Range Fixes: Interactive Error Resolution for Software Configuration

Yingfei Xiong, Hansheng Zhang, Arnaud Hubaux, Steven She, Jie Wang, and Krzysztof Czarnecki

Abstract—To prevent ill-formed configurations, highly configurable software often allows defining constraints over the available options. As these constraints can be complex, fixing a configuration that violates one or more constraints can be challenging. Although several fix-generation approaches exist, their applicability is limited because (1) they typically generate only one fix or a very long fix list, difficult for the user to identify the desirable fix; and (2) they do not fully support non-Boolean constraints, which contain arithmetic, inequality, and string operators. This paper proposes a novel concept, *range fix*, for software configuration. A range fix specifies the options to change and the ranges of values for these options. We also design an algorithm that automatically generates range fixes for a violated constraint. We have evaluated our approach with three different strategies for handling constraint interactions, on data from nine open source projects over two configuration platforms. The evaluation shows that our notion of range fix leads to mostly simple yet complete sets of fixes, and our algorithm is able to generate fixes within one second for configuration systems with a few thousands options and constraints.

Index Terms—Consistency management, error resolution, range fix, software configuration

1 Introduction

Configuration is a common task faced by software developers. Modern operating systems, embedded software, and large enterprise systems often expose configurability to handle variations in user and platform requirements. Given a configuration space consisting of a set of options and constraints over the options, the user produces a configuration by deciding a proper value for each option, ensuring all constraints are satisfied.

For example, Linux kernel can be configured to run on different hardware systems and be equipped with different software modules. To configure a kernel, the user typically selects the CPU architecture (e.g., x86 or ARM), the type of file system (e.g., ext4 or JFS), and other options. Various constraints exist on the options. For example, there is an option that allows the kernel to make use of the "CMPXCHG" instruction, which is only available when x86 architecture is selected and 386 family is not selected.

To support the configuration process, large software systems often come with dedicated configuration systems, such as Linux Kconfig [1] and eCos CDL [2]. These configuration systems often provide a specification language, where the configuration options and the constraints over the options are explicitly specified. The configuration system then translates the specification into an interactive configuration interfaces to allow users to specify values for the options. When a constraint is violated, the configuration system will report an error which must be resolved before the configuration becomes operational. Thus, an important constituent of the configuration process is the resolution of errors.

However, resolving an error in large configurable systems is not easy. First, it can be difficult to figure out how an error occurs. Modern configuration systems often introduce hidden constraints, which are specified in the configuration system but are not explicitly presented to the user [3] and can be easily overlooked. For example, in Linux, a child option[1] may inherit the dependencies of its parent options, and these inherited dependencies are usually not directly presented to the user. Second, it can be difficult to fix the error. Constraints in modern configurable systems can be very complex. User-defined constraints often interplay with the hidden constraints [3] introduced by the configuration system, forming very large constraints. One constraint we found in eCos [4]—an embedded operating system—contains 55 variable references and 35 constants, connected by 66 logical, arithmetic, and string operators. Furthermore, constraints often interact with each other. A solution satisfying one constraint may violate other constraints. It is difficult to locate all potentially interacting constraints and consider all constraints together. Third, even if we have figured out a solution, it is still cumbersome to make the actual changes. Modern configurable systems can have large number of options, e.g., the Linux kernel has 6,320 options [3], and the options referred in one constraint are not necessarily shown to the user next to each other. As a result, the user has to navigate across a large number of options, which is a

- *Y. Xiong, H. Zhang, and J. Wang are with the School of Electronics Engineering and Computer Science, Institute of Software, Peking University, Beijing 100871, PR China, and the Key Laboratory of High Confidence Software Technologies, Ministry of Education, PR China. E-mail: xiongyf@pku.edu.cn, zhanghs12@sei.pku.edu.cn, 617072740@qq.com.*
- *A. Hubaux is with ASML, Eindhoven, the Netherlands. E-mail: contact@ahubaux.com.*
- *S. She and K. Czarnecki are with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L 3G1, Canada. E-mail: {shshe, kczarnec}@gsd.uwaterloo.ca.*

Manuscript received 22 Jan. 2014; revised 19 June 2014; accepted 3 Dec. 2014. Date of publication 17 Dec. 2014; date of current version 17 June 2015. Recommended for acceptance by M. Cohen. For information on obtaining reprints of this article, please send e-mail to: reprints.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TSE.2014.2383381

1. In Linux Kconfig, the options are organized as a tree.

0098-5589 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

challenging and time-consuming task. The difficulty of error resolution is also confirmed by empirical studies. In our previous survey with 97 Linux users and nine eCos users, both groups report that error resolution is a major challenge in configuring the respective system [5].

To overcome the difficulty of error resolution, researchers and tool vendors have proposed various approaches. One class of approaches is to automatically generate concrete changes to the options (called a *concrete fix*) in the faulty configuration to resolve the error. Examples in this class are the eCos configurator [2] and White et al.'s automatic diagnosis [6]. However, there is usually more than one way to fix the error, and a single automatically generated concrete fix may not fulfill the user's requirements. Empirical studies on the eCos configurator confirm this: more than 75 percent users have encountered situations where the generated fix is not useful [5], and similar complaints can also be found on the mail list [3].

Another possibility is to give a complete list of all possible concrete fixes to allow the user to choose from. However, modern configurable systems often contain non-Boolean options such as numbers and strings [3], [7], to which a very large space of values can be assigned. As reported by Passos et al. [7], the 116 configuration models in the codebase of eCos contain between 1,159 and 1,312 options, where 53-56 percent of the options are non-Boolean options, including 23-26 percent integer or floating-point numbers, and 28-32 percent strings. Furthermore, among the 916 to 1,269 constraints per model, 750 to 1,095 constraints per model contain non-Boolean operators, such as `>`, `+`, or `substr`. When non-Boolean options and constraints are present, the fix list is usually very long, if not infinite. It would be infeasible for the user to pick a desirable fix from a very long list, not to mention how to generate such a list.

In this paper we approach the error resolution problem from a different angle. Instead of giving concrete fixes, we characterize the solution space of all possible fixes in a compact form, and let the user choose the most appropriate fix. More concretely, our contributions are threefold:

- *Range fixes.* We propose a novel concept, *range fix* (Section 3). Unlike a concrete fix, a range fix tells the user what options should be changed and in what range the value of each option can be chosen. As a result, a set of concrete fixes can be represented by one range fix, and the list of concrete fixes can be compacted into a much shorter and intentional list of range fixes, allowing the user to scan and choose from. Furthermore, we discuss the desired properties of range fixes, which formalize the requirements of the fix generation problem.
- *Fix generation algorithm.* We design an algorithm that generates range fixes automatically and ensures the desired properties of the generated fixes (Section 4). Our algorithm is inspired by the theory of diagnosis [8], [9] that obtains diagnoses from unsatisfiable cores calculated by constraint solvers. In addition, we also discuss how constraint interactions should be handled in our framework (Section 5).

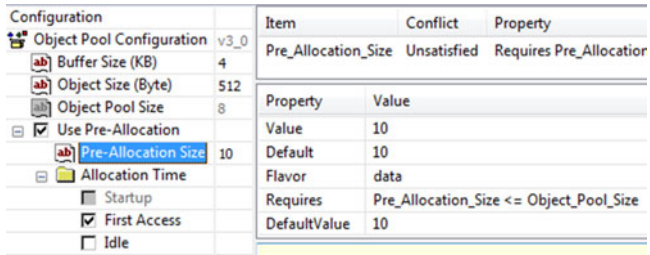


Fig. 1. The eCos configurator. (Option "Pre-Allocation Size" has a requires property. The constraint declared in this property is violated, resulting in an error reported in the upper-right corner.)

- *Evaluation.* We have implemented our algorithm on eCos CDL and Linux Kconfig (Section 6) and evaluated the approach on 310 constraint violations from nine open source projects using eCos and Linux (Section 7). Our notion of range fix leads to mostly simple yet complete sets of fixes, and our algorithm is able to generate fixes within one second for configuration systems with a few thousands options and constraints. We also compare three strategies for handling constraint interactions.

In the rest of the paper, we first motivate and outline our approach in Section 2. After introducing the main contributions, we discuss threats to validity in Section 8, limitations and future work in Section 9, and the related work in Section 10. We conclude the paper in Section 11.

An earlier version of this work appeared at ICSE'12 [10]. Compared to the conference version, this version is significantly extended and improved. First, the previous approach ensures only three properties of range fixes, while the fourth desirable property, "minimality of fix units", is only discussed but not supported in the conference version. In this version, a novel minimization step is added and evaluated, which ensures minimality of fix units. Second, the core algorithm of generating diagnosis is completely replaced by a new algorithm which is much simpler and has better overall performance. A detailed comparison between the two algorithms is also given. Third, in the previous version, the approach is only evaluated on eCos, while in this version, we evaluate the approach on both eCos and Linux. Fourth, all theorems and lemmas are presented with proofs in this version. Fifth, the related work is updated to include the newest progress in this field. Finally, the text has been revised for better clarity and readability.

2 Motivating Example and Solution Overview

Problem. We now motivate our work with an adapted example which utilizes the eCos configurator [2] to configure an object pool. Fig. 1 shows a set of options for configuring an object pool. The left panel shows the set of options that can be changed by the user. The lower-right panel shows the properties of the currently selected option. Particularly, the *flavor* property indicates whether the option is a Boolean option or a data option. A Boolean option can be either selected or unselected; a data option can be assigned an integer or a string value. In Fig. 1, "Pre-Allocation Size" is a data option; "Use Pre-Allocation" is a Boolean option.

Fig. 2. Option "Startup". (The option is selected on the left and its properties are shown on the right. It is disabled because the active-if constraint is not satisfied.)

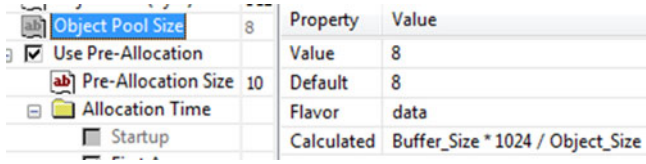


Fig. 3. Option "Object Pool Size". (This option is calculated based on the formula specified under "Calculated" property.)

Under the hood, all the options, the properties, and other information are defined by a configuration model described in a modeling language—CDL [11].

Besides the flavor, each option may also declare constraints using *requires* property or *active-if* property. When a requires constraint is violated, an error is reported in the upper-right panel. In Fig. 1, option "Pre-Allocation Size" declares a requires constraint requiring its value be smaller than or equal to "Object Pool Size", and an error is reported because the constraint is violated. When an active-if constraint is violated, the option is disabled in the GUI and its value is considered as zero. Fig. 2 shows the properties of the "Startup" option. This option declares that at most half of the object pool can be pre-allocated. Since this constraint is violated, the "Startup" option is disabled and the user cannot change its value.

Though behaving differently, both requires and active-if constraints are essentially the same: they both declare constraints on the configuration space, and the user will have to figure out ways to satisfy them during the configuration process—when the user needs to fix a configuration error or change an inactive option.

