SmartFixer: Fixing Software Configurations based on Dynamic Priorities

Bo Wang[12], Leonardo Passos[3], Yingfei Xiong[12]*,
Krzysztof Czarnecki[3], Haiyan Zhao[12], Wei Zhang[12]

[1]Key Lab. of High Confidence Software Technologies (Peking University) Ministry of Education Beijing 100871, China

[2]Institute of Software School of EECS Peking University Beijing 100871, China

[3]Generative Software Development Lab University of Waterloo Waterloo, Canada

{wangbo07,xiongyf04,zhhy,zhang}@sei.pku.edu.cn {lpassos,kczarnec}@gsd.uwaterloo.ca

ABSTRACT

Large modern software systems are often organized as product lines, requiring specialists to configure variability models before delivering a product. Variability models capture both the commonality and variability of different products, and help detect the configurations errors. Existing approaches can recommend fixes for the errors automatically. However, the recommended fixes are sometimes large and complex, and existing approaches lack guidance to help users identify a desirable fix. This paper proposes an approach to provide such guidance using dynamic priorities. The basic idea is to first generate one fix, and then gradually reach the desirable fix based on user feedback. To this end, our approach (1) automatically translates user feedback into a set of implicit priority levels on configuration variables, using five priority assignment and adjustment strategies and (2) efficiently generates potential desirable fixes by calculating new values for the variables with low priority. The experiments on real variability models show that we can reduce up to 89% of the fixes, and up to 98% of the variables shown to the user, compared to when no priorities are used.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*

General Terms

Design,Algorithm

Keywords

Priority, Fixing, Configuration Error

*corresponding author

SPLC '13 Tokio, Japan

1. INTRODUCTION

Large modern software systems, such as enterprise resource planning and operating systems, are often organized as product lines, requiring specialists to assign values to a set of variables before delivering a product. The assignments of variables are called a *configuration*. The configuration usually has to satisfy various constraints, stemming from the application domain and the underlying implementation. If a constraint is not satisfied, the product may fail to compile, or worse, may misbehave or crash at runtime. Without a proper detection mechanism, satisfying all constraints is hard. On the one hand, many systems are not well documented, and thus the stakeholders often do not know all the constraints existing on the system. On the other hand, the number of variables and constraints can be very large. For example, operating systems such as eCos and Linux contain thousands of variables and constraints (Linux with over 6000 variables and 10000 constraints; eCos with over 1000 variables and 1000 constraints) [15, 11, 2]. In such a setting, it is very easy to miss some constraints during configuration. From an analysis of 500 configurations of large scale software, Yin et al. [19] finds that a significant percentage (27%) of user reported issues are related to configuration errors.

To solve this problem, configurable software may use variability modeling languages and configuration tools (called *configurators*). A variability modeling language allows developers to document the variables and constraints in a software product line as a variability model, and the corresponding configurator translates the variability model into an interactive configuration interface, where an error will be reported when a constraint is violated. Examples of variability languages include Linux Kconfig,[1] eCos CDL,[2] and feature models [8].

With variability modeling languages and configurators, we can detect errors early. However, users still have to resolve the errors, which is also not an easy task. The constraints in variability models can be very complex and highly inter-

[1]`http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt`

[2]`http://ecos.sourceware.org/docs-2.0/cdl-guide/cdl-guide.html`

connected. Berger et al. [2] report that a variable in the Linux model can connect up to 56 other variables via its constraints. Thus, it can be time-consuming and laborious to analyze the constraints and figure out a solution. Hubaux et al. [7] report that 20% of the Linux users sometimes need "a few dozen minutes" to figure out a solution to satisfy a violated single constraint. Furthermore, even if the user has figured out a solution, he has to navigate to the corresponding variables and make changes. As variability models contain up to thousands of options, navigation is not easy.

To assist users in resolving errors, researchers have proposed automated approaches that suggest a list of fixes for an error [17, 18]. A fix is a set of changes that, when performed on the configuration, resolve the current error. However, the recommended fixes in these approaches are sometimes large in number and size. Without proper guidance, it is difficult for users to identify a desirable fix from such an overwhelming list of fixes. For instance, Xiong et. al. [18] report that fix lists for eCos configurations contain up to nine fixes, and some fixes change up to nine variables (when using the propagation strategy defined in [18]). In total, 17% of all fix lists contain changes over more than 10 variables, which can easily overwhelm the user. When faced with these fix lists, users have to read through potentially large lists and decide the most desirable fix.

We propose a novel approach to guide users in fixing variability model configuration errors based on dynamic priorities. The basic idea is to first generate one fix, and then to gradually reach the desirable fix based on user feedback. To this end, our approach (1) automatically translates user feedback into a set of implicit priority levels on variables, using five priority assignment and adjustment strategies and (2) efficiently identifies potentially desirable fixes that change only the variables with low priorities. The priorities are not exposed to the users, thus avoiding the potential burdens to them. The main contributions of the paper are as follows:

- We introduce priorities to recommend fixes. The priority of a variable represents the likelihood of its current value being desirable to the user. We first recommend fixes that change variables with less desired values (variables with low priority). If users are not satisfied, we update the fixes by including variables with higher priority.
- We design a mechanism that dynamically adjusts the priorities of the variables by translating feedback. Users provide feedback by accepting and rejecting the changes to individual variables in the fixes. Rejecting a change to a variable means that the current value of the variable is correct. Therefore, we increase the priority of the variable. Users can use three rejection duration scopes (*durations*, for short) namely, *fix*, *error*, and *permanent*, to specify if the variables of the rejected change can be changed in the future. Five strategies are introduced to automatically translate the rejections and durations into priorities.
- We have implemented a tool, SmartFixer, and evaluated our approach by fixing errors in configurations of five real variability models. For configurations with errors that result in large fixes, on average, our approach is able to reduce the number of fixes presented to the user by 22% and reduce the number of variables presented to the user by 23%. Moreover, we are able to reduce the number of fixes by a maximum of 89% and the number of variables by a maximum of 98%.

