to each thread, and to read that name from an arbitrary, given thread.

Once this option is enabled, the kernel's internal thread abstraction data structure (a C++ class named *Cyg_Thread*) is extended by a *name* element (again by means of preprocessor macros, cf. Figure 4), thus saving memory (and OS code, in other parts of *eCos*) if this functionality is not needed.

Nevertheless, the kernel's C API is not affected at all: Regardless of how the application developer configures *eCos*, the API function *cyg_thread_create(...)* always takes a *name* parameter (which is ignored if the kernel cannot store thread names), and the API function *cyg_thread_get_info(...)* always fills a data structure (*struct cyg_thread_info*) which has a *name* element. An application developer who does not want to give a thread a name simply passes *NULL* instead of a character string. If the kernel does not know a thread's name, or is not capable of storing thread names, *cyg_thread_get_info(...)* returns *NULL* in the *name* element of *cyg_thread_info*.

This API definition allows to concentrate on an application's usage of *cyg_thread_create(...)* and *cyg_thread_get_info(...)* for automatic configuration of this feature:

> *Iff the application passes a non-NULL parameter value to* cyg_thread_create(...)*'s* name *parameter,* ***and*** *calls* cyg_thread_get_info(...) ***and*** *reads the* name *element of its return value, the feature* "Support optional name for each thread" *is needed.*

This condition is not completely conclusive, though: An application developer may create threads with names just for self-documenting purposes without needing the kernel to

store this information. An application may even read the *name* element and expect it to be NULL. Nevertheless there is a strong *indication* that the feature is needed, so the configuration tool could give the application developer a *hint* to enable it.

3) Domain specific patterns: Some options exhibit an inherent complexity, and are, due to their interrelation with concurrency and synchronization aspects, highly OS specific. An extreme example is the option *"Priority inversion protection protocols"* in the *"Synchronization primitives"* component: It controls whether synchronization primitives should be protected against priority inversion [11].

As this feature comes with quite some cost (program memory, CPU cycles at runtime), it should only be enabled if priority inversion is a problem in the application at all:

> *Iff the application is structured in a way that priority inversion can occur, the feature* "Priority inversion protection protocols" *is needed.*

Unfortunately, detecting this property in an application is not trivial and can only be approached with an approximation:

> *Iff at least two threads with different priority levels share a common mutex,* ***and*** *a thread with a priority level between these two exists, the feature* "Priority inversion protection protocols" *is needed.*

There are several reasons why an application developer still may not need this feature: The application may be designed in a way that priority inversion is prevented by other means, or the occurrence of this effect may even be *intended*. So, again, the configuration tool should only give a *hint*.

4) Not derivable: For the remaining features we were unable to pose feature conditions, for different reasons. A detailed analysis of this category follows in section V.

B. Implications for Static Analysis and Model Checking

The *eCos* features we examined and the resulting *feature conditions* imply several requirements on the application analysis:

- Feature conditions that relate only to the *existence* of certain API calls, the least complex category identified in the previous subsection, could more or less be checked by simple text search utilities like *grep*. The capabilities missing (and essential for well-founded automatic configuration decisions) are a deeper understanding of C syntax and semantics – for example to differentiate between an API function mentioned in a source code comment, and real API usage – and control flow properties that indicate whether the API calls are *reachable* at all.
- Patterns as observed in the *"Support optional name for each thread"* example also demand for control flow information (connecting a *cyg_thread_get_info(...)* call to a subsequent use of its return value) and actual parameter values, implying the need for static analysis techniques like constant folding, constant propagation, inter-procedural reaching definitions analysis [21], or interval analysis.
- Highly OS domain specific feature patterns concerning thread concurrency and synchronization aspects call for a very specialized program analysis technique which may not be possible to generalize further. Our first attempt to deal with these is described in subsection IV-D.

IV. Static Application Analysis and Determination of Feature Need

A. Static Analysis of eCos Applications

The first step in statically deriving information from application source code is transforming it into a model suitable for the subsequent analysis steps – suitable in a way that the transformation does not lose information relevant for the problem space, leading to incorrect configuration decisions. Our examination of *eCos* features suggests that a C-statement-level *control flow graph* (CFG) fulfills our requirements, as it maintains temporal interrelations between API calls and return value usage, and allows for reachability analysis. A CFG can also be subject to further data flow analysis and transformation steps that propagate actual values to nodes where they are passed through the OS API. Additionally, a CFG is generic enough that it still holds enough program information for the highly OS specific analyses sketched in subsection III-A3.