Satisfying a constraint is often not a straightforward task. To illustrate this, let us take a look at the error on "Pre-Allocation Size" in Fig. 1. In order to fix this error, we need to look up the definition of "Object Pool Size". In Fig. 3, we see that "Object Pool Size" declares a *calculated* property, meaning that the value of the option is determined by the declared expression and cannot be modified by the user. As a result, the constraint declared on "Pre-Allocation Size" is, in fact, the following.

```
Pre_Allocation_Size <=
  Buffer_Size * 1024 / Object_Size
```

Furthermore, according to the CDL semantics, when an option is inactive, the constraints it declares are not considered by the error checking system. An option is inactive when its active-if constraint is violated or its parent option is deselected. "Pre-Allocation Size" has a parent, yielding the following complete constraint:

```
Use_Pre_Allocation -> (Pre_Allocation_Size <=
  Buffer_Size * 1024 / Object_Size)
```

By analyzing the constraint, we realize that we may fix the error by one of the following changes: decreasing "Pre-Allocation Size", or increasing "Buffer Size", or decreasing "Object Size", or, more simply, disabling the pre-allocation function. Now we could choose one of these possibilities and navigate to the respective option to make the change.

This example illustrates the three difficulties in the introduction. The hidden constraints are easily overlooked, and thus it is not easy to figure out all root causes of the error. When mixed with hidden constraints, a simple constraint becomes complex, and it could be difficult to analyze the constraint. Finally, even when the user has figured out a solution, the user has to navigate to the options to make the changes. This becomes difficult when model size grows.

Solution. Our approach automatically generates a list of range fixes to help satisfy a constraint. For the error in Fig. 1, we will generate the following list of alternative fixes.

- `[Use_Pre_Allocation := false]`
- `[Pre_Allocation_Size: Pre_Allocation_Size <= 8]`
- `[Buffer_Size: Buffer_Size >= 5]`
- `[Object_Size: Object_Size <= 409.6]`

Each range fix consists of two parts: the option to be changed and a constraint over the options showing the range of values. The first fix is also a concrete assignment, and will be automatically applied when selected. The other fixes are ranges. If the user selects, for example, the second fix, the configurator will highlight option "Pre-Allocation Size", prompt the range, and ask the user to select a value in the range, as shown in Fig. 4.

Range fixes resolve the three difficulties by automating most of the tasks. The hidden constraints are automatically taken into account and the constraints are automatically analyzed. The navigation is also performed automatically when applying a fix. The user has to only choose a fix and decide a value within the range of the fix.

Until now we have considered only one requires constraint (though potentially augmented with multiple hidden constraints) in the configuration system. In an eCos configuration model, usually there are many requires constraints, each defining a class of errors. These constraints may interact with each other, where fixes for one constraint may violate another constraint, thereby introducing new errors. We shall discuss the problem of one constraint first (called *single-constraint violation*), and discuss problem of multiple interacting constraints (called *multi-constraint violation*) in Section 5.

3 Defining Range Fixes

Basic definitions. Our definitions are built upon the insight [3], [4], [12] that common configuration systems can be

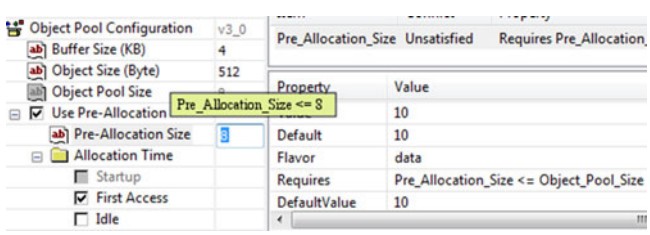


Fig. 4. Executing a range fix. (The user sets the value of "Pre-Allocation Size" with a floating label indicating its range.)

represented by a set of variables (options) and a set of constraints. The constraints are described in a constraint language, in the form of quantifier-free predicate logic. We use Φ to denote a constraint language and $\Phi(V)$ to denote all constraints expressible in the constraint language over the variable set V.

In particular, we consider the situation where a configuration error occurs, i.e., a constraint violation exists. Formally, a *(single-)constraint violation* consists of a tuple (V, e, c), where V is a set of typed variables; the current configuration e is a function assigning a type-correct value to each variable in V; and $c \in \Phi(V)$ is a constraint over V violated by e. A fix generation problem for a violation (V, e, c) is to find a set of range fixes to help users produce a new configuration e' such that c is satisfied, denoted as $e' \models c$. Note that in a configurable system there could also be multiple interrelated constraints, and we consider such situations in Section 5.

Consider the following example of a constraint violation:

$$\begin{array}{lcl} V & : & \{m : \texttt{Bool}, a : \texttt{Int}, b : \texttt{Int}\} \\ e & : & \{m = \texttt{true}, a = 6, b = 5\} \\ c & : & (m \rightarrow a > 10) \wedge (\neg m \rightarrow b > 10) \wedge (a < b). \end{array} \quad (1)$$

The range fixes we have seen so far change only one variable, but more complex fixes are sometimes inevitable. For example, we cannot solve violation (1) by changing only one variable. Several alternative fixes are possible:

- $[m := \texttt{false},\ b : b > 10]$
- $[(a, b) : a > 10 \wedge a < b]$.

The first fix contains two parts separated by ",", each changing a variable. We call each part a *fix unit*. The second fix is more complex. This fix contains only one fix unit, but the range of this fix unit is defined over two variables. When the fix is executed, the user has to choose a value for each variable within the range.

Taking the above forms into consideration, we can define a range fix. A *range fix* r for a violation (V, e, c) is a set of fix units. A fix unit can be either an *assignment unit* or a *range unit*. An assignment unit has the form of "$var := val$" where $var \in V$ is a variable and val is a value conforming to the type of var. A range unit has the form of "$U : cstrt$", where $U \subseteq V$ is a set of variables and $cstrt \in \Phi(U)$ is a satisfiable constraint over U specifying the new ranges of the variables. A technical requirement is that the variables in fix units should be disjoint, otherwise two units may assign two different values to one variable.

For any fix unit $u \in r$, we use $u.V$ to denote the variables to be changed by this unit and use $u.c$ to denote the constraint of the unit. More specifically, for an assignment unit "$var := val$", $u.V$ denotes $\{var\}$ and $u.c$ denotes $var = val$; for a range unit "$U : cstrt$", $u.V$ denotes U and $u.c$ denotes $cstrt$. We further use $r.V$ to denote the union of $u.V$ and use $r.c$ to denote the conjunction of $u.c$, for all $u \in r$. For example, let r be the range fix $[m := \texttt{false},\ b : b > 10]$, then $r.v = \{m, b\}$ and $r.c$ is $m = \texttt{false} \wedge b > 10$.

Applying range fix r of violation (V, e, c) to e will produce a new configuration interactively. We denote all possible configurations that can be produced by applying r to e as $r \rhd e$, where $r \rhd e = \{e' \mid e' \models r.c \wedge \forall_{v \in V}(v \notin r.V \rightarrow e'(v) = e(v))\}$.

Desired properties. A trivial way to generate a fix from a violation is to produce a range unit where the variables to change are all variables in the constraint and the range of these variables is the constraint itself. For example, the fix for violation (1) could be $[(m, a, b) : (m \rightarrow a > 10) \wedge (\neg m \rightarrow b > 10) \wedge (a < b)]$. However, such a fix provides no more information than the original constraint. In this section, we discuss the desired properties of range fixes.

Suppose that r is a range fix for a violation (V, e, c). The first desired property is that a range fix should be correct: all configurations that can be produced from the fix must satisfy the constraint.

Property 1 (Correctness). $\forall e' \in (r \rhd e),\ e' \models c$.

Second, correct range fixes over the same set of variables are often highly overlapping. For example, $[m := \texttt{false},\ b : b > 11]$ is included in $[m := \texttt{false},\ b : b > 10]$, though both are correct. To give users all possible choices, we would like to present the maximal range of the variables.

Property 2 (Maximality of ranges). *There exists no fix r' for (V, e, c) such that r' is correct, $r'.V = r.V$ and $(r \rhd e) \subset (r' \rhd e)$.*

Third, even with the above two properties, the number of possible fixes may still be large. Thus, we further rely on a heuristic rule to reduce the number of fixes: a fix should change a minimal set of variables. The reason is that each value currently assigned to a variable is a configuration decision made by the user, and a fix should not unnecessarily break user decisions. For example, $[m := \texttt{false},\ b : b > 10]$ is preferable to $[m := \texttt{false},\ b : b > 10,\ a : a = 9]$ because the latter unnecessarily changes a, which does not contribute to the satisfaction of the constraints.

Property 3 (Minimality of variables). *There exists no fix r' for (V, e, c) such that r' is correct and $r'.V \subset r.V$.*

As a heuristic rule, minimality of variables sometimes excludes possible changes that resolve a violation. For example, given a constraint $i > j$ and a configuration $\{i = 5,\ j = 10\}$, we can find a possible fix $[i := 6, j := 5]$, but this fix is excluded because we can satisfy the constraint by only changing i or j. As a result, it is important to evaluate whether the excluded changes are really needed by users. In our evaluation described in Section 7, the excluded changes are never adopted by the user.

Another possible way to define the minimality of variables is to consider concrete changes implied by the range fixes, and require the generated fix list to contain exactly all minimal concrete changes. Minimality is defined by considering both the variables and the new values, e.g., $[i := 5, j := 6]$ is not a minimal change when $[i := 5]$ is present, but is a minimal change when only $[i := 11]$ is present. In this way we can ensure that no possible change is excluded. However, we discard this idea because it can lead to complex or confusing fixes. For the above example, the fix list that contains all minimal concrete changes would be as follows.

- $[i : i > 10]$
- $[j : j < 5]$
- $[(i, j) : i <= 10 \wedge j >= 5 \wedge i > j]$.

The last fix is even more complex than the original constraint. Such a fix list would hardly be useful to the user.

Fourth, after deciding the range over the variables, we would like to represent the range in the simplest way possible. One way to simplify the range is to present the range for each variable individually when possible. Thus, another desired property is that a fix unit should change as few variables as possible. In other words, no fix unit can be divided into smaller equivalent fix units.

Property 4 (Minimality of (fix) units). *For any $u \in r$, there does not exist u_1, u_2, where $u_1.V \cap u_2.V = \emptyset$, $u_1.V \cup u_2.V = u.V$, and $u_1.c \wedge u_2.c \equiv u.c$.*

Armed with these properties of range fixes, we can define the desired properties of a list of fixes. Basically, we would expect that a list is complete, contains no duplicated fixes, and all fixes in a list are desired. Since the same constraint can be represented in different ways, we need to consider the semantic equivalence of fixes. Two fixes r and r' for violation (V, e, c) are *semantically equivalent* if $(r \rhd e) = (r' \rhd e)$, otherwise they are *semantically different*.

Property 5 (Well-definedness of fix lists). *Given a constraint violation (V, e, c), a list of fixes L is well-defined iff*

[Minimality] any two fixes in L are semantically different,

[Correctness] each fix in L satisfies Property 1, 2, 3 and 4, and

[Completeness] any fix that satisfies Property 1, 2, 3 and 4 is semantically equivalent to a fix in L.

Thus, a *fix generation problem* is to find a well-defined list of fixes for a given constraint violation (V, e, c).