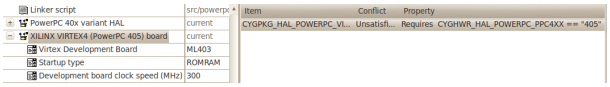


Figure 1: An error whose fix is an assignment change

In our previous work [12], we fix errors in variability models through manually assigning priorities to the constraints in the variability models. This paper is inspired by such work. This paper focuses on fixing configuration errors, and introduces the self-adaptive mechanism to assign priorities automatically.

The rest of this paper is organized as follows: Section 2 defines the terminologies used throughout the paper. Section 3 gives an overview of our approach and illustrates it with an eCos example. Section 4 describes how we recommend fixes and assign and adjust priorities automatically. Section 5 provides an overview of our implemented tool—SmartFixer. Section 6 evaluates how well our approach works, when compared with existing proposals. Section 7 discusses the threats to validity of our evaluation. Section 8 describes related work. Section 9 concludes the paper and highlights future work.

2. BACKGROUND

We use eCos-based projects to illustrate and evaluate our approach. eCos is a configurable embedded operating system. To capture the rich variability of the different target hardware to which the operating system can be deployed, eCos relies on variability models encoded in CDL (***C****omponent* ***D****efinition Language*), a domain-specific language created as part of the eCos build infrastructure. In projects that rely on such infrastructure (eCos-based projects), the configuration process is supported by means of a graphical configurator: the *eCos configtool*. When no confusion arises, we refer to eCos configtool as eCos for short. In eCos, an error is reported when the user configuration violates a constraint. Figure 1 shows an error reported by eCos, where the Property column shows the violated constraint.

Existing approaches provide fixes for configuration errors. A fix consists a set of changes. A change is a re-assignment for a variable. Executing a fix will set new values to variables, thus removing errors. For instance, setting `PowerPC HAL` to `"405"` fixes the error shown in Figure 1. eCos has a built-in fix generator that provides users with at most one fix for a given error. Xiong et al. [18] report that the generated fixes are incomplete, and propose a new fix generation algorithm to overcome the problem.

3. APPROACH OVERVIEW

Our approach relies on assigning priorities to variables. The priority of a variable represents *the likelihood of its current value being desirable to the user*. If a fix tries to change a

variable and the user rejects this change, the current value of the variable is more likely to be a desirable one. Whenever the user rejects a change, we adjust the priority of the variables in that change.

There are two basic ideas in our approach:

1. Identify fixes that only change variables with less desired values (i.e., variables with low priority).
2. Dynamically adjust the priority of variables through implicit translation of user feedback.

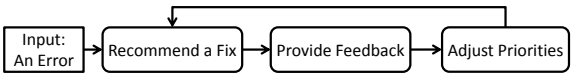


Figure 2: The interactive fixing process for an error

These two basic ideas are reflected in the fixing process, as shown in Figure 2. Users first choose an error to fix. Our approach recommends a potentially desirable fix for that error, according to variables' current priority. Users examine the recommended fix and provide feedback by accepting and rejecting changes. We translate such feedback into priorities, adjust them accordingly and identify an improved fix for users. This interactive fixing process continues until users accept the recommended fix. Next, we illustrate this iterative fixing process with a concrete example taken from ReconOS,[3] an open source eCos-based project.[4]

Example.
Our example is based on an error in a real world configuration. Xiong et al.'s approach [18] generates five fixes: four of the fixes contain four changes; one fix contains two. Here is one of the fixes, containing four changes:

```
1.  POSIX timers = false;
2.  POSIX message queues = false;
3.  POSIX signal configuration = false
4.  Allow empty queue notification = false
```

Recommend a potential desirable fix.

When generating potential desirable fixes, we set a threshold to confine the fix generation scope. Only the variables whose priorities are under the threshold can be changed to fix the error. Our approach identifies a fix for these changeable variables. If no fixes can be found under the current threshold, we increase the threshold and identify a new fix. For each error, we initially set the threshold to 0. The threshold realizes the first basic idea, and it decreases the number of variables in the fixes.

For example, suppose the priority of "POSIX timers" is 2. If the threshold is 1, this variable is unchangeable under the current threshold. Hence, our identified fixes do not propose changes for this variable, i.e., the example fix would not be generated. If the priority of all the variables in the five changes is 2, and the threshold is 1, no fixes can be generated. In this case, we increase the threshold by 1 (i.e., to 2), and generate another fix.

Provide feedback and adjust priorities.

After a fix is recommended, the users provides feedback to each change in the fix.

When users are satisfied with a change in the proposed fix, they can **accept the change**.