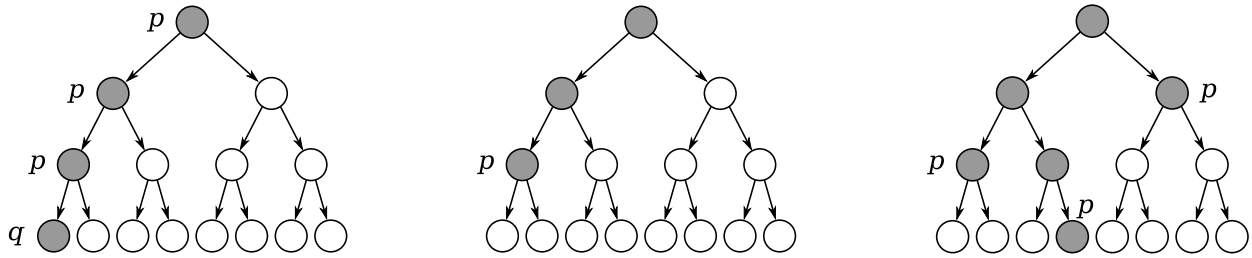
B. Formalizing Feature Conditions and Model Checking

Motivated by our own earlier results [24] and promising achievements in other projects applying model checking to identify patterns in application code (such as [10]), we decided to use the temporal logic CTL as a language to formalize feature conditions. CTL can express temporal relationships, and there exist efficient model checking algorithms [4], [10] for this logic.

We model the CFG of the application program as a Kripke structure [4] $M = (S, S_0, R, L)$ over atomic propositions P, where S is a (finite) set of states (representing statement-level CFG nodes), S_0 is a set of initial states, $R \subseteq S \times S$ is a total transition relation, and $L : S \to 2^P$ is a labeling function that associates a subset of P to each state. A proposition p *holds* in a state s, if p is contained in the label of s, i.e., $p \in L(s)$. A path π, starting at state s, is a sequence of states $\pi = s_0 s_1 s_2 s_3 \ldots$ with $s_0 = s$ and $R(s_i, s_{i+1})$ for all $i \geq 0$.

CTL formulas allow to specify temporal properties of Kripke structures by extending classical predicate logic by temporal connectives which establish a relationship to future states:

- Every atomic proposition $p \in P$ is a CTL formula.
- If p and q are CTL formulas, $\neg p, p \wedge q, p \vee q, AX\,p, EX\,p, AG\,p, EG\,p, A[p\,U\,q]$ and $E[p\,U\,q]$ are valid formulas, too. X is the *nexttime* operator: $AX\,p$ (resp. $EX\,p$) means that p holds in every (in some) immediate successor state. G is the *global* operator: $AG\,p$ (resp. $EG\,p$) states that p holds in all (in some) future states. U is the *until* operator: $A[p\,U\,q]$ (resp. $E[p\,U\,q]$, cf. Subfigure 5a) expresses that for every (for some) path starting with the current state, there exists an initial prefix of this path such that q holds at the last state of this prefix, and p holds in all other states of the prefix [3].
- The *finally* operators AF and EF can be seen as an abbreviated form of the *until* operators with $p = true$



(a) $E\,p\,U\,q$ (b) $EF\,p$ (c) $AF\,p$

Figure 5: Examples for simple CTL formulas and corresponding state trees.

(cf. Sub-figures 5b and 5c): They mean that on all (on some) paths the operand will hold sometime in the future ($AF\,q \Leftrightarrow A[true\,U\,q]$).

Figure 5 and the examples later in this subsection may help gaining a more intuitive understanding of CTL.

For the feature conditions identified in subsection III-A we additionally define an elementary set of atomic propositions:

- The trivial proposition $[true]$ holds in every, $[false]$ in no state.
- $[call$:$\mathbf{somefunction}(\ldots)]$ holds in every state that contains a call to **somefunction**, ignoring its actual parameters. $[call$:$\mathbf{somefunction}(?,\ \mathbf{4},\ ?)]$ holds if the second actual parameter is equal to 4.