4 Generating Range Fixes

In Section 3 we claimed five desired properties. To achieve them, our generation algorithm consists of three stages. (i) We find all minimal sets of variables that need to be changed. For example, in violation (1), a minimal set of variables to change is $D = \{m, b\}$. (ii) For each such set of variables, we replace any unchanged variable in c by its current value, obtaining a maximal and correct range of the variables. In the example, we replace a by 6 and get $(m \rightarrow 6 > 10) \wedge (\neg m \rightarrow b > 10) \wedge (6 < b)$. (iii) We simplify the range to get a set of minimal fix units. In the example we will get $[m := \texttt{false},\ b : b > 10]$. Intuitively, stage (i) ensures Property 3 for each fix, as well as the minimality and the completeness of the fix list. Stage (ii) ensures Property 1 and Property 2 for each fix. Stage (iii) ensures Property 4 for each fix.

Next we explain the stages in detail. Stage (ii) is trivial and does not demand further elaboration. We now concentrate on stages (i) and (iii).

4.1 From Constraints and Configuration to Variable Sets

Most constraint solvers have the ability of finding unsatisfiable cores. Our approach builds upon this ability. Given a set of unsatisfiable constraints, an *unsatisfiable core* is a subset of the constraints that is still unsatisfiable. An unsatisfiable core is said to be *minimal* if all its proper subsets are satisfiable. Many constraint solvers further allow the constraints to be split into soft constraints and hard constraints, and seek unsatisfiable cores only within soft constraints.

Our approach first represents a violation as a set of constraints. Given a constraint violation (V, e, c), we treat the configuration e as soft constraints on the variables and constraint c as a hard constraint. Since c is unsatisfied by e, the whole constraint sets must be contradictory and unsatisfiable. For example, violation (1) can be represented by the following constraint set, which is unsatisfiable.

$$
\begin{aligned}
&\text{Hard constraint(c)}: \\
&\quad [0]\ (m \rightarrow a > 10) \wedge (\neg m \rightarrow b > 10) \wedge (a < b) \\
&\text{Soft constraints(e)}: \\
&\quad [1]\ m = \texttt{true} \\
&\quad [2]\ a = 6 \\
&\quad [3]\ b = 5.
\end{aligned}
$$

To find what variables should be changed to resolve the violation, we need to find a subset of soft constraints that, when removed from the constraint set, restores the satisfiability of the whole set. In our example, we could remove subsets $\{1, 3\}$ or $\{2, 3\}$, which correspond to two variable sets $\{m, b\}$ and $\{a, b\}$. Following Reiter [8], we call such a subset a *diagnosis*. A diagnosis is *minimal* iff none of its subsets is a diagnosis. In this stage we need to find all minimal diagnoses.

To get all minimal diagnoses, we rely on unsatisfiable cores. To restore satisfiability, a diagnosis must eliminate all unsatisfiable cores in the constraint set, and thus it must contain at least one constraint from each minimal unsatisfiable core. Furthermore, a minimal diagnosis should not contain extra constraint, so every constraint in a minimal diagnosis must be contained in at least one minimal unsatisfiable core. In our example, the minimal unsatisfiable cores are $\{1, 2\}$, and $\{3\}$. A minimal diagnosis, say $\{1, 3\}$, contains one constraint from each core and no extra constraint.

As a result, we can pick one constraint from each minimal unsatisfiable core to form a diagnosis, and since every constraint in a minimal diagnosis must be contained in a minimal unsatisfiable core, diagnoses constructed in this way contain all minimal diagnoses, plus non-minimal ones. We just need to identify the minimal ones from all diagnoses.

However, modern constraint solvers do not have the interface of returning all minimal unsatisfiable cores. First, they only return one unsatisfiable core at a time, and do not return the full list. Second, though for most of the time the unsatisfiable cores returned are minimal, minimality is not guaranteed.

To solve the second problem, we introduce a minimization step for each unsatisfiable core returned, as shown in Algorithm 1. In this step, we try to remove each constraint from an unsatisfiable core. If we remove a constraint from a core and the core is still unsatisfiable, we proceed with the new core because it is smaller. In the end we will have a core where removing any of its constraints makes it satisfiable. Because the subset of a satisfiable constraint set is still satisfiable, all proper subsets are satisfiable. By definition,

this is a minimal unsatisfiable core. This algorithm will be called iteratively to get the needed unsatisfiable cores.

Algorithm 1. Minimize an unsatisfiable core

Input: C, an unsatisfiable core
1: **for** each constraint c in C **do**
2: [ICS]* check if $C\backslash c$ is satisfiable
3: **if** $C\backslash c$ is unsatisfiable **then**
4: $C \leftarrow C\backslash c$
5: **end if**
6: **end for**
7: **return** C
* "[ICS]" indicates that we need to invoke the constraint solver.

To solve the first problem, we build the diagnoses incrementally. First we invoke Algorithm 1 to get an unsatisfiable core, then we pick one constraint from the core, and remove that constraint from the set. Then we invoke the constraint solver again and retrieve another constraint. We repeat this until there is no unsatisfiable core, and the constraints we have picked form a diagnosis. By iterating all possible combinations from this process, we can ensure to get all minimal diagnoses, but we cannot ensure all diagnoses we get are minimal, and thus we just need to compare the cores against each other and eliminate all non-minimal diagnoses. In our running example, we may first get an unsatisfiable core $\{1, 2\}$, and then we pick 1 and invoke the constraint solver for $\{2, 3\}$. The solver returns $\{3\}$. We pick this constraint and invoke the constraint solver for $\{2\}$. This time the constraint solver returns no core, and we get the diagnosis $\{1, 3\}$.

Algorithm 2. Find minimal diagnoses

Input: C: the (unsatisfiable) constraint set
Data: **E**: a set of partial diagnoses
Data: **R**: a set of completed diagnoses
1: **E** $\leftarrow \{\emptyset\}$ *//Note that $\emptyset$ is an empty diagnosis*
2: **while** **E** $\neq \{\}$ **do**
3: $E_0 \leftarrow$ a random diagnosis in **E**
4: [ICS] find an unsatisfiable core in $(C\backslash E_0)$
5: **if** there is no unsatisfiable core **then**
6: **E** $\leftarrow$ **E**$\backslash E_0$
7: **R** $\leftarrow$ **R** $\cup \{E_0\}$
8: **else**
9: $X \leftarrow$ the unsatisfiable core
10: minimize X using Algorithm 1
11: **for** each E in **E** where $E \cap X = \emptyset$ **do**
12: **for** each x in X **do**
13: $E' \leftarrow E \cup \{x\}$
14: **if** $\nexists E'' \in$ (**E**$\backslash E$) $\cup$ **R**, $E'' \subseteq E'$ **then**
15: **E** $\leftarrow$ **E** $\cup \{E'\}$
16: **end if**
17: **end for**
18: **E** $\leftarrow$ **E**$\backslash E$
19: **end for**
20: **end if**
21: **end while**
22: **return R**

However, invoking the constraint solver is a costly operation and we would like to do it as infrequently as possible. This idea leads to Algorithm 2. In Algorithm 2, we construct all diagnoses together in set **E**. **E** is initialized with a set that contains an empty diagnosis (line 1), which allows to check for unsatisfiable cores in C alone (line 4). Every time a new unsatisfiable core is returned, we extend all partial diagnoses that do not overlap with the new core (lines 11-19). We also drop all diagnoses that are not minimal when extending a diagnosis (lines 14-15).

Lemma 1. *If all invocations to the constraint solver are successful, the set of diagnoses produced by Algorithm 2 contains all minimal diagnoses.*

Proof. First, the algorithm terminates because in each iteration, if no set is moved from **E** to **R**, at least one set in **E** will grow larger, and there is an upper bound of the growth, C.

Second, each set in **R** is a diagnosis because we add E to **R** only when $C\backslash E$ is satisfiable (no unsatisfiable core).

Third, every minimal diagnosis is included in **R**. We show this by proving that at the end of each iteration, every minimal diagnosis is either in **R** or has a subset in **E**. Since **E** is initialized with a non-empty set, there is at least one iteration. Since in the end **E** is empty, we know every minimal diagnosis is included in **R**. Consider the minimal diagnosis D. Initially we have the empty set in **E**, so D has a subset in **E**. Let us assume that at the beginning of an iteration D has a subset E in **E**. In this iteration, there are three possibilities: 1) E is added to **R**, and in this case $E = D$ because D is a minimal diagnosis; 2) E is kept the same, and D still has a subset in E; 3) E is removed from **E** at line 18. In the last case, $D \cap X$ must be non-empty because D is a diagnosis. Let c be an arbitrary constraint in $D \cap X$. As a result, $E \cup \{c\}$ is either added to **E** at line 15 or there is a subset of $E \cup \{c\}$ in **E**$\backslash E \cup$ **R**, which in terms ensure that D is either in **R** or has a subset in **E**.

Fourth, each diagnosis in **R** is minimal. This is because at each iteration, every minimal diagnosis or one of its subsets is contained in **R** or **E**, any non-minimal diagnosis will not be added in **E**. □

It is worth noting that the correctness of Algorithm 2 does not depend on whether the unsatisfiable core, X, is minimal. In other words, Algorithm 2 works correctly even if we remove line 10 from the algorithm. However, based on our test, though line 10 incurs a slight runtime overhead, it could significantly boost the performance when the constraint solver does not return a minimal unsatisfiable core, as a non-minimal unsatisfiable core may noticeably increase the search space afterward. As a result, we keep line 10 in our implementation and evaluation.

In our previous work, we reused the Greiner's algorithm [9] developed for the theory of diagnosis [8] to find the diagnoses. In the evaluation section we shall see that our new algorithm provides overall better performance than Greiner's algorithm.

4.2 From Variable Sets to Fixes

Equipped with diagnoses, we can replace the variables not in these diagnoses with their configuration values in c

(stage (ii)). We denote this modified constraint as c_r. The purpose of stage (iii) is to convert c_r into a fix r.

We perform this conversion in three steps. In step 1, we heuristically convert c_r into a conjunction of smaller constraints, called *part clauses*, where two different part clauses have disjoint sets of variables. For instance, $(m \rightarrow 6 > 10) \wedge (\neg m \rightarrow b > 10) \wedge (6 < b)$, the c_r from our running example, is converted into two clauses $\{\neg m, b > 10\}$. In step 2, to meet Property 4, we also need to ensure that the part clauses are minimal, i.e., it cannot be further divided into smaller part clauses. We employ a novel algorithm that uses a constraint solver to check whether the part clauses are minimal, and divide them into smaller ones if they are not. In step 3, we convert each part clause into a fix unit. In our example we shall get $[m := \texttt{false},\ b : b > 10]$.

In step 1, we employ a set of heuristic rules to convert a constraint into part clauses. First, if the constraints contain any operators convertible to propositional operators, we convert them into propositional operators. For example, eCos constraints contain the conditional operator "?:" such as $(m\,?\,a : b) > 10$. We convert it into propositional operators: $(\neg m \vee a > 10) \wedge (m \vee b > 10)$.

Second, we convert the constraint into CNF. In our example, with diagnosis $\{m, b\}$, we have $(m \rightarrow 6 > 10) \wedge (\neg m \rightarrow b > 10) \wedge (6 < b)$, which gives three clauses in CNF: $\{\neg m \vee 6 > 10,\ m \vee b > 10,\ 6 < b\}$.

Third, we apply the following rules to simplify the CNF repetitively until we reach a fixed point.

Rule 1. Apply constant folding to all clauses.