When the user is not satisfied with a change, they can **reject the change** with three different durations:

- *fix*. The *fix* duration means that the current change proposed by the fix should be rejected, but future fixes can still propose changes for the related variable. For example, suppose that there exists an integer variable whose current value is 1, and a fix proposes to change its value to 10. If the user does not want to change the variable to 10 but considers it changeable to a smaller value, he should choose *fix* duration. As another example, in our example fix, if the user believes that selecting "POSIX message queues" is correct, but still wants to allow the option of assigning "false" in the future, he can reject the second change with *fix* duration.
- *error*. The *error* duration means that the current value of the variable is correct and should not be modified until the error is fixed. For example, if the user thinks the "POSIX signal configuration =true" is correct, and does not want any fix to change its value while fixing the current error, he can reject the fifth change with *error* duration.
- *permanent*. The *permanent* confirmation means that the current value of the variable is always correct and should not be modified during the whole configuration process. For instance, if users believe "POSIX signal configuration" should always be true, they can reject the third change with *permanent* duration.

We automatically assign and adjust the priorities of the rejected and accepted variables, according to five strategies (see section 4.2). When the error is not fixed, we generate another potential fix based on the new priority.

Constructing priority hierarchy.

When an error is fixed, the dynamically adjusted priorities are stored for future fix generation. These priorities serve as the basis for generating potentially desirable fixes.

For instance, suppose that during the process of fixing the error in our example, the priority of variable "POSIX message queues", "POSIX signal configuration" and "POSIX signal configuration" is increased. When we need to fix another error, which could be achieved by changing the value of these three variables, we do not include these three variables if their priorities is greater than the current threshold.

[3]http://reconos.de/
[4]Due to space limit, we pick an error that only needs Boolean assignment changes. Our approach also supports errors that need non-Boolean changes.

Handling no fix.

When some of the variables are rejected with *error* and *permanent* duration, they cannot be included in any fix during the fixing process for the current error. This may lead to the situation where no fix is generated. If this happens, we provide the user with a list of variables with *error* and *permanent* duration, and ask him to change the values, durations or both. After users have changed the durations, we identify a new fix for them.

4. RECOMMENDING FIXES AND ADJUSTING PRIORITIES

4.1 Fix recommendation

Our fix recommendation approach requires integrating with an existing fix generator. For integration purposes, we only require the fixer to take an error, a set of constraints, a set of variables, and a set of assignments to those variables, and produce a set of fixes as output.

In our approach, when there is a variable v whose priority is higher than the current threshold, we create a new constraint $v = x$, where x is the current value of v. These created constraints, together with the original constraints in the variability model, are given to the fix generator. In this way we ensure that the variables with a higher priorities are not changed by the generated fixes.

From the set of returned fixes, we randomly select one and present it to the user; as our evaluation shows, randomizing the fix selection already gives satisfactory results. In future, we aim to study alternative selection strategies such as selecting the one changing the smallest number of variables.

4.2 Automatic priority assignment and adjustment

In our approach, we provide five strategies to assign and adjust priorities:

Strategy 1: *The priorities of the variables with newly assigned values are set to 0.*

If a variable is assigned with a new value, no change has been rejected since the variable was given its new value. Therefore, we set the priority to 0. In that case, we also reset the threshold to be 0, since we should re-attempt to find fixes involving only variables with priority zero. From this, the whole fixing process restarts, and other variables with priorities greater than 0 will appear as the threshold increases along the way. In our approach, before any error fixing process begins, the priority of all the variables are set to 0.

Strategy 2: *When a variable is rejected with fix duration, add 1 to its current priority.*

If a variable is rejected with *fix* duration, we increase its priority by 1. When the threshold increases and is higher than the variable's priority, future fixes can include this variable.

Strategy 3: *When a variable is rejected with error duration, set its priority to (threshold+1).*

If a variable is rejected with duration *error*, it should not be included in any fixes during the fixing process of the current error. Therefore, we set its priority to (threshold+1). If the threshold increases, we also update priority of the variables with *error* duration.

Strategy 4: *When a variable is rejected with permanent duration, set its priority to* $\langle \mathsf{MAX} \rangle$.

If a variable is rejected with duration *permanent*, it should not be included in any fixes during the configuration process. Therefore, we set its priority to $\langle \mathsf{MAX} \rangle$. $\langle \mathsf{MAX} \rangle$ represents the highest priority. Therefore, no fixes can include such variables.

Strategy 5: *If the duration of a variable is reset from error or permanent duration to fix duration, set its priority to 0.*

Variables with *error* and *permanent* durations may cause no fixes. Users change their durations to *fix* to allow these variables back to future fixes, regardless of the threshold value. In this case, we set their new priority as 0.

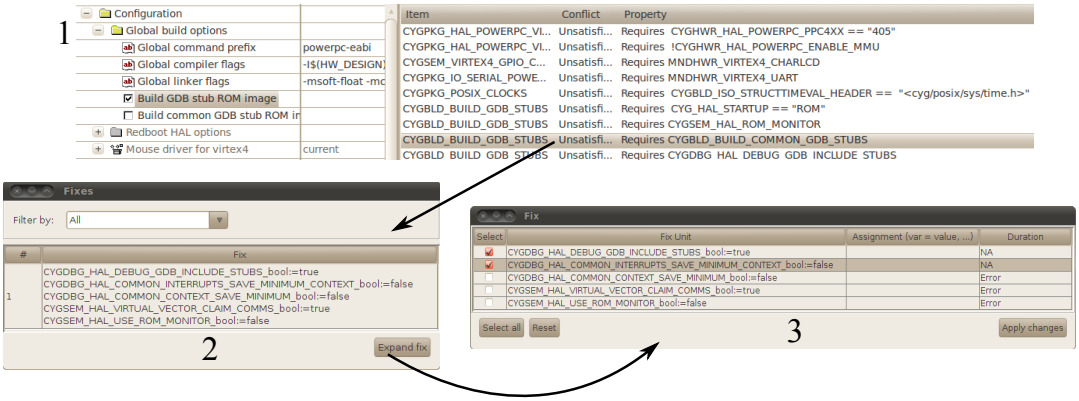
To better explain how we adjust priority and identify the desirable fix, we give the pseudo code to explain the main points in our approach.[5]