Now we can formulate a subset of the feature conditions from section III-A in CTL. The condition informally introduced in subsection III-A1 can be expressed as follows:

$$EF\,(\,[call\ :\mathbf{cyg\_mbox\_put}(\ldots)]\ \vee\ [call\ :\mathbf{cyg\_mbox\_timed\_put}(\ldots)]\,)$$

For now ignoring the *"... and reads the* name *element of its return value, ..."* part of the condition, we can also express the condition from subsection III-A2:

$$\begin{aligned} EF\,(\,[call\ :\mathbf{cyg\_thread\_create}(\ldots)] \quad &\wedge \\ \neg[call\ :\mathbf{cyg\_thread\_create}(?,\ ?,\ ?,\ \mathbf{0},\ ?,\ ?,\ ?,\ ?)] \quad &\wedge \\ EX\,EF\,[call\ :\mathbf{cyg\_thread\_get\_info}(\ldots)]\,) \end{aligned}$$

In order to formulate this condition completely, we would need another atomic proposition for *data structure access* and a means to express the connection between the variable returned by *cyg_thread_get_info(...)* and the point where its member *name* is accessed. (See the discussion in subsection IV-D for a solution to this problem.)

A model checking algorithm can now check whether the application's Kripke structure is a model for a CTL formula – iff this is the case, the feature condition is fulfilled. Starting with the atomic propositions used in the CTL formula, and recursively continuing with more and more complex sub-formulas, each Kripke node is labeled with the sub-formulas holding for this particular node. This procedure is iterated until it can be determined whether the complete formula holds in a starting node. For more details on the model checking algorithm we refer to Clarke's original publication [3].

C. Inferring Feature Need

Although not accounted for in section III-A, our examination of *eCos* features showed that often *multiple* feature conditions are necessary to exhaustively describe the possible API usage patterns that signify an application's need or non-need for a certain feature. Some conditions only help deciding *for* or *against* feature need, not both: Once a single *necessary* (in the mathematical sense) condition is *not* fulfilled, the feature this condition belongs to is *not* needed; once a single *sufficient* condition is fulfilled, the feature is definitely needed. In both cases, the respective converse decision cannot be taken: The fulfillment of a necessary condition does not allow a statement on feature need.

Another orthogonal condition property is *weakness:* If a condition should only be used to give the user a configuration *hint* instead of making a fully automated configuration decision, it is called a *weak condition* (examples are the conditions in subsections III-A2 and III-A3), or else a *definite condition.*

The model checking results for all conditions of a certain feature can be combined to a resulting statement on whether the feature should be enabled or disabled. Figure 6 assembles the following propositions:

N	$\Leftrightarrow$	all definite, necessary conditions are fulfilled (or none exists).
n	$\Leftrightarrow$	all weak, necessary conditions are fulfilled (or none exists).
S	$\Leftrightarrow$	some definite, sufficient condition is fulfilled.
s	$\Leftrightarrow$	some weak, sufficient condition is fulfilled.

One remaining case still needs to be dealt with: If a condition cannot be successfully checked (e.g., because data flow analysis fails, and therefore the possible value(s) of an actual parameter cannot be determined), we can safely proceed pretending this condition did not exist in the first place, as evidently the resulting statement is not falsified by that measure. The user should still be informed about the omission of a feature condition, as a simple code change may make the analysis possible again.

Finally, the internal dependency and restriction constraints in the *eCos* configuration tree need to be taken into account – a feature not directly needed by the application may still be mandatory because another feature depends on it. This

	$\neg N$	$N \wedge n$	$N \wedge \neg n$
S	Conditions or application source code **ill-formed**	**Yes**, feature definitely needed	**Yes**, feature definitely needed
$\neg S \wedge \neg s$	**No**, feature definitely not needed	**Maybe**, no definite decision possible	**Hint: No**, probably not needed
$\neg S \wedge s$	**No**, feature definitely not needed	**Hint: Yes**, probably needed	**Maybe**, no definite decision possible

Figure 6: Combining different classes of feature conditions, yielding a configuration decision. Definite decisions, based solely on *definite conditions*, have a shaded background.

dependency resolution can already be conducted by the *eCos Configtool*, though.