Rule 2. Apply unit propagation to all clauses, i.e., if a clause contains only one literal, delete the negation of this literal from all other clauses.

Rule 3. If clause C_1 contains all literals in C_2, delete C_1.

Rule 4. If a clause has the form $v = c$ where v is a variable and c is a constant, replace all occurrences of v with c.

Rule 5. Two clauses are conjoined into one if they share variables.

Rule 6. Eliminate operators that are obviously eliminable by algebraic laws, such as replacing $a + 0$ with a.

Rule 7. If a part clause contains only linear equations or inequalities with one variable, we solve the equations and inequality as a system.

In our example, applying Rule 1 to the above CNF, we get $\{\neg m,\ m \vee b > 10,\ 6 < b\}$. Applying Rule 2 to the above CNF, we get $\{\neg m,\ b > 10,\ 6 < b\}$. Applying Rule 5, we get $\{\neg m,\ b > 10 \wedge 6 < b\}$. Applying Rule 6, we get $\{\neg m,\ b > 10\}$. No further rule can be applied to this CNF.

In step 2, we employ a novel procedure that uses a constraint solver to check each part clause to see whether it can be further divided into smaller part clauses. This procedure takes as input a constraint c, two disjoint sets of variables V_1 and V_2, where $c \in \Phi(V_1 \cup V_2)$, and tests whether there exist c_1 and c_2, where $c_1 \in \Phi(V_1)$, $c_2 \in \Phi(V_2)$, and $c = c_1 \wedge c_2$. The basic idea of this algorithm is that, when $c = c_1 \wedge c_2$, we can obtain c_1 from c by replacing the variables in V_2 with an assignment that make c_2 evaluate to $true$. To obtain such an assignment of V_2, we can obtain an assignment that satisfies c, because it will also satisfies c_2. The same procedure can be used to obtain c_1. The algorithm is shown in Algorithm 3.

We first invoke the constraint solver to find an assignment of the variables in c that satisfies c. Next we construct two constraints, c_1 and c_2 that only contain variables in V_1 and V_2, respectively. This construction is achieved by replacing the variables that should not appear in the two constraints by their values in e. Finally, we check whether $c_1 \wedge c_2 = c$ hold by checking the unsatisfiability of its negation. If it holds, we know c can be divided into c_1 and c_2, otherwise we return c itself to indicate c cannot be divided this way.

Algorithm 3. Divide a fix unit

Input: c, a satisfiable constraint to be divided
Input: V_1, the variable set of the first sub constraint
Input: V_2, the variable set of the second sub constraint
1: [ICS] $e \leftarrow$ an assignment that satisfies c
2: $c_1 \leftarrow c[V_2 \backslash e]^*$
3: $c_2 \leftarrow c[V_1 \backslash e]$
4: [ICS] check if $c_1 \wedge c_2 \neq c$ is satisfiable
5: **if** unsatisfiable **then**
6: **return** (c_1, c_2)
7: **else**
8: **return** (c)
9: **end if**
* $c[V_1 \backslash e]$ indicates replacing each $v \in V_1$ in c with $e(v)$.

Lemma 2. *If all invocations to the constraint solver terminate with an answer, Algorithm 3 returns (c_1, c_2) iff for the given input V_1, V_2 and c, there exist such c_1 and c_2 that (1) $c = c_1 \wedge c_2$, and (2) $c_1 \in \Phi(V_1) \wedge c_2 \in \Phi(V_2)$.*

Proof. The forward direction is straightforward. we have verified $c_1 \wedge c_2 = c$ by the constraint solver and have constructed c_1 and c_2 by keeping only the variables in V_1 and V_2, respectively.

To prove the backward direction, let us suppose c can be divided into c_a and c_b that meet the above conditions. Let e' be an arbitrary assignment that coincides with e on V_2, that is, $\forall v \in V_2.\ e'(v) = e(v)$. Since e satisfies c, we have $c_b[V_2 \backslash e] = true$, which leads to $c_b[V_2 \backslash e'] = true$. From this we have $c_a[V_1 \backslash e'] = c_a[V_1 \backslash e'] \wedge true = c_a[V_1 \backslash e'] \wedge c_b[V_2 \backslash e'] = (c_a \wedge c_b)[(V_1 \cup V_2) \backslash e'] = c[(V_1 \cup V_2) \backslash e'] = c[V_2 \backslash e][V_1 \backslash e'] = c_1[V_1 \backslash e']$. Because e' effectively covers any assignment to the set of V_1, c_a and c_1 evaluates the same on all values in their domain, and thus $c_a = c_1$. Similarly, we have $c_b = c_2$. As a result, $c_1 \wedge c_2 = c$, and the procedure will return (c_1, c_2) which is equal to (c_a, c_b). □

With this procedure, we divide each part clause into smaller units by checking all combinations of V_1 and V_2, e.g., for a part clause over $\{a, b, c\}$, we check whether it can be divided into two constraints over $(\{a, b\}, \{c\})$, $(\{b, c\}, \{a\})$, $(\{a, c\}, \{b\})$, respectively. We repeat this process until no part clause can be further divided. This is an inherently inefficient process because for each unit constraint containing n variables we need to test $\Sigma_{i=1}^{\lfloor n/2 \rfloor} C_n^i$ combinations, which grows exponentially. However, since we have applied heuristic rules to minimize the part clause in step 1, we seldom need to deal with a large n if the heuristic rules are effective. In our evaluation, 98.8 percent of part clauses after step 1 contains only one variable, which are already minimal. The rest of clauses

usually contain only two variables, and a very small number contains three variables.

In step 3, we convert each clause into a fix unit. If the clause has the form of v, $\neg v$, or $v = c$, we convert it into an assignment unit, otherwise we convert it into a range unit. In the example, we convert $\neg m$ into an assignment unit and $b > 10$ into a range unit and get $[m := \texttt{false},\ b:\ b > 10]$.

4.3 Correctness of the Generation Algorithm

Theorem 1. *Any fix list produced by the generation algorithm satisfies Property 5.*

Proof. With the lemmas, this theorem is easy to prove. Here we give a sketch of the proof.

(Correctness). Property 1: let c' be the constraint produced by stage (ii); since we only perform equivalent transformation in stage (iii), we have that $r.c = c'$ and r only changes the variables in c', and thus any configuration producible by the fix will satisfy the original constraint c. Property 2: assuming that there exists a correct r' that could produce a configuration e' that is not producible by r, we have a contradiction where $e' \models \neg c$ because $r.c = c'$, and thus r' does not exist. Property 3: directly from Lemma 1. Property 4: because we tried to divide each fix unit in all possible ways by Algorithm 3, we ensure that each fix unit is minimal.

(Minimality). As every diagnosis produced by stage (i) is minimal, each variable is needed to be changed to satisfy the constraint. As a result, two fixes produced from two different diagnoses are semantically different.

(Completeness). Let us assume there exists a fix r'' that satisfies Property 1, 2, 3, and 4 but not in the generated list. Because of Property 2 and Property 3, $r''.V$ must be different from any fix in the list, which contradicts Lemma 1. □

5 Constraint Interaction

So far we have only considered fixes for one constraint. However, the constraints in configuration systems are often interrelated; satisfying one constraint might violate another. As a result, we have to consider *multi-constraint* violation rather than single-constraint violation. A multi-constraint violation is a tuple (V, e, c, C), where V and e are unchanged, c is the currently violated constraint, and C is the set of constraints defined in the model and satisfied by e. The following example shows how a fix satisfying c can conflict with other constraints in C that were previously satisfied.

$$
\begin{aligned}
&V: \{m : \texttt{Bool}, n : \texttt{Bool}, x : \texttt{Bool}, y : \texttt{Bool}, z : \texttt{Bool}\}\\
&e:\ \{m \mapsto \texttt{true}, n \mapsto \texttt{false}, x \mapsto \texttt{false},\\
&\qquad y \mapsto \texttt{false}, z \mapsto \texttt{false}\}\\
&c:\ m \wedge n \qquad (2)\\
&C: \{c_2, c_3\} \text{ where}\\
&\qquad c_2 \text{ is } n \rightarrow (x \vee y)\\
&\qquad c_3 \text{ is } x \rightarrow z.
\end{aligned}
$$

If we generate a fix from (V, e, c), we obtain $r = [n := \texttt{true}]$. However, applying this fix will violate c_2.

Note that a multi-constraint violation involves only one violated constraint. If, for example, an eCos configuration contains multiple errors, we treat each of them as a multi-constraint violation and fix them one by one.

Existing work has proposed three different strategies to deal with this problem; each has its own advantages and disadvantages. We now revisit these three strategies, and show that they can all be used with range fix generation by converting a multi-constraint violation into a single-constraint one. In the evaluation section we will give a comparison of the three strategies.

Ignorance. All constraints in C are simply ignored, and only fixes for (V, e, c) are generated. This strategy is used in fix generation approaches considering only one constraint [13]. This strategy does not solve the constraint interaction problem at all. However, it has its merits: first, the fixes are only related to the violated constraint, which makes it easier for the user to comprehend the relation between the fixes and the constraints; second, this strategy does not suffer from the problems of incomplete fix list and large fix list, which exist in the other two strategies and will be discussed later; third, this strategy requires the least computation effort and is the easiest to implement.

Elimination. When a fix contains changes that violate other satisfied constraints, these changes are excluded from the range of the fix, i.e., any changes with side effect are "eliminated". In the example in violation (2), fix r contains only one change and this change violates c_2. Thus, fix r is eliminated. This strategy is proposed by Egyed et al. [14] and used in their UML fix generation tool.

To apply this strategy to range fix generation, we first find a subset of C that shares variables with c, then replace the variables not in c with their current values in e, and conjoin this subset of constraints with c. For example, to apply the elimination strategy to violation (2), we first find the constraints sharing variables with c, which includes only c_2, and then replace x and y in c_2 with their current values, getting $c_2' = n \rightarrow \texttt{false} \vee \texttt{false}$. Then we generate fixes for $(V, e, c \wedge c_2')$.

Although the elimination strategy prevents the violation of new constraints, it has two noticeable drawbacks. First, it excludes many potentially useful solutions. Bringing new errors is often inevitable. Simply excluding the changes will only provide less help. In our example, we will get an empty fix set, which does not provide any solution to the current error. Second, since we need to deal with the conjunction of several constraints, the resulting constraint is much more complex than the original one. Our evaluation showed that some conjunctions can contain more than ten constraints. Nevertheless, compared to the propagation strategy, this increase in complexity is still small.

Propagation. When a fix violates other constraints, we modify further variables in the violated constraints to keep these constraints satisfied. In other words, the fix is "propagated" through other constraints. For example, fix r will violate c_2, so we also modify variables x or y to satisfy c_2. Then the modification of x will violate c_3, and we also modify z. In the end, we get two fixes $[n := \texttt{true}, x := \texttt{true}, z := \texttt{true}]$ and $[n := \texttt{true}, y := \texttt{true}]$. This approach is used in the eCos configuration tool [2] and the feature model diagnosis approach proposed by White et al. [6].