5. IMPLEMENTATION

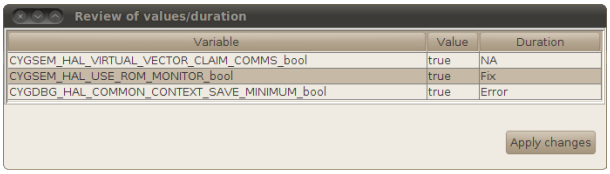
We have implemented the proposed dynamic priorities strategy as an extension to the eCos configurator.[6] In particular, our implementation replaces eCos built-in conflict support, which might provide users with incomplete fixes. The process described in Section 3 is realized in the following GUIs (see Figure 3a): users are first presented with a list of errors (GUI-1), from which they can select one to solve. Based on priorities and on the results returned by the underlying fixer, we produce a potentially desirable fix, which is then presented to the user (GUI-2). Users might then accept the fix, or provide the tool with some feedback on the proposed changes. By expanding a conflict, users have a fine-grained control of which changes should be accepted or rejected. It is also possible that a user fully rejects all changes, or chooses a certain rejection duration. In all cases, priorities are updated properly. In case no fix exists and the error persists, we ask the user to review past rejections. In Figure 3b, the user has changed the value of a variable, which automatically sets the duration to be empty (N/A), since the new value has never been rejected before. In addition, the user has also changed a duration from error to fix.

In our current implementation, we rely on eccFixer[7] as the backend fixer. eccFixer can generate complete sets of fixes for errors that may occur in the configuration of CDL variability models [18]. Since we validate our approach over such models, eccFixer is a suitable fixer for integration. Moreover, eccFixer supports non-Boolean constraints, which often appear in CDL models [11].

[5] `http://gsd.uwaterloo.ca/node/382`
[6] `https://code.google.com/p/smart-fixer/`
[7] `http://gsd.uwaterloo.ca/eccfixer`



(a) SmartFixer: Interactive process GUI for fix resolution



(b) SmartFixer: Review of previously assigned values and durations

Figure 3: SmartFixer GUI

6. EVALUATION

We present our evaluation using data from four open source projects whose variability models are encoded in CDL. In our evaluation, we compare our results with those obtained from eccFixer. The characteristics of each variability model are shown in Table 1.

In CDL, two types of constraints might cause errors: *requires* and *legal-values*. Requires constraints impose conditions among variables that depend on each other. Legal value constraints restrict valid values a variable may assume, as determined by its specified domain. eCos allows incorrect values to be assigned to variables, but it warns users that such values violate the domain of the given variable. Our analysis concerns only requires constraints, as users are unlikely to provide values that do not conform to a variable's domain since they will instantly receive a warning message if they do so.

We rely on these projects as evaluation subjects for two reasons. First, they contain real-world variability models targeting different requirements. For instance, Talktic[8] is a tiny virtual machine for Atmel Micro Computer; ReconOS[9] is an operating system for reconfigurable hardware; redboot4lpc[10] is a port of the Redboot[11] system boot monitor to ea2468 architecture; and gps4020 is the target architecture for the GPS system developed as part of the Portland State Aerospatial Society[12]. All these projects are based on eCos as their operating system. Second, these projects were used in the evaluation of eccFixer, thus allowing a direct comparison of results.

The evaluation simulates the configuration process from an initial configuration, which already contains errors (Table 1), to a final consistent configuration of each project. The initial configurations were obtained in each project repository. The final configurations were obtained from the experimental data that the creators of the eccFixer made publicly available.[13] Figure 4 presents the intuition behind our evaluation procedure. The top timeline concerns results generated by eccFixer; the bottom one represents the application of our approach.

The procedure starts by randomly choosing an error to solve, given the initial configuration. Next, we request eccFixer to calculate a fix list FL_1, from which we randomly choose a fix. We then simulate the decisions users are likely to make in the interactive process. One *interactive process* concerns only one error. While in the interactive process, several *iterations* may occur, each concerning a single fix. In each iteration, users provide feedback for the proposed changes of a fix. In each iteration of an interactive process in Figure 4, we simulate the user decisions that provide feedback for one specific fix. In case the selected error is not removed after an iteration, we call eccFixer to generate a new fix list according to the current priorities of variables and associated values. From it, we randomly choose a fix, and a new iteration starts. For each interactive process in our timeline, we keep track of how many variables each randomly proposed fix contained (*nvars*), along with the number of fixes (*fls*)

[8] `http://code.google.com/p/talktic/`
[9] `http://reconos.de/`
[10] `http://github.com/laltrasponda/redboot4lpc`
[11] `http://sourceware.org/redboot/`
[12] `http://psas.pdx.edu/`