D. Discussion

The quality of results obtained from model checking can only be as good as the model being checked, which in the case of control and data flow graphs (at least for real-world code input) may not yield all necessary information (e.g., failing to trace actual parameters to the place they are defined, or not being able to determine the number of iterations a loop can make). Static code analysis is well-known to be limited in theory and practice [10], [12], [19], [21], but nevertheless we believe that these limits are not pivotal for our approach: Even if some configuration decisions cannot be automated because static analysis fails, and the application developer has to configure the respective features manually, the overall configuration effort is still significantly reduced. Additionally, this problem could be tackled by trying to reduce the complexity of static analysis. An interesting research question in this context would be: What modifications to an infrastructure's API might be adequate to facilitate static analysis and therefore improve the quality of automatic configuration decisions?

Another related open question is what API the application should be initially written against: As the API may be affected by configuration decisions, a circular dependency exists. In principle, a "full" API encompassing all possible configurations (something Fröhlich [7] calls an *inflated interface*) needs to be generated. This may not be possible if two API variants conflict (e.g., a function cannot have two different signatures in the C programming language), although this was not the case for *eCos*: We were able to manually create a set of header files valid for all possible variants.

The additional expressiveness demanded in subsection IV-B – CTL's lack of a means to connect several usage points of a single variable – could be handled by a CTL extension: With *CTPL*, Kinder et al. [10] proposed such an extension to CTL that allows for expressing such connections by introducing free variables (placeholders). Their modified model checking algorithm was shown to be still efficient.

Unfortunately, neither CTL nor CTPL are expressive enough to cover complex feature conditions such as described in subsection III-A3. Our first attempt to deal with these highly OS domain specific conditions is to encapsulate them in small plug-ins, so-called *configlets*, written in C++ against an analysis API giving direct access to the underlying CFG (cf. Figure 7). A configlet can be used in place of a CTL formula, its output is processed the same way (cf. Figure 6). We are still unsure about what the interface between a *configlet* and the remaining analysis tool interface should look like, and are trying to find an adequate balance between a complex API with complete access to the Kripke model, and a minimal, easier to use API which still is generic enough to deal with new use cases.

V. Results

A. eCos examination

The manual examination of 39 eCos features (cf. subsection III-A) resulted in a categorization into four feature categories, bearing various levels of complexity for automatic analysis. Table I lists all features, their type, their feature category, and the types of feature conditions we formulated.

- 7 out of 39 features were categorized into *"Simple API call usage"*.
- 5 features fall in the *"API call patterns and actual parameters"* category: Two of these need the CTL extension mentioned in subsection IV-D.
- 6 features are highly OS specific and belong to the *"Domain specific patterns"* category. We were able to formulate conditions for these features, but CTL (and CTPL) turned out to be insufficient.

19 features were categorized to be not derivable from an application's source code, for different reasons:

- The largest group (13) are configurable *non-functional properties* that have an effect on code size (e.g., explicit

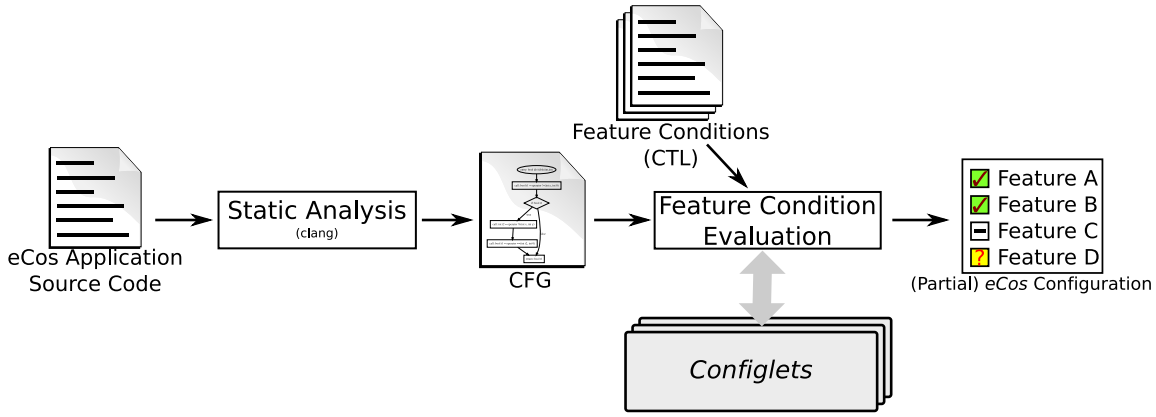


Figure 7: Analysis process overview: *Configlets* complement CTL model checking for complex feature conditions.