TABLE 1
Comparison of the Three Strategies

	Ignorance	Elimination	Propagation
Complexity of Fix Lists	Simple	Simplest	*Possibly complex*
Execution Time	Shortest	Short	*Possibly long*
Fix Completeness	Complete (for one constraint)	*Incomplete*	Complete (for all constraints)
Introducing new errors	*Possible*	Never	Never

To apply this strategy, we first perform a static slicing on C to get a set of constraints directly or indirectly related to c. More concretely, we start from a set D containing only c. If a constraint c' shares any variable with any constraint in D, we add c' to D. We keep adding constraints until we reach a fixed point. Then we conjoin all constraints in D, and generate fixes for the conjunction. For example, if we want to apply the propagation strategy to violation (2), we start with $D = \{c\}$; then we add c_2 because it shares n with c; next we add c_3 because it shares x with c_2. Now we reach a fixed point. Finally, we generate fixes for $(V, e, c \wedge c_2 \wedge c_3)$.

The propagation strategy ensures that no satisfied constraint is violated and no fix is eliminated. However, there are two new problems. First, the performance cost is the highest among the three strategies. The constraints in real-world models are highly interrelated. In large models, the strategy often led to conjunctions of hundreds of constraints. The complexity of analyzing such a large conjunction is significantly higher than analyzing a single constraint. Second, since many constraints are considered together, this strategy potentially leads to large fixes (i.e., fixes that modify a large set of variables), and large number of fixes, which are not easy to comprehend and apply.

We summarize the differences between the three strategies in Table 1. In the table, we use *italic* font to indicate a potential problem in a strategy.

6 Implementation

We have implemented our approach on eCos CDL and Linux Kconfig as two command line tools, ECC Fixer[2] and Kconfig Fixer, respectively. Our implementation contains two frontends and one backend. The frontends interact with the user, and convert eCos and Kconfig models and configurations into violations where the constraints in the violations are solvable by the constraint solver. The backend implements our approach to generate fixes for the violations. We use Microsoft Z3 SMT solver [15] in our backend, and it supports the standard input format of SMT solvers: SMT-LIB [16]. In the rest of the section we describe how the two frontends are implemented.

6.1 eCos Implementation

As mentioned before, the eCos system has two forms of violations to be solved: requires violation and active-if violation. Our implementation caters for both types of violations. The user can point an error or an inactive option, and our tool presents a list of fixes for the user.

To convert CDL models into SMT-LIB, we carefully studied the formal semantics of CDL [4], [11] obtained through reverse engineering from its configurators and documents. There are mainly two challenges in the conversion. First, CDL is an untyped language, while SMT-LIB is a typed language. To convert CDL, we implement a type inference algorithm to infer the types of the options based on their uses. When a unique type cannot be inferred or type conflicts occur, we manually decide the feature types. Please note that this typing process is required once for a CDL model, and is never needed during a configuration process.

The second challenge is dealing with string constraints. The satisfiability problem of string constraints is undecidable in general [17], and general SMT solvers do not support string constraints [15]. Yet, string constraints are heavily used in CDL models. Nevertheless, our previous study on CDL constraints [7] shows that all string constraints used in these models implement a set semantics: a string is considered as a set of substrings separated by spaces, and string functions are actually set operations. In particular, `is_substr` is always used as a set member test to check if a particular word is contained in a list of words, and concatenation is only applied to literals. Based on this observation, we encode each string as a bit vector, where each bit indicates whether a particular substring is presented or not. Since in fix generation we will never need to introduce new substrings, the size of the bit vector is always finite and can be determined by collecting all substrings in the model and the current configuration.

6.2 Linux Implementation

Different from eCos, Linux Kconfig does not have requires constraints. Instead, Kconfig equips each option with a function that computes a range for the current option based on the values of other options, and the user value is enforced in this range. Furthermore, Kconfig also has the concept of active/inactive options. As a result, Linux configuration is organized in a way which never introduces errors, but there are constraint violations when the user wants to modify a value outside the range or changes an inactive option. Our implementation generate fixes for both cases.

Similar to the eCos implementation, we also carefully studied the formal semantics of the Kconfig language through reverse engineering [1]. A key issue to convert a Kconfig model to SMT-LIB is to convert the tristate options used in Kconfig. A tristate option has a value of 'n', 'm', or 'y', which respectively means "disabled", "modularized" and "activated". Internally, Kconfig converts 'n', 'm', or 'y' to integers (0, 1, 2) and applies different numeric operators on them. As a result, we could also convert these options into integers in SMT-LIB, but this requires extra constraints for each tristate option to limit its value in the interval [0, 2]. Our method to solve this problem is to define an enumeration type which has three values, and define the operators to simulate the needed numeric computations on the enumeration. This method makes expressions more complex, but leads to a significant reduction of constraints.

In Kconfig, tristate options are often used in mix with Boolean option, where a Boolean option is consider to

2. Available at: gsd.uwaterloo.ca/eccfixer

TABLE 2
eCos Configuration Files

Architecture	Project	Options	Constraints	Changes
virtex4	ReconOS	933	330	49
xilinx	ReconOS	765	272	53
ea2468	redboot4lpc	658	96	14
aki3068net	Talktic	817	195	3
gps4020	PSAS	535	85	23
arcom-viper	libcyt	771	189	26

TABLE 3
Linux Configuration Files

Project	Kernel Version	Options	Constraints	Changes
astlinux	2.6.17	3,544	5,112	192
CdMa-HeRoC	2.6.29	5,565	6,887	5
crux-arm	2.6.19	3,885	4,771	7
padxroom	2.6.32	6,188	7,734	170

have the value 'n' or 'y'. When tristate options are converted into enumerations, we need to deal with the mixed calculation of Booleans and tristates. There are two methods to solve this problem. One method is to convert Boolean options into tristate type and add constraints to disallow value 'm'. Another method is to add a conversion function for Booleans when they are used with tristates. The first method will lead to redundancy of constraints while the second method makes the conversion and calculation a bit more complex. We have tested both methods and decide to use the second method because it requires less execution time.

7 Evaluation

7.1 Research Questions

To really know whether the approach works in practice, several research questions need to be answered by empirical evaluation:

- *RQ1*. How efficient is our algorithm?
- *RQ2*. How complex are the generated fix lists?
- *RQ3*. How often are the actual final user changes suggested by our approach? Does our approach suggest more such changes than existing approaches?
- *RQ4*. What are the differences among the three strategies for multi-constraint violations, namely, ignorance, elimination, and propagation?
- *RQ5*. How efficient does the new diagnosis algorithm (Algorithm 2) compare with Greiner et al.'s algorithm [9] used in our previous work [10]?
- *RQ6*. How often is the fix unit division procedure (Algorithm 3) needed? And how much extra time does it cost?

RQ1, RQ2 and RQ3 examine the general usability of the proposed approach. RQ3 also checks a potential problem of Property 3, which excludes theoretically possible fixes. When the user changes are always covered by the generated fixes, the excluded fixes are probably not useful and can be excluded. RQ4 compares the strategies for multi-constraint violations. RQ5 and RQ6 evaluate the individual usefulness of the two newly introduced components in our fix generation approach.

7.2 Data Set and Methodology

To answer the research questions, we need a set of constraint violations. We obtained the data set by re-enacting the real-world configuration process from the configuration histories at open source repositories. As shown in Tables 2 and 3, we used six eCos configuration files from five eCos-based open projects and four Linux configuration files from four open source projects based on Linux kernel code. Each eCos file targets a different hardware architecture (the first column in Table 2), which leads to a different set of options and constraints. Each Linux configuration file uses a different kernel version (the second column in Table 3), which also leads to a different set of options and constraints.

We re-enacted the real-world configuration process by replaying the user changes extracted from the version histories of the configuration files. We assume that the user starts from the default configuration provided by the system for a given architecture, and added it as the earliest version in the corresponding version history. Then we compared each pair of consecutive versions to find changes to the options. Next we replayed these changes to simulate the real configuration process: we apply the changes one by one on the earlier version of the configuration until we reach the later version, at which point we have applied all changes. Since we did not know the order of changes within a revision, we used three orders: a top-down exploration of the configuration file, a bottom-up one, and a random one. The rationale for the first two orders is that expert users sometimes edit the textual configuration file directly rather than using the graphical configurator. In this case, they will read the options in the order that they appear in the file, or the inverse if they scroll from bottom to top. We use the entire version history in our experiment.

During the replay of the changes, violations may occur, and we collected these violations for our experiments. In the eCos case, we collected two kinds of violations. (i) *Errors*: when a requires constraint in eCos is violated. (ii) *Activation violations*: when an option in eCos should be changed, but is currently inactive. In the Linux case, we collected *range violations*: when a Kconfig option needs to be changed to a value outside its range. Any attempt to change an inactive Kconfig option will also be considered as a range violation. This is because even if we activate this option, the expected value may still be outside its range. To unify the UI concepts, we generated the fix once for both the activation and the value change. The activation violations and the range violations will prevent the replay of the current change. After recording the violation, we defer this change until we reach a point that applying this change causes no violation. Because in the final configuration there is no violation, such a point must exist. Note that the differences among the three types of violations lie only in the user interface level. Conceptually they are all violations consisting of constraints, variables, and assignments.

Interestingly, the default configurations of eCos models already contains errors. We also added these errors

TABLE 4
eCos Constraint Violations

Architecture	Errors in defaults	Errors in changes	Activating
virtex4	56	5	15
xilinx	48	1	2
ea2468	8	8	1
aki3068net	26	3	0
gps4020	12	10	4
arcom-viper	26	0	0

into our data sets for generating fixes. We assume the user will fix these errors before starting the configuration process. However, since we do not know how the user would respond to these fixes, we just pick the fix proposed by the eCos configurator (which proposes one concrete fix for one error), and used the resolved configurations as the first version in the above replay process. On the other hand, most stored revisions of the configuration files contain no error. We found only one error in one stored revision, which is never fixed as the revision is the latest one in the version history.

Tables 4 and 5 show the number of violations we collected using the above process. Because different eCos architectures may use shared packages, there are duplicated violations in the eCos case. After removing the duplications, we have a total of 117 multi-constraint violations (including 68 errors from the defaults, 27 errors from the user changes and 22 activation violations) from the eCos configuration files and 193 single-constraint violations from the Linux configuration files. There are only single-constraint violations in the Linux configuration files because, as stated before, there is no user-visible notion of "constraints" in Kconfig. The Kconfig configurator will always consider all constraints together, and the individual constraints we derived from Kconfig semantics are not known to the user. As a result, it is not sensible to fix for one constraint, and we always have to consider the conjunction of all constraints, which results in a single-constraint violation. Note that this design effectively enforces the propagation strategy in Linux.

After we obtained the data set, we generated range fixes for all the violations. For multi-constraint violations, we generated fixes using all the three strategies to answer RQ4. To answer RQ1, we invoke our algorithm 100 times and calculate the average time. We also kill the fix generation process if it takes more than 20 seconds. We chose 20 seconds because this is known to be the maximum time a user can wait for an interactive tool under a wait cursor [18]. To answer RQ3, we also invoked the built-in fix generator of the eCos configurator on the 27 errors from user changes.

TABLE 5
Linux Constraint Violations

Project	Range Violations
astlinux	83
CdMa-HeRoC	2
crux-arm	2
padxroom	106

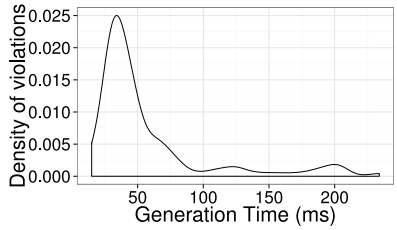


Fig. 5. Fix generation time (eCos).