[13] http://gsd.uwaterloo.ca/eccfixer

Project	Architecture	Variables	Constraints	Errors (initial conf.)
ReconOS	virtex4	933	330	56
	xilinx	765	272	48
redboot4lpc	ea2468	658	96	8
Talktic	aki3068net	817	195	26
PSAS	gps4020	535	85	12

Table 1: Characteristics of variability models of each project

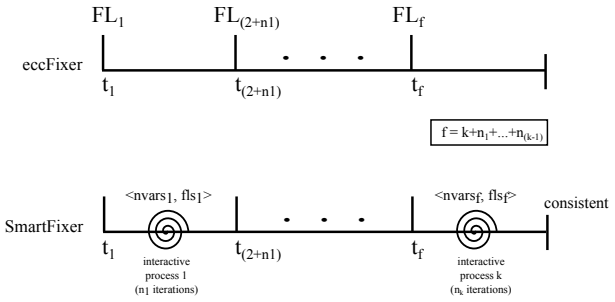


Figure 4: Evaluation procedure

users have been exposed to. Since in each iteration we only propose one fix, *fls* is equal to the number of iterations used in resolving the error.

After n_1 iterations, the error is resolved, and our timeline reaches time t_{2+n1}. In that case, we compare the final values of fls_1 and $nvars_1$ with the fix list size of FL_1 and the number of variables it contains. If there are others errors, the whole process is repeated until the configuration reaches a consistent state at time t_f.

Before describing the simulation process, we define range fixes. Range fixes are a generalization of concrete fixes, which eCos and some other fix generation algorithms use. A range fix is a set of changes (called fix units in [18]) that remove an error. A fix change can be of two types: an *assignment* or a *range* change [18]. An assignment change $a := c$ contains only one variable and c is a constant, such as `"405"` in the previous example. A range change $vars : exp$ may change a set of variables *vars*, and *exp* is a logic expression defining the ranges of the variables. For instance, the violated constraint shown in Figure 5 can be fixed by this range change: `Node pool size :` ≥ 18. Range changes are important in the context of eCos, because CDL has support for numeric variables and a rich set of arithmetic operators [11]. In the rest of the paper, we will refer to a range fix as a fix for short.

The simulation of user decisions in the interactive process is performed by first handling fix changes with a single variable, only then inspecting fix changes with two or more variables. In fix changes with a single variable, two situations may occur:

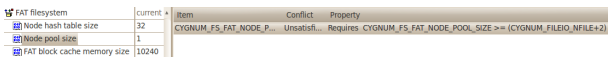


Figure 5: An error whose fix is a range change

1. if there exists an assignment change of the form $a := v_{fix}$, and a is currently set to v_c, and the variable has value v_f in the final configuration, three situations might occur: (a) $v_{fix} = v_f$ and $v_c \neq v_f$: we accept the change; (b) $v_{fix} \neq v_f$ and $v_c = v_f$: we reject the fix and set its duration to *error*, thus capturing a possible uncertainty of users not being totally confident whether the current value should stand throughout the configuration process; (c) $v_{fix} \neq v_f$ and $v_c \neq v_f$: we reject the fix and set its duration to *fix*. Since the value of the feature still has not yet converged to its final value, a fix duration guarantees that new fixes for that variable will continue to be proposed.

2. As in the previous situation, we also have three cases to cover range changes with one variable. The only difference is checking whether the final value v_c is in the proposed range change.

After all changes with one variable have been processed, we inspect the ones with two or more variables. In these cases, the type of change can only be range change. We check whether the values in the final configuration are within the range of the change. For example, given a change (a; b) : a > b + 2, we replace a and b with their associated values in the final configuration, and then check if a > b+2 holds. If the final values are within the range, we accept the change, and all variables receive their associated values in the final configuration. Otherwise, the change is rejected, and the duration is set to fix.

Note that in all situations we set the durations in a conservative way, which do not favor our algorithm.

The two metrics we used in our experiment, namely fix list size and the total number of variables in all proposed fixes, allow us to assess how much cognitive effort is required from users: if more fixes are presented, users have to inspect each of them to make a sound decision; having more variables to inspect in any given fix also increases such effort. Figure 6a constrasts the size of proposed fix lists (y-axis) obtained for each error (x-axis) in the virtex4 configuration. Note that the fix list size for SmartFixer is the *fls* value which we have defined, showing how many fixes the user has been exposed in all iterations resolving an error, equal to the number of iterations used in resolving an error. The number of fixes is decreased in $\approx 31\%$ of the errors. In average, there is a reduction of $\approx 22\%$, with a maximum reduction of 89% in the number of fixes (see errors 16, 19, 25 and 29). Each decrease in the fix list size is accompanied by a reduction on the number of variables to inspect, as shown in Figure 6b. In average, the total number of variables is decreased by $\approx 23\%$ in average, with a maximum reduction of $\approx 98\%$ (see error 25). For xilinx, as shown in Figure 7, the number of fixes

is decreased in ≈ 28% of the errors. In average, there is a reduction of ≈ 16%, with a maximum reduction of 2/3 in the number of fixes (see errors 10, 13, 14, 30, 33 and 37). The total number of variables is decreased by ≈ 18% in average, with a maximum reduction of ≈ 86% (see errors 30, 33 and 37). Note that for most errors, the fix lists are already small, so little reduction is obtained from SmartFixer.