Feature name	Type	Feature condition types found	Category
Multi-level queue scheduler	one-from-many	sufficient	3
Bitmap scheduler	one-from-many	necessary	3
Number of priority levels	value (1–32)	3×sufficient	3
Dequeue highest priority threads first	binary	–	4
Scheduler timeslicing	binary	1×necessary, 1×sufficient *(weak)*	3
Number of ticks between timeslices	value	–	4
Support runtime enable of timeslice per-thread	binary	necessary/sufficient	1
Enable ASR support	binary	–	4
Make ASR function global	binary	–	4
Make ASR data global	binary	–	4
Priority inversion protection protocols	binary	necessary/sufficient *(weak)*	3
Enable priority inheritance protocols	binary	necessary/sufficient	2
Enable priority ceiling protocol	binary	necessary/sufficient	2
Default priority inversion protocol	one-from-many	–	4
Specify mutex priority inversion protocol at runtime	binary	necessary/sufficient	1
Use mboxt_plain mbox implementation	binary	(sufficient)	4
Message box blocking put support	binary	necessary/sufficient	1
Message box queue size	value	–	4
Condition variable timed-wait support	binary	necessary/sufficient	1
Condition variable explicit mutex wait support	binary	necessary/sufficient	2
Avoid inlines in mqueue implementation	binary	–	4
Allow per-thread timers	binary	–	4
Support optional name for each thread	binary	1×necessary, 1×sufficient	2
Keep track of all threads using a linked list	binary	1×sufficient	1
Keep track of the base of each thread's stack	(binary)	–	(4)
Check thread stacks for overflows	binary	–	4
Check all threads whenever possible	binary	–	4
Signature size in bytes, at stack top and bottom	value (8–512)	–	4
Measure stack usage	binary	necessary/sufficient	2
Output stack usage on thread exit	binary	–	4
Support for per-thread data	binary	necessary/sufficient	1
Number of words of per-thread data	value (4–32)	–	4
Thread destructors	binary	necessary/sufficient	1
Number of possible destructors	value (1–65535)	necessary/sufficient	3
Per-thread destructors	binary	–	4
Stack size for the idle thread	value (512-65536)	–	4
Maximal suspend count	value	–	4
Maximal wake count	value	–	4
Idle thread must always yield	(binary)	–	(4)

Table I: The examined *eCos* features, listed with the feature category (cf. subsection III-A) and the condition types. The condition types found are *definite* unless explicitly denoted *weak*. Category legend: 1) Simple API call usage, 2) API call patterns and actual parameters, 3) Domain specific patterns, 4) Not derivable.

code inlining), timing/throughput behavior, stack sizes, or runtime safety checks (e.g., stack canaries).
- 4 features could not be examined in detail due to their unclear semantics.

Among these 19 features, 6 were additionally identified to be of interest only for the development process and/or debugging purposes. For example, *"Output stack usage on thread exit"* creates debugging output once a thread terminates – a feature the developer, despite being eager for automatic configuration, still would want to configure manually due to its purpose in the development process context.

The remaining two features even seem only to be there for self-documentation reasons, as they are tightly coupled with other features by dependency relationships and cannot be directly configured by the user anyways (e.g., *"Idle thread must always yield"* is automatically enabled once there is only a single priority level configured).

Altogether, we were able to pose feature conditions for about half of the examined features (18 out of 37 "real" features).

B. Implementation and preliminary evaluation

The prototype implementation of the described analysis and model checking steps is still very preliminary and has not yet been thoroughly tested with real-world code input. Nevertheless, our test runs with simple CTL formulas and a suite of *eCos* test programs (shipped with *eCos 3.0*), cloned and adapted for testing specific units of the prototype, are promising and will soon be extended to a full-scale evaluation.

The analysis tool prototype is based on *clang*, an LLVM [13] compiler front-end for the C, C++ and Objective-C languages. The complete static analysis phase (scanning and parsing the application's C code, performing constant folding and propagation) is externalized to *clang*. The resulting CFG is then transformed to a Kripke structure. A C++ implementation of Clarke's CTL model checking algorithm [4] is currently capable of checking single CTL formulas and combining the results for a configuration decision. Besides the sub-condition requiring a CTL extension (cf. subsection IV-D), all feature conditions categorized in subsections III-A1 and III-A2 can already be checked. A prototype *configlet* for the example condition in subsection III-A3 is under development.