The errors from the defaults were not considered because we do not have the user changes to compare with. The other two types of violations were not compared because they are not supported by the eCos configurator. To answer RQ5, we also generate fixes using Greiner et al.'s algorithm. To answer RQ6, we also generate fixes with the step of Algorithm 3 removed.

The experiments were executed on computer with Intel Core i3-2120 CPU and 6 GB memory.

7.3 Results

We shall present the results on eCos data using only the propagation strategy except for RQ4. This is because 1) the fix generation on the Linux data is effectively equivalent to the propagation strategy, and 2) as will be shown later, propagation strategy is relatively the best strategy among the three.

RQ1. RQ1 focuses on the efficiency of the algorithm. To answer this research question, we record the time used to generate fixes for each violation, and kill the fix generation process if it takes more than 20 seconds. The fix generator returns within 20 seconds on all eCos violations and 126 Linux violations, in total 80 percent of all violations. The concrete generation time is presented as density graphs in Figs. 5 and 6. The maximum generation time for an eCos violation is 234 ms, completely within the acceptance range of interactive tools. However, the Linux violations have much longer generation time. There are 67 violations for which our algorithm does not generate a fix within 20 seconds, constituting 20 percent of all violations and 34 percent of Linux violations. This is because the Linux models are much larger, and require much more time to process. As a matter of fact, the 67 violations all belong to the padxroom model, which is the largest model in our data set. However, even for this largest model (cf. Table 3), our approach generates fixes for 22 violations within one second. On the other hand, our approach works quite well on middle-size models. For example, our approach generates fixes within one second for 96 percent violations of the astlinux model, which is a fairly large model.

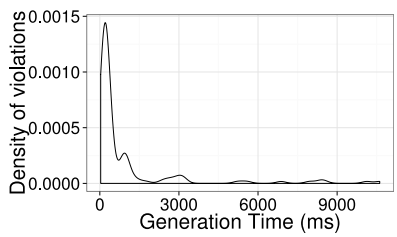


Fig. 6. Fix generation time (Linux).

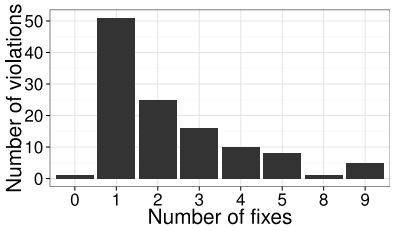


Fig. 7. The number of fixes per violation (eCos).

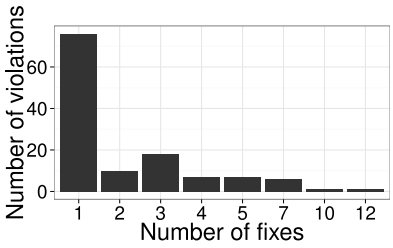


Fig. 8. The number of fixes per violation (Linux).

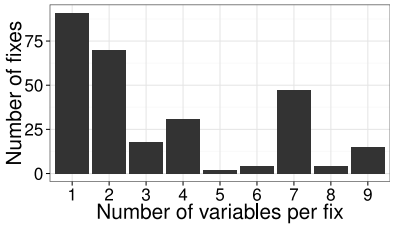


Fig. 9. The number of variables per fix (eCos).

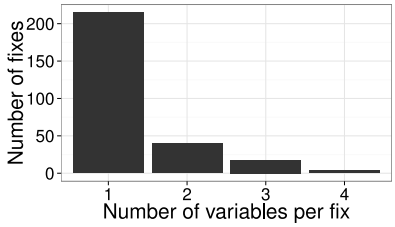


Fig. 10. The number of variables per fix (Linux).

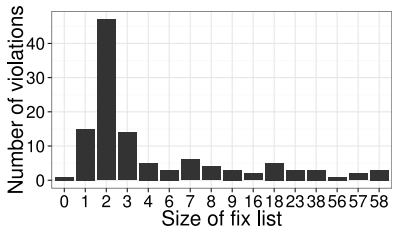


Fig. 11. The sizes of fix lists (eCos).

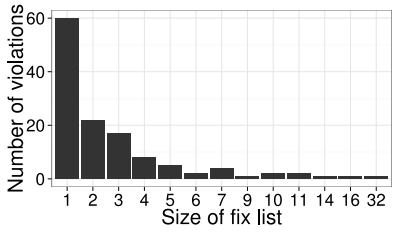


Fig. 12. The sizes of fix lists (Linux).

To understand where the bottleneck lies, we further investigate the executions of our algorithm for five timed-out violations. The investigation showed that all executions were still within stage (i) at the end of 20 seconds. This indicates that at least we need a more efficient method for generating diagnoses.

RQ2. RQ2 focuses on the complexity of the generated fixes. To answer RQ2, we first calculated two basic measures over the 243 violations that have fixes generated: the distribution of the number of fixes per violation (see Figs. 7 and 8) and the distribution of the number of variables changed by each fix (see Figs. 9 and 10). From these figures we see that most fix lists were short and most fixes changed a small number of variables. More concretely, 94 percent of the fix lists contain at most five fixes and 87 percent of the fixes change less than five variables. There was also an activation violation that did not produce any fix. A deeper investigation of this violation revealed that the option is not supported by the current hardware architecture, and cannot be activated without introducing new configuration errors. The extracted changes actually lead to an unsolved configuration error in the subsequent version.

It is still unclear how the combination of fix number and fix size affect the size of a fix list, and how the large fixes and long lists are distributed in the violations. To understand this, we measured the size of a fix list. The size of a fix list is defined as the sum of the number of variables in each fix. The result is shown in Figs. 11 and 12. From the figure we can see that the propagation strategy did lead to large fix lists. The largest involves 58 variables, which is not easily readable. However, the long lists and large fixes tend to appear only on a relatively few number of violations, and the majority of the fix lists are still small: 90 percent of the violations have fix lists with no more than 10 variables.

We also measured the number of variables per unit. The vast majority of the fix units contain only one variable. Among all the fixes generated, there are only seven fix units each with more than one variable, six with two variables each and one with three variables. This shows that fix units are mostly simple, and complex units such as the one in the second fix of our running example are rare.

Overall, the fixes for the Linux models are simpler. This is probably because Linux models have simpler constraints, which do not involve complex arithmetic or string operations [3].

RQ3. RQ3 focuses on the coverage of the generated fixes over user changes. To answer RQ3, we evaluated how often the final user changes are covered by our fixes. Given an error or activation violation, we examined the change history to identify a subsequent configuration that corrected the problem, and then we checked if the values in the corrected configuration fell within one of the ranges proposed by our generated fixes.

For the Linux violations, all violations have subsequent user changes, and the generated fixes cover all of them (100 percent).

For the eCos violations, out of all 49 violations from user changes, a total of 47 had corrections in subsequent revisions. The fixes generated by our tool covered 46 of these violations (98 percent). An investigation into the remaining violation showed that the erroneous option discussed in RQ2 was responsible for that discrepancy. Since the

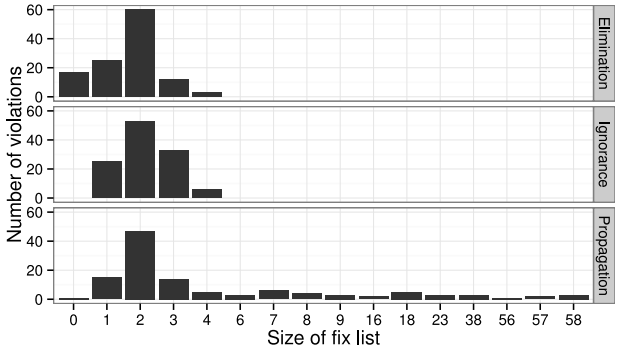


Fig. 13. The sizes of fix lists in the three strategies.

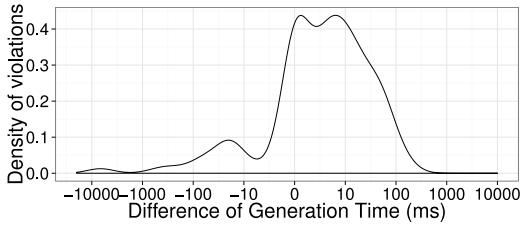


Fig. 14. Comparison of Algorithm 2 and Griner et al.'s algorithm.

propagation strategy ensures to introduce no new error, the re-enacted user change was not proposed as a fix.

These results suggest that Property 3 is a good heuristic in practice that does not exclude useful changes needed by the user.

For comparison with the fix generator built into the eCos configurator, we consider the 26 out of 27 errors from user changes. The last one is not considered because it has no subsequent correction. The eCos configurator was able to handle 19 of the 26 errors, giving a coverage of 73 percent. Comparatively, our tool covered all 26 errors.

RQ4. RQ4 focuses on the differences of the three strategies for multi-constraint violations. To answer this research question, we examine the lines in Table 1 one by one. The first line suggests that the propagation strategy potentially produces large fix lists. At this stage, we would like to know if the other two strategies actually produce simpler fixes. We compared the size of fix lists generated by the three strategies in Fig. 13. The elimination and ignorance strategies completely avoided large fix lists, with the largest fix list containing four variables in total. The elimination strategy changed even fewer variables because some of the larger fixes were eliminated.

We also compared the generation time of the three strategies. For all violations, the average generation time for the propagation strategy was 57 ms, while the elimination strategy was 29 ms and the ignorance strategy was 28 ms. Since the overall generation time is small, it does not make a big difference in tooling, at least for the eCos violations.

Next, we want to understand to what extent the elimination strategy affects completeness. In all 117 violations, the elimination strategy generated no fix for 17 violations. This is significantly more than ignorance and propagation, which had zero and one violation with no fix, respectively. We also measured the coverage of user changes using the elimination strategy. In the 47 violations, only 27 were covered, giving a coverage of 57 percent. This is even lower than the eCos configurator, which generates only one fix, showing that a lot of useful fixes were eliminated by this strategy.

Finally, we want to understand how often the ignorance strategy brings new errors. We compared the fix list of the ignorance strategy with the fix list of the elimination strategy. If a fix does not appear in the list of elimination strategy, it must bring new errors. As a result, 32 percent of the fixes generated by the ignorance strategy brought new errors, and those fixes were generated from 44 percent of the constraint violations. This shows that the constraints in practice are usually inter-related and the ignorance strategy potentially causes new errors in many cases.

RQ5. RQ5 focuses on the efficiency of Algorithm 2. To measure its efficiency, we calculate the difference between our algorithm (NEW) and Greiner et al.'s algorithm (GSW), i.e., difference = [generation time using NEW] − [generation time using GSW]. The result is presented as a density graph in Fig. 14. The average generation times between the two algorithms are quite close, with 539 ms using NEW and 588 ms using GSW. However, while in the majority of the cases NEW is less than 100 ms slower than GSW, GSW is significantly slower than NEW in the rest of the cases. In the worst case, GSW uses 6,752 ms more time than NEW, and is four times as slow as NEW. On the other hand, in the best case where GSW outperforms NEW, GSW is 170 ms faster than NEW, which is only 15 percent faster. We argue that in interactive tools NEW will give overall better performance because a time difference of a few hundred milliseconds is hardly noticeable, and a time difference of a few seconds is quite obvious and annoying. Further considering NEW is much simpler than GSW (GSW builds a directed acyclic graph and employs a set of non-trivial operations such as node reusing, closing, and pruning), we believe NEW is a better diagnosis algorithm for range fix generation.