For the other three architectures, the fix lists generated by eccFixer are already short. Our approach further reduces these fix lists. However, we do not notice significant reduction. For the complete results, refer to our project web page [14].

These results suggest that the proposed dynamic priorities perform better than eccFixer when there is a fix list with more than one variable and previous fixes were already proposed and contained such variables, for which the user provided some feedback. Saving such feedback in the form of priorities and associating them with these variables allows us to effectively eliminate the shortcomings of fixers generating complete sets of fixes by reducing the number of fixes and related variables presented to the user.

7. THREATS TO VALIDITY

Our first threat is that we restricted our analysis to projects whose variability models are encoded in CDL. We claim, however, that CDL is one of the richest languages for variability modeling currently available, supporting a rich set of operators and constraints [11]. Existing research already confirms this: in a comparison of CDL, Linux Config Language, and feature models, Berger et. al. shows that CDL specifications are more complex in the type of constraints and operators involved [2]. We therefore believe that this is a minor threat and should not prevent the generalization of our approach to other variability modeling languages and their configurators.

Another threat is that other domains may have variability models with other characteristics. One such characteristic is the cross-tree constraint ratio (CTCR), which is the percentage of variables of a model that participate in its cross-hierarchy constraints. The CTCR is a key characteristic determining the hardness of finding a configuration satisfying the model constraints [10]. Berger at al. [2] have shown that the eCos model has higher CTCR than the feature models collected in the popular SPLOT [15] repository. Thus, we believe that the approach will work for a wide range of models.

The third threat regards our experimental procedure. At each time a decision has to be made, we use the values from the final configuration to guide our decision. It is not clear, however, to which extent users know what the final configuration should be. We also made conservative choices regarding durations. It is not clear which durations users would choose. However, our choices do not favor our algorithm. We aim to address these issues in the future by conducting studies with experienced users in industrial settings.

8. RELATED WORK

Several approaches have been proposed to generate fixes for configuration errors. In the eCos configurator [16], an internal fix generator recommends a fix online when a configuration error is detected. However, no completeness property is guaranteed for the generated fixes. White et al. [17] adopt constraint solvers for fix generation, enabling the generation of minimal or cost-optimal fixes. However, this approach still generates only one fix per configuration. Xiong et al. [18] propose the concept of range fix, and further incorporate Reiter's theory of diagnosis to generate a complete set of fixes rather than one fix. Our approach builds on top of the fix generation approaches, but we further introduce priorities to break large fix lists into small, guided steps so as not to overwhelm users.

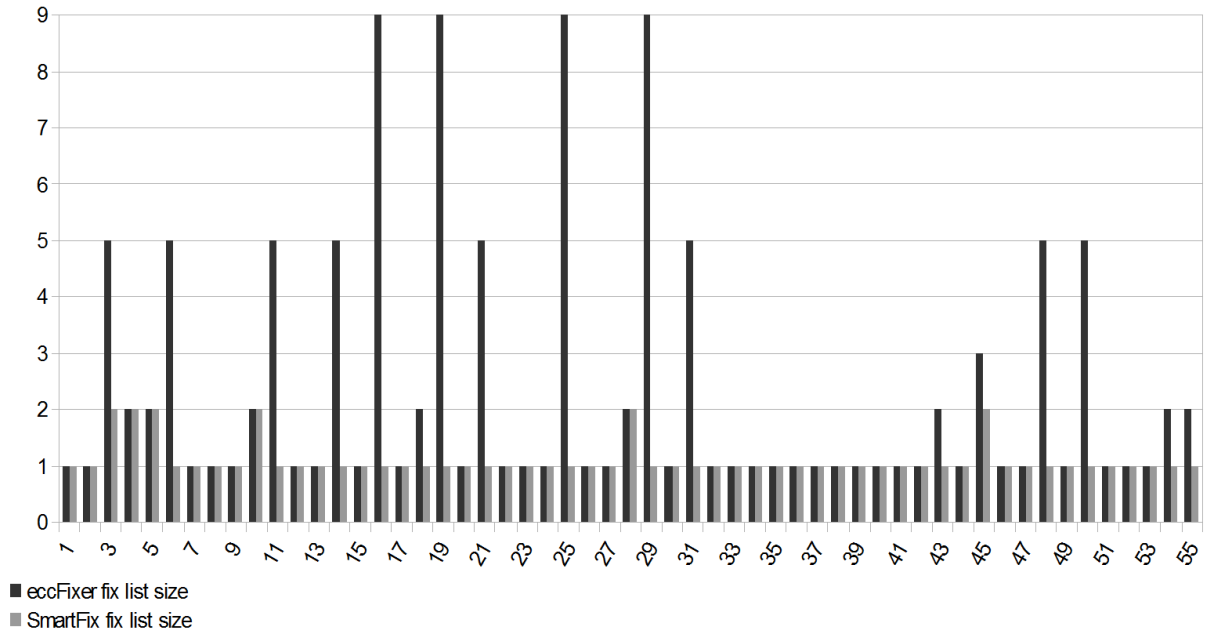
Priorities have been previously used to resolve errors. The constraint hierarchy theory [3] and the SkyBlue algorithm [13] assign fixed priorities to constraints, and automatically resolve any error by disabling constraints with low priorities. Based on SkyBlue, Wang et al. [12] propose an approach to automatic resolution of errors in feature model construction. In [12], the priorities of constraints are assigned manually by users. Our approach focuses on the varibility model configurations, and the priorities of variables are assigned through automatically interpreting the feedback from users. Another similar approach is Junker's QuickXplain algorithm [14]. This algorithm assumes a total order over constraints, and disables the smaller constraints to resolve errors. Felfernig et al. [5] applies QuickXplain to configuration knowledge bases so as to explain and resolve configuration errors. A common characteristics of all these approaches is that the priorities are static; no change to the priority will occur once assigned. On the other hand, the priorities in our approach are dynamic and automatically adjusted based on user feedback. Thus, our approach allows us to incorporate user feedback adaptively.