VI. Related Work

There exist numerous research projects engaging in application-driven tailoring of infrastructure software. The concept of *Application-Oriented System Design* was introduced by Fröhlich [7] in the EPOS operating system. Here the set of infrastructure symbols (i.e., a temporally unordered list of API calls) referenced by the application determines which product line variant is needed. If multiple variants fulfill the requirements, the least resource expensive one (in a fixed, manually defined cost ordering of variants) is chosen. Apart from the comparably simple static analysis the main difference to our approach is the lack of logical isolation between analysis and configuration, established by a feature model.

Evaluating their *JiM* (*Just-in-time middleware*) paradigm, Zhang et al. [29] tailor the middleware product line *Abacus* (a CORBA middleware implementation) according to the application's needs. In contrast to the focus of our work, they completely ignore the task of derivation of feature need from the application. Source code annotations are added to the application code to make the feature requirements explicit; the difference to manual infrastructure configuration is marginal and debatable. In their customizable embedded DREAMS OS, Böke, Chivukula et al. [1], [2] also achieve infrastructure tailoring by supplying an explicit *requirements specification*. The specification is on a lower abstraction level than in the *Abacus* case, though: The system calls and their properties (precedence relation, number of calls, etc.) made by each process are explicitly listed. The DREAMS tailoring approach also benefits from the OS being explicitly designed for automatic configuration.

Focusing mostly on *functional* infrastructure properties, the aforementioned projects insufficiently deal with *non-functional* properties (such as code size, memory usage at runtime, timing behavior etc.). These properties have an emergent nature and therefore often cannot be directly configured, nevertheless some research work exists in this field. Pu, Massala et al. developed the research operating system *Synthesis* [22] which adapts to and self-optimizes for the application's API usage patterns at runtime. Supplied by a just-in-time compiler, often used code paths are partially evaluated and tuned to high throughput. Sincero, Hofer et al. [8], [26] are developing feedback approach which systematically collects information on emerging non-functional properties in the *CiAO* [14] operating system to be able to make educated predictions for unknown configurations. This information could be used to automatically (based on a user-defined non-functional optimization goal) select one of several variants which are functionally equivalent but exhibit different non-functional properties, therefore complementing our approach.

Outside the context of infrastructure software tailoring, there exist many other research projects that apply static analysis and model checking to problems in their domain. For example, Padioleau, Lawall et al. [20] utilize CTL for matching and replacing code in order to deal with *collateral evolutions* in the Linux kernel. The SmPL language can be used to define so-called *semantic patches* which can automatically be transformed into CTL formulas. Another example comes from the systems security domain: Kinder et al. [10] formidably show the applicability of static analysis model checking to the generic detection of malicious code.

VII. Conclusions and Future Work

Our examination of a subset of *eCos* features showed that for about half of these we can, at least informally, define *feature conditions* which allow deducing feature need from an application's source code. Aiming at automating the evaluation of these conditions, we identified control flow graphs, transformed to a Kripke structure, as an adequate model which carries enough information for subsequent analysis steps.

The branching-time logic CTL was attested to be expressive enough for many but not all feature conditions we found: For a few cases it needs to be extended with free variables, or even supplemented by *configlets*, specialized analysis plug-ins written in a normal programming language. The first preliminary evaluation results with our analysis tool prototype are promising, although we cannot already draw conclusions regarding the scalability of our approach.

Although the configuration automation of about 50% of the examined features may not seem exceedingly successful, extrapolating this to *all* configurable *eCos* features[3] we could still save the application developer about 750 configuration decisions. For parts of the remaining half, especially the features dealing with non-functional properties, the feedback approach of Sincero et al. [8], [26] could offer other means of automatic configuration.

Our results are comparable to an earlier case study we conducted with a modified version of the Berkeley DB [24], an embedded DBMS for which we could automate an even larger portion of configuration decisions – mostly due to simpler API usage patterns and a lower quantity of features dealing with non-functional properties.

More future work includes examining additional *eCos* features in order to gain more confidence on the general soundness of our approach. We also intend to evaluate the outlined extension to CTL, to compare it to other (potentially more powerful) temporal logics, and to integrate the analysis results in the *eCos Configtool*. In order to alleviate the infrastructure developer's task to describe feature conditions, we are also going to look into simpler description languages that can be automatically transformed into temporal logic formulas; Kinder et al. [10] suggest using macro shortcuts for the most commonly used patterns, which may prove to suffice.