RQ6. RQ6 focuses on the necessity and efficiency of Algorithm 3. While fixing the violations, the heuristic rules in step 1, stage (iii) produce 1,287 part clauses, where 16 unit clauses contain more than one variables and need to be checked and furthered divided with Algorithm 3. These clauses contain two or three variables (in average 2.06 variables). By applying Algorithm 3, nine clauses are split into smaller units, resulting in an increase of nine clauses and a new average of 1.32 variables in the 16 part clauses. Furthermore, the execution time of Algorithm 3 is very short: the average execution time per part clause is only 15 ms . We believe the short execution time is mainly contributed by the heuristic rules. After applying the heuristic rules, each part clause is already very simple, and is easy for the constraint solver to analyze. Also, Z3 supports incremental constraint solving, which further reduces the execution time. This result shows that Algorithm 3 does play a useful role in the fix generation, and its added runtime cost is negligible.

8 Threats to Validity

The main threat to external validity is that our evaluation is a simulation rather than an actual configuration process. We addressed this threat by using real-world configuration history rather than random simulation. We also replayed the changes in different orders, and used different configuration files from different systems and different open source

projects, in the hope that our simulation would have a high chance to cover the real world configuration processes.

Another threat is that our results on Linux and eCos configurations may not be generalizable to other configuration systems. However, as shown by Berger et al. [19], Linux and eCos are the most complex ones among different kinds of configuration systems. Linux is the largest among several systems that use Kconfig, while eCos has the most complex constraints, especially the non-Boolean constraints. As a result, our approach would probably have a better performance on other systems as they are less complex than those used in our evaluation.

A threat to internal validity is that our translation from the configuration models into logic constraints could be incorrect. To address this threat, we have carefully studied the formal semantics of the configuration languages [1], [4], [11]. In particular, we have developed two formal semantics for the CDL language, one from the variability modeling perspective, and one from the configurator perspective. We have carefully inspected and compared both against each other and tested them on examples with respect to the eCos configurator.

9 Limitations and Future Work

Undocumented constraints. Our approach requires the constraints and options to be explicitly specified in the configuration system. Yet in practice many constraints are implicitly enforced by the system implementation but not explicitly specified in the configuration system. We call such unspecified, but actually existing constraints as *undocumented constraints*. Please note the difference between undocumented constraints and hidden constraints mentioned in Section 1, where hidden constraints are specified but not explicitly presented to the user. Undocumented constraints are one of the major reasons for configuration failures [20]. Fortunately, some recently proposed approaches [21], [22] enable the inference of these undocumented constraints from the source code. These approaches can be possibly combined with our approach to enable the analysis of undocumented constraints.

User constraints. Besides the constraints imposed by the system, for each configuration, there may be user constraints relating to the goal of the particular configuration. For example, in the example of object pool configuration, a particular user constraint could be to accommodate at least 500 objects. User constraints can be incorporated into our approach as long as the user constraints can be expressed as quantifier-free predicate logic. However, there are cases where user constraints cannot be easily converted into predicate logic. First, user constraints often come with different priorities. For example, a user configuring a Linux kernel may have the goal "the system consumes less than half of the physical memory, on condition that a network connection is operational." In this case, the constraint on memory is secondary to the constraint on network. Second, user constraints may come in the form of optimization problems. For example, a user constraint may be "the system consumes as little memory as possible". To support these cases, configuration approaches based on constraint hierarchies [23] and multi-objective optimization [24] may be combined with our approach. This is one of the future directions that could be explored to enhance this work.

Complex configuration systems. Our approach can be applied to a configuration system if (1) there are a fixed number of variables in the system, (2) there are a fixed number of constraints in the system, and (3) the constraints can be presented in quantifier-free predicate logic, and can be reasoned in a constraint solver. The two configuration systems used in our experiments, also reported as the most complex ones among different kinds of configuration systems [19], fall into this category. However, we also realize that there are experimental configuration systems that do not fall into this category. For example, in the Clafer language [25], an experimental modeling language, the options (known as clafers) can be dynamically added or removed, and constraints are specified in first-order logic. A possible way to model such a system into our approach is to set an upper bound on the options to be inserted in one fix, and pre-allocate variables for future possible insertions. This is a future direction to be explored.

User study. Our evaluation quantitatively measures several aspects of the proposed approach, including the efficiency of the algorithm, the complexity of the generated fixes, and the coverage of user fixes. Although these quantifications measure several aspects relating to the usability of our approach, it cannot cover all aspects. In particular, although a fix list containing fewer variables is probably easier to read, it is not necessarily true in all cases: the constraints characterizing the range of variables may be difficult to interpret, or the fix list may contain meaningless fixes according to undocumented domain knowledge. To fully understand the usability of the proposed approach, more studies involving real users need to be performed. This is one of the future directions.

10 Related Work

The idea of automatic fix generation is not new. Different approaches have been proposed to fix different types of software artifacts. Two typical types of artifacts are software models and programs. The approaches on software models are close to ours. Nentwich et al. [13] propose an approach that generates abstract fixes from first-order logic rules. Their fixes are abstract because they only specify the variables to change and trust the user to choose a correct value. In contrast, our approach also gives the range of values for a variable. Furthermore, their approach only supports "$=$" and "$\neq$" as predicates and, thereby, cannot handle models like eCos. Scheffczyk et al. [26] enhance Nentwich et al.'s approach by generating concrete fixes. However, this approach requires manually writing fix generation procedures for each use of a predicate in every constraint. As configuration systems often contain hundreds or thousands of constraints, manually writing generation procedures is not feasible. Egyed et al. [14] propose to write such procedures for each type of variable rather than each constraint to reduce the amount of code written and apply this idea to UML fix generation. Yet, in configuration systems, the number of variables is often larger than the number of constraints. The actual reduction of code is thus not clear. Also, their approach generates only fixes that change one variable at a time, and does not support changing multiple variables

together. Steimann and Ulke [27] propose to use constraint solvers to generate fixes. Different from us, their approach enumerates all concrete fixes one by one, and does not try to characterize the solution space.

The problem of fixing programs have been approached from different angles, and the one closest to us is explored by Jose and Majumdar [28]. This approach employs a similar diagnosis algorithm to localize faults in imperative programs, by considering each statement as a retractable constraint. However, this approach cannot generate fixes for the localized faults, except for very special cases. Also, they propose at most one fix each time rather than a complete list.

Fix generation approaches for configurations also exist. The eCos configurator [2] has an internal fix generator, producing fixes upon request or on-the-fly when the user makes changes. White et al. [6] propose an approach to generate fixes that resolve all errors in one step. Compared with our work, both related approaches produce one concrete fix rather than a complete list of range fixes. Furthermore, they have very limited support of non-Boolean constraints. White et al.'s approach does not handle non-Boolean constraints at all, while eCos configurator supports only non-Boolean constraints in a simple form: $v \oplus c$ where v is a variable, c is a constant, and $\oplus$ is an equality or inequality operator. A particular sub-domain of software configuration is package management, where automatic approaches [29], [30] are proposed to determine how to install a new package as well as its dependencies. These approaches [29], [30] try to also find minimal solutions, but, in contrast to our approach, do not attempt to find complete sets of minimal solutions. Some other approaches [31], [32] consider the relation between a software system and its configurations, trying to localize problematic options when the system fails. Compared with them, our approach focuses on the configuration subsystem itself, and generate precise and complete fixes rather than identifying potentially problematic options. Furthermore, our approach still has the problem of large fixes. Wang et al. [33] try to address this problem by using dynamic priorities, and change only the options that are more likely to be changed by the user. Finally, a recent approach [34] addresses the problem of generating complete and optimal fixes for all errors in a Boolean feature model, and the feature model is partitioned to confine the number of fixes and accelerate the generation. Compared with this approach, our approach supports non-Boolean feature models, and generates fixes on a per-error basis to ensure the conciseness of the fix lists.

Another set of approaches maintain the consistency of a configuration. Valid domains computation [35], [36] is an approach that propagates decisions automatically. Initially all options are set to an unknown state. When the user assigns a value to an option, it is recorded as a decision, and all other options whose values are determined by this decision are automatically set. In this way, no error can be introduced. Janota et al. [37] propose an approach to complete a partial configuration by automatically setting the unknown options in a safe way. However, both approaches require that the configuration starts with variables in the unknown state. Software configuration in real world is often "reconfiguration" [3], i.e., the user starts with a default configuration, and then makes change to it. In reconfiguration cases, variables have assigned concrete values rather than the unknown state. Furthermore, the related approaches are designed for small finite domains, and it is not clear whether they are scalable to large domains such as integers.

Several approaches have been proposed to test and debug the construction of configuration models themselves. Trinidad et al. [38] use Reiter's theory of diagnosis [8] to detect several types of deficiencies in FODA feature models. Wang et al. [23] automatically fix deficiencies based on the priority assigned to constraints. These approaches target the construction of configuration models and cannot be easily migrated to the configuration process.

Others automatically fix errors without user intervention. Demsky and Rinard [39] propose an approach to fix runtime data structure errors according to the constraint on the data structure. Mani et al. [40] use the hidden constraints in a transformation program to fix input model faults. Xiong et al. [41] propose a language to construct an error-fixing program consistently and concisely. Compared with our approach, these approaches also infer fixes from constraints, but they only need to generate one fix that is automatically applied. Completeness is not considered by these approaches.

Interval propagation [42] is a technique to obtain the interval domain of each variable in a set of constraints without removing any value that is consistent with the constraints. Though interval constraint propagation has a similar form to range fixes, this technique cannot be easily used to obtain range fixes for the following reasons. (1) The intervals that we obtain from interval propagation can be larger than the precise range of variables, and it is difficult to satisfy Property 1 using interval propagation. (2) Interval propagation cannot support the case where several variables have to be involved in one fix unit.

Different diagnosis algorithms have been proposed [43], [44], [45], [46]. Some algorithms aim at different applications. For example, QUICKXPLAIN [43] incorporates user preferences to locate a diagnosis with a high preference. Felfernig et al. [44] modify HS-DAG [8], [9] to work with knowledge bases. Some other algorithms focus on finding diagnoses (called *correction subsets* in those approaches) without relying on the ability of finding unsatisfiable cores but employing various heuristics to cut off the search space. In particular, several new algorithms [45], [46] in this line are developed in parallel with our work. As a result, it would be interesting to compare the performance of our algorithm with these algorithms. We implemented the algorithm CLD [45] which is reported to be the fastest on SAT data set, and tested this algorithm on the largest Linux model, padx-room, in the same experiment setup as explained in Section 7. On this model, our algorithm returns a fix list within 20 seconds for 36.79 percent of the violations. On the other hand, CLD returns diagnoses within 20 seconds for 35.92 percent of the violations. The performances of CLD and ours are very close in this small experiment. A possible explanation is that modern SMT solvers have already exploited heuristics similar to those used in CLD to generate unsatisfiable cores, and relying on unsatisfiable cores could achieve a similar performance boost to using these heuristics.