Apart from error resolution, another direction is to avoid errors. Czarnecki et al. [4] propose staged feature model configuration. A staged configuration divides the configuration process into several steps, and in each step only a small portion of variability model is exposed to the user. As the user is not exposed to a large model, the possibility of introducing error is reduced. Another branch of approaches is valid domain computation [6, 1, 9]. These approaches assume that all variables start with an unknown state. When the value of a variable is determined by other variables during the configuration, the configurator automatically sets these variables. In this way, no error can be introduced in configuration. Different from these approach, our approach assumes that errors can be introduced, and focuses on how to resolve errors. This setting has practical value as many real world configurators rely on the reconfiguration scenario, where users freely modify a complete initial configuration, either default one or one loaded form disk [2].

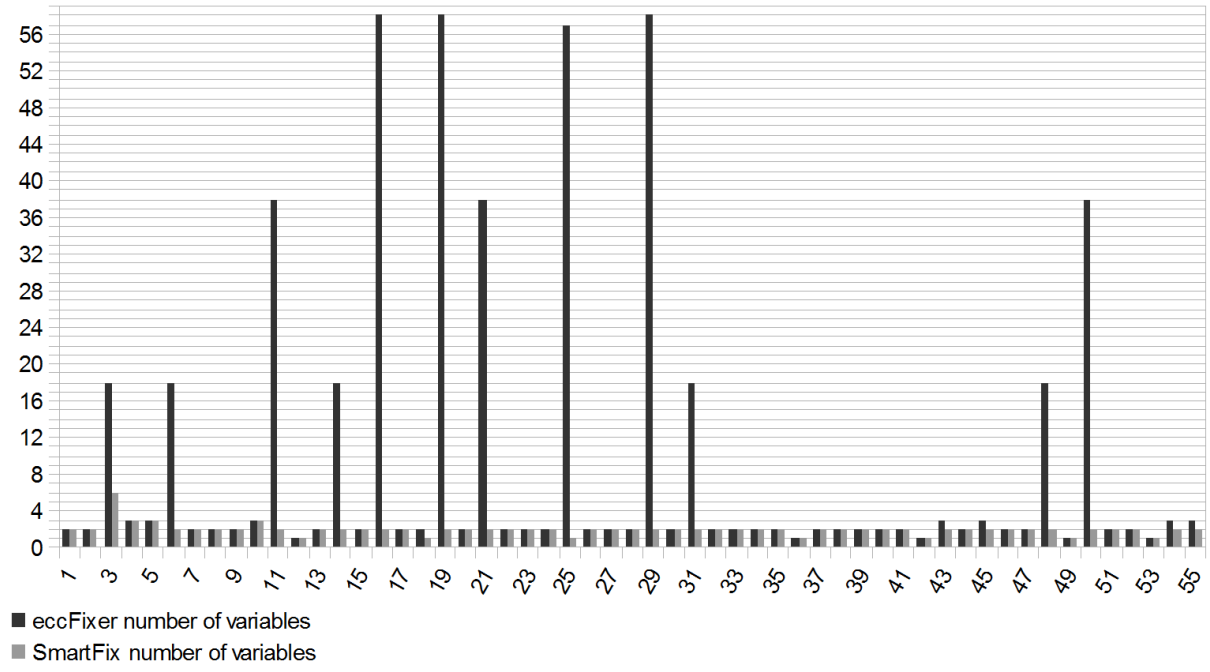
9. CONCLUSION AND FUTURE WORK

We have proposed a dynamic priority based approach to help users identify their desirable fixes efficiently. We define the priority and assign it based on the user feedback. For every error, we identify a potential desirable fix from the variables lower than the threshold. Then users provide feedback through three types of durations. This feedback

[14] `https://code.google.com/p/smart-fixer/`
[15] `http://www.splot-research.org/`



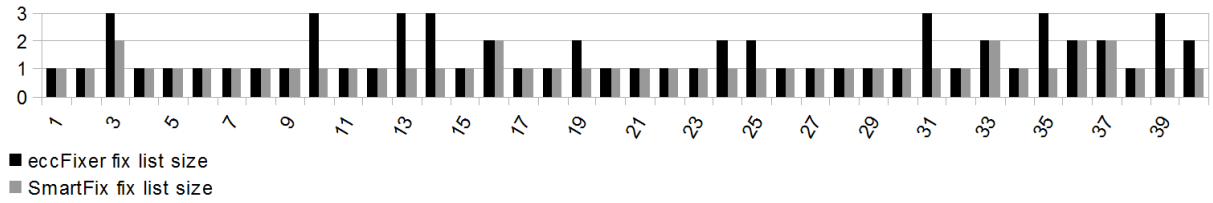
(a) Fix list size
(the fix list size (*fls*) of SmartFixer is all fixes exposed to the user in resolving an error, equal to the number of iterations used in resolving the error)