Beyond these enhancements and a thorough evaluation of the outlined approach, we are looking into the research question what *modifications* to the infrastructure API are necessary to simplify the analysis task such that a vast majority of features can be automatically configured. Ideally, only a tiny fraction consisting of development time (debugging) and non-functional (runtime safety checks, stack sizes, runtime/code size trade-offs) configuration decisions remains, and we believe this could be achieved by designing the API with this ambition in mind.

References

[1] Carsten Böke. *Automatic Configuration of Real-Time Operating Systems and Real Time Communication Systems for Distributed Embedded Applications*. Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Entwurf Paralleler Systeme, 2004.

[2] Ramakrishna Prasad Chivukula, Carsten Boeke, and Franz J. Rammig. Customizing the configuration process of an operating system using hierarchy and clustering. *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '02)*, 0:0280, 2002.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(8):244–263, 1986.

[3](which may be, due to the small feature sample we examined, not perfectly valid)

[4] Edmund M. Clarke, Orna Grumberg, and Doron A. Pele. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, England, 3 edition, 2001.

[5] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[6] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison-Wesley, May 2000.

[7] A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

[8] Wanja Hofer, Julio Sincero, Daniel Lohmann, and Wolfgang Schröder-Preikschat. *Configuration of Non-Functional Properties in Embedded Operating Systems: The CiAO Approach*. Methodologies for Non-Functional Requirements in Service Oriented Architecture. IGI Global, Hershey, PA, USA, 2010. (To Appear).

[9] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, November 1990.

[10] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In Klaus Julisch and Christopher Krügel, editors, *GI SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2005)*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187. Springer-Verlag, 2005.

[11] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, 1980.

[12] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[13] LLVM homepage. http://www.llvm.org/.

[14] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX Association.

[15] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the EuroSys 2006 Conference (EuroSys '06)*, pages 191–204, New York, NY, USA, April 2006. ACM Press.

[16] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Lean and efficient system software product lines: Where aspects beat objects. In Awais Rashid and Mehmet Aksit, editors, *Transactions on AOSD II*, number 4242 in Lecture Notes in Computer Science, pages 227–255. Springer-Verlag, 2006.

[17] Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[18] Matthias Meier, Michael Engel, Matthias Steinkamp, and Olaf Spinczyk. LavA: An open platform for rapid prototyping of MPSoCs. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL '10)*, Milano, Italy, 2010. IEEE Computer Society Press. to appear.

[19] Toshio Ogiso, Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Software tamper resistance based on the difficulty of interprocedural analysis, August 2002.

[20] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the EuroSys 2008 Conference (EuroSys '08)*, pages 247–260, New York, NY, USA, 2008. ACM Press.

[21] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for c systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, 1994.

[22] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[23] Marko Rosenmüller, Norbert Siegmund, Horst Schirmeier, Julio Sincero, Sven Apel, Thomas Leich, Olaf Spinczyk, and Gunter Saake. FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In *Workshop on Software Engineering for Tailor-made Data Management*, pages 1–6. School of Computer Science, University of Magdeburg, March 2008.

[24] Horst Schirmeier and Olaf Spinczyk. Tailoring Infrastructure Software Product Lines by Static Application Analysis. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, pages 255–260. IEEE Computer Society Press, 2007.

[25] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is The Linux Kernel a Software Product Line? In Frank van der Linden and Björn Lundell, editors, *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, Kyoto, Japan, 2007.

[26] Julio Sincero, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Towards Tool Support for the Configuration of Non-Functional Properties in SPLs. In *Proceedings of the 42nd Hawai'i International Conference on System Sciences (HICSS '09)*, Waikoloa, Big Island, Hawaii, January 2009. IEEE Computer Society Press.

[27] Olaf Spinczyk and Holger Papajewski. Using Feature Models for Product Derivation. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, pages 5–6, Kyoto, Japan, September 2007. Tutorial Description.

[28] Bart Veer and John Dallaway. *The eCos Component Writer's Guide*. Free Software Foundation, Inc., 2001. http://ecos.sourceware.org/docs-3.0/.

[29] Charles Zhang, Dapeng Gao, and Hans-Arno Jacobsen. Towards just-in-time middleware architectures. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 63–74, New York, NY, USA, 2005. ACM.