11 Conclusion

Range fixes provide alternative solutions to constraint violations in software configuration. For single-constraint violations, they are correct, minimal in the number of variables per fix and per unit, maximal in their ranges, and complete. Range fixes can be generated efficiently on models with a few thousands of options and constraints.

We also evaluated three different strategies for handling the interaction of constraints (multi-constraint violations): ignorance, elimination, and propagation. No single strategy is absolutely better than the others, but on our data set, the propagation strategy provides the most complete fix lists without introducing new errors, and the fix sizes and generation times are within acceptable ranges. However, if more complex situations are encountered, elimination or ignorance can provide simpler fix lists and faster generation time, at the expense of completeness or the guarantee not to introduce new errors.

Traditional use of constraint solvers in software engineering research is to convert software engineering problems directly into constraint solving problems. Our design of Algorithms 2 and 3 suggests that, given the computational power of modern solvers, there is more potential to be exploited in these solvers in future. We may call them interactively, or explore their abilities of generating model instances and unsatisfiable cores, to solve a potentially larger class of software engineering problems.

Acknowledgments

The authors would like to thank Thorsten Berger at University of Leipzig for his patient lessons on eCos and CDL, and Gelin Zhou at University of Waterloo for fruitful discussion on the algorithm of this approach. This research was supported by the National Basic Research Program of China (973) under Grant No. 2014CB347701, the High-Tech Research and Development Program of China under Grant No.2013AA01A605, the National Natural Science Foundation of China under Grant No.61202071, 61332010, 61421091, and NSERC. Yingfei Xiong is the corresponding author.

References

[1] S. She and T. Berger. Formal semantics of the Kconfig language, Technical Note. [Online]. Available: http://gsd.uwaterloo.ca/node/274, 2010.

[2] B. Veer and J. Dallaway. (2001). The eCos component writer's guide. [Online]. Available: ecos.sourceware.org/ecos/docs-latest/cdl-guide/cdl-guide.html

[3] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability modeling in the real: A perspective from the operating systems domain," in *Proc. 26th Int. Conf. Autom. Softw. Eng.*, 2010, pp. 73–82.

[4] T. Berger and S. She. (2010). Formal semantics of the CDL language. [Online]. Available: www.informatik.uni-leipzig.de/berger/cdl_semantics.pdf

[5] A. Hubaux, Y. Xiong, and K. Czarnecki, "A user survey of configuration challenges in Linux and eCos," in *Proc. 6th Int. Workshop Variability Model. Softw.-Intensive Syst.*, 2012, pp. 149–155.

[6] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated diagnosis of product-line configuration errors in feature models," in *Proc. 12th Int. Softw. Product Line Conf.*, 2008, pp. 225–234.

[7] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wasowski, "A study of non-Boolean constraints in variability models of an embedded operating system," in *Proc. 15th Int. Softw. Product Line Conf.*, 2011, pp. 2:1–2:8.

[8] R. Reiter, "A theory of diagnosis from first principles," *Artif. Intell.*, vol. 32, no. 1, pp. 57–95, 1987.

[9] R. Greiner, B. Smith, and R. Wilkerson, "A correction to the algorithm in Reiter's theory of diagnosis," *Artif. Intell.*, vol. 41, no. 1, pp. 79–88, 1989.

[10] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 58–68.

[11] Y. Xiong, "Configurator semantics of the CDL language," Generative Software Development Laboratory, Univ. of Waterloo, Waterloo, ON, Canada, Tech. Rep. GSDLAB-TR 2011-06-05, 2011.

[12] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again," in *Proc. 11th Int. Softw. Product Line Conf.*, 2007, pp. 23–34.

[13] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *Proc. 25th Int. Conf. Softw. Eng.*, 2003, pp. 455–464.

[14] A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in UML design models," in *Proc. 23rd Int. Conf. Autom. Softw. Eng.*, 2008, pp. 99–108.

[15] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. 14th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.

[16] C. Barrett, A. Stump, and C. Tinelli. (2010). The SMT-LIB standard: Version 2.0. [Online]. Available: http://www.smtlib.org/

[17] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *Proc. 15th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2009, pp. 307–321.

[18] P. Bickford, "Worth the wait?" *Human Interface Online*, 1999, http://web.archive.org/web/20040913083444/http://developer.netscape.com/viewsource/bickford_wait.htm

[19] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *IEEE Trans. Softw. Eng.*, vol. 39, no. 12, pp. 1611–1640, Dec. 2013.

[20] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proc. 23rd ACM Symp. Oper. Syst. Principles*, 2011, pp. 159–172.

[21] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 140–151.

[22] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proc. 24th ACM Symp. Oper. Syst. Principles*, 2013, pp. 244–259.

[23] B. Wang, Y. Xiong, Z. Hu, H. Zhao, W. Zhang, and H. Mei, "A dynamic-priority based approach to fixing inconsistent feature models," in *Proc. 13th Int. Conf. Model Driven Eng. Lang. Syst.: Part I*, 2010, pp. 181–195.

[24] R. Olaechea, D. Rayside, J. Guo, and K. Czarnecki, "Comparison of exact and approximate multi-objective optimization for software product lines," in *Proc. 18th Int. Softw. Product Line Conf. - Volume 1*, 2014, pp. 92–101.

[25] K. Bak, K. Czarnecki, and A. Wasowski, "Feature and meta-models in clafer: Mixed, specialized, and coupled," in *Proc. 3rd Int. Conf. Softw. Lang. Eng.*, 2011, pp. 102–122.

[26] J. Scheffczyk, P. Rödig, U. M. Borghoff, and L. Schmitz, "Managing inconsistent repositories via prioritized repairs," in *Proc. ACM Symp. Document Eng.*, 2004, pp. 137–146.

[27] F. Steimann and B. Ulke, "Generic model assist," in *Model-Driven Engineering Languages and Systems*, New York, NY, USA: Springer, 2013, pp. 18–34.

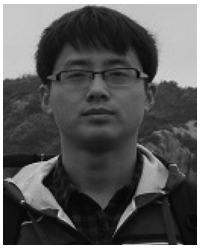
[28] M. Jose and R. Majumdar, "Cause clue clauses: Error localization using maximum satisfiability," in *Proc. 32nd Conf. Program. Lang. Des. Implementation*, 2011, pp. 437–446.

[29] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner, "Opium: Optimal package install/uninstall manager," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 178–188.

[30] J. Argelich, D. Le Berre, I. Lynce, J. Marques-Silva, and P. Rapicault, "Solving linux upgradeability problems using Boolean optimization," in *Proc. 1st Int. Workshop Logics Component Configuration*, 2010, pp. 11–22.

[31] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 312–321.
[32] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2011, pp. 193–202.
[33] B. Wang, L. Passos, Y. Xiong, K. Czarnecki, H. Zhao, and W. Zhang, "SmartFixer: Fixing software configurations based on dynamic priorities," in *Proc. 17th Int. Softw. Product Line Conf.*, 2013, pp. 82–90.
[34] J. Barreiros and A. Moreira, "A cover-based approach for configuration repair," in *Proc. 18th Int. Softw. Product Line Conf. - Volume 1*, 2014, pp. 157–166.
[35] 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 *Proc. Int. Conf. Econ., Techn. Org. Aspects Product Configuration Syst.*, 2004, pp. 131–138.
[36] M. Mendonça, "Efficient reasoning techniques for large scale feature models," Ph.D. dissertation, Univ. of Waterloo, Waterloo, ON, Canada, 2009.
[37] M. Janota, G. Botterweck, R. Grigore, and J. Marques-Silva, "How to complete an interactive configuration process?" in *Proc. 36th Conf. Current Trends Theory Practice Comput. Sci.*, 2010, pp. 528–539.
[38] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro, "Automated error analysis for the agilization of feature modeling," *J. Syst. Softw.*, vol. 81, no. 6, pp. 883–896, 2008.
[39] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *Proc. 18th ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl.*, 2003, pp. 78–95.
[40] S. Mani, V. S. Sinha, P. Dhoolia, and S. Sinha, "Automated support for repairing input-model faults," in *Proc. 26th Int. Conf. Autom. Softw. Eng.*, 2010, pp. 195–204.
[41] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, "Supporting automatic model inconsistency fixing," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Foundations Softw. Eng.*, 2009, pp. 315–324.
[42] E. Davis, "Constraint propagation with interval labels," *Artif. Intell.*, vol. 32, no. 3, pp. 281–331, 1987.
[43] U. Junker, "QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems," in *Proc. 19th Conf. Artif. Intell.*, 2004, pp. 167–172.
[44] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner, "Consistency-based diagnosis of configuration knowledge bases," *Artif. Intell.*, vol. 152, pp. 213–234, 2004.
[45] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov, "On computing minimal correction subsets," in *Proc. 23rd Int. Joint Conf. Artif. Intell.*, 2013, pp. 615–622.
[46] A. Nöhrer, A. Biere, and A. Egyed, "Managing sat inconsistencies with humus," in *Proc. 6th Int. Workshop Variability Model. Softw.-Intensive Syst.*, 2012, pp. 83–91.

Yingfei Xiong received the PhD degree from the University of Tokyo in 2009 and worked as a postdoctoral fellow at the University of Waterloo between 2009 and 2011. He is an assistant professor under the young talents plan at Peking University. His research interests include software engineering and programming languages.



Hansheng Zhang received the bachelor's degree in computer science and technology in 2012. He is a graduate student for the master's degree in computer software and theory at Peking University. He is expected to graduate in July, 2015.

Arnaud Hubaux received the PhD and postdoc degrees in computer science at the University of Namur, Belgium. He joined ASML in 2013, and is an enterprise architect expert in configuration and product variant management. His research interests include the formal integration of software and hardware variability, and collaborative configuration. Between October 2010 and September 2011, he was a visiting researcher at the GSD Lab of the University of Waterloo, Canada, where he worked on conflict detection and resolution in software configuration. He was an active member of the inter-university MoVES (Modelling, Verification and Evolution of Software) network from which he received the most promising young researcher award in 2010. He has published articles in premier journals and conferences including the *ACM Computing Surveys*, *IEEE Transactions on Software Engineering*, and the *International Conference on Software Engineering*.

Steven She received the PhD degree from the University of Waterloo, working on reverse engineering variability models. He is a data scientist at Canopy Labs. He developed techniques for reverse engineering feature models by reasoning on logical and probabilistic constraints. He also developed tools for analyzing the Linux Kconfig variability modelling language and creating a propositional formula for use with a reasoner.

Jie Wang is a first-year master student at Peking University. His research interests include software engineering and programming languages.

Krzysztof Czarnecki is a professor of electrical and computer engineering at the University of Waterloo. Before coming to Waterloo, he was a researcher at DaimlerChrysler Research (1995-2002), Germany, focusing on improving software development practices and technologies in enterprise, automotive, space and aerospace domains. He co-authored the book on *Generative Programming* (Addison- Wesley, 2000), which deals with automating software component assembly based on domain-specific languages. While at Waterloo, he held the NSERC/Bank of Nova Scotia Industrial Research Chair in Requirements Engineering of Service-oriented Software Systems (2008-2013) and has worked on a range of topics in model-driven software engineering, including software-product lines and variability modeling, consistency management and bi-directional transformations, and example-driven modeling. He received the Premier's Research Excellence Award in 2004 and the British Computing Society in Upper Canada Award for Outstanding Contributions to IT Industry in 2008.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.