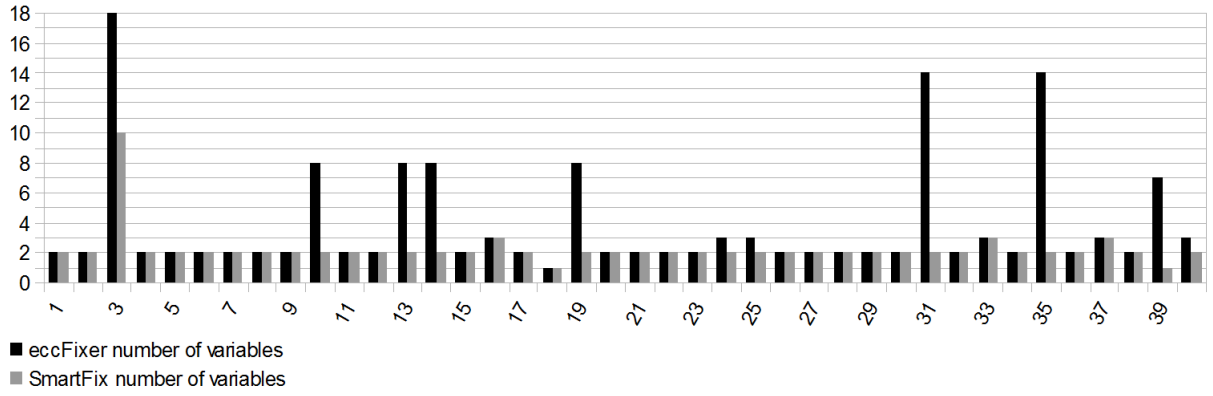
(b) Number of variables

Figure 6: Experimental results for virtex4

is translated into priorities. If no more fixes can be found under the current threshold, we increase the threshold to generate new fixes. When no fixes are generated, caused by having variables with *error* and *permanent* durations, we ask users to change durations or values or both. With errors fixed, we construct a priority hierarchy, which will decrease the size of the fixes for the following errors. We also successfully implemented our idea on the eCos configurator, extending it with SmartFixer. We have evaluated our approach on real variability models and configurations. The results show that we can reduce up to 89% of the fixes, and up to 98% of the variables presented to the user, compared to when no priorities are used. Our idea is not bound to any existing fix generator. In future work, we plan to evaluate this approach in user experiments; currently we don't know whether a user typically has the idea of priority or not, nor



(a) Fix list size



(b) Number of variables

Figure 7: Experimental results for xilinx

do we know if the invisibility of priorities causes problem to the user.

Acknowledgment

This work was supported by the National Natural Science Foundation of China under Grant No.61202071 and 61121063, the National Basic Research Program of China (973) under Grant No. 2011CB302604, the High-Tech Research and Development Program of China under Grant No.2012AA011202, the Key Program of Ministry of Education, China under Grant No.313004, and CAPES under Grant BEX 0459-10-0.

10. REFERENCES

[1] D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[2] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In C. Pecheur, J. Andrews, and E. D. Nitto, editors, *ASE*, pages 73–82. ACM, 2010.

[3] A. Borning, B. N. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.

[4] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[5] A. Felfernig, G. E. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. pages 146–150, 2000.

[6] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *PETO*, pages 131–138. DTU-tryk, 2004.

[7] A. Hubaux, Y. Xiong, and K. Czarnecki. A user survey of configuration challenges in linux and ecos. In *VaMoS*, 2012.

[8] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

[9] M. Mendonca. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2009.

[10] M. Mendonça, A. Wasowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *13th International Conference on Software Product Lines (SPLC 2009)*, San Francisco, CA, USA, 2009.

[11] L. T. Passos, T. Berger, M. Novakovic, K. Czarnecki, Y. Xiong, and A. Wasowski. A study of non-boolean constraints in variability models of an embedded operating system. In I. Schaefer, I. John, and K. Schmid, editors, *SPLC Workshops*, page 2. ACM, 2011.

[12] D. C. Petriu, N. Rouquette, and Ø. Haugen, editors. *A Dynamic-Priority Based Approach to Fixing Inconsistent Feature Models*, volume 6394 of *Lecture Notes in Computer Science*. Springer, 2010.

[13] M. Sannella. SkyBlue: A multi-way local propagation constraint solver for user interface construction. In *ACM Symposium on User Interface Software and Technology*, pages 137–146, 1994.

[14] M. Schubert, A. Felfernig, and M. Mandl. Fastxplain:

Conflict detection for constraint-based recommendation problems. In N. García-Pedrajas, F. Herrera, C. Fyfe, J. M. Benítez, and M. Ali, editors, *IEA/AIE (1)*, volume 6096 of *Lecture Notes in Computer Science*, pages 621–630. Springer, 2010.

[15] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The variability model of the Linux kernel. In *VaMoS*, pages 45–51, 2010.

[16] B. Veer and J. Dallaway. The eCos Component Writer's Guide. ecos.sourceware.org/docs-2.0/cdl-guide/cdl-guide.html, 2001.

[17] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. R. Cortés. Automated diagnosis of product-line configuration errors in feature models. In *SPLC*, pages 225–234. IEEE Computer Society, 2008.

[18] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *ICSE*, 2012.

[19] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011.