Extending Static Analysis by Mining Project-Specific Rules

Boya Sun, Gang Shu, Andy Podgurski
EECS Department
Case Western Reserve University
Cleveland, OH
{boya.sun, gang.shu, podgurski}@case.edu

Brian Robinson
ABB Corporate Research
Raleigh, NC, USA
brian.p.robinson@us.abb.com

***Abstract*—Commercial static program analysis tools can be used to detect many defects that are common across applications. However, such tools currently have limited ability to reveal defects that are specific to individual projects, unless specialized checkers are devised and implemented by tool users. Developers do not typically exploit this capability. By contrast, defect mining tools developed by researchers can discover project-specific defects, but they require specialized expertise to employ and they may not be robust enough for general use. We present a hybrid approach in which a sophisticated dependence-based rule mining tool is used to discover project-specific programming rules, which are then transformed automatically into checkers that a commercial static analysis tool can run against a code base to reveal defects. We also present the results of an empirical study in which this approach was applied successfully to two large industrial code bases. Finally, we analyze the potential implications of this approach for software development practice.**

***Keywords*-static program analysis; defect mining; program dependence graphs**

I. INTRODUCTION

Static analysis tools are commonly used by software development teams to detect defects and enforce coding standards. The most basic tools, such as PC Lint [28], are essentially advanced compiler front-ends that check for violations of simple rules about proper use of particular language constructs. Advanced static analysis tools, such as Klocwork [14] and Coverity [6], can detect more complicated defects, such as ones that cause memory leaks or invalid data flows, by checking rules involving control flow and data flow paths (the rules may be intraprocedural or interprocedural). These advanced static analysis tools find potential defects by using *checkers*, which are written to search for specific patterns or pattern violations in source code. Issues are reported to the tool user, who must decide if each issue is a real defect or a false positive result (false alarm).

Advanced static analysis tools are designed to work on source code written with different design paradigms (object-oriented, procedural, etc.), different operating systems, and different application domains (real-time, embedded, server, web, etc). For this reason, the checkers that are shipped with a tool are intended to reveal defects that are common across many types of software systems. Ideally, the checkers should collectively find a high proportion of the actual defects in a system (exhibit high recall) and generate relatively few false positive results (exhibit high precision). In reality, recall and precision must be traded off to some extent.

Developers dislike using tools with low precision [3]; therefore commercial static analysis tools are typically configured to use general checkers having high precision across various types of software. While general checkers are useful, their recall may be quite low, since there are many programming rules thsat are specific to a software project. For example, implicit programming rules of a project may require that an application-specific function *f* be called before another application specific function *g* is called, or they may require that the return value of a given function be checked for error codes. Commercial static analysis tools do not check such rules by default, because they are not applicable across projects.

Advanced static analysis tools allow users to create new, "custom" checkers, based on project-specific rules, to detect additional defects (which are violations of these rules). These checkers can be created manually for rules involving properties of control flow or data flow paths. Unfortunately, project-specific rules are rarely documented, and some may be known by only a few programmers. In order to improve the defect detection of static analysis tools, these rules must be identified and converted into checkers that the tool can use. Due to the difficulty of obtaining project-specific rules and manually writing checkers for them, the advanced functionality of creating customized checkers is not fully utilized by software development organizations today.

This paper presents a framework for automatically identifying project-specific rules and creating custom checkers for a commercial static analysis tools. The framework is illustrated in Figure 1. It employs a recently developed dependence-based pattern mining technique to identify frequent code patterns that are likely to represent programming rules. These patterns are currently transformed automatically into customized checkers for *Klocwork*, a commercial static analysis tool used by ABB, although our framework can be adapted to work with other tools. Finally, the Klocwork analysis engine uses these checkers to find rule violations in the code base, which are then examined by developers. To evaluate this framework, we applied it to the code bases of two large ABB software products. The results indicate that: (1) Klocwork checkers can be created for most of the identified rules and (2) the transformed checkers accurately report defects involving violations of project-specific rules, which were not detected by Klocwork's built-in checkers.

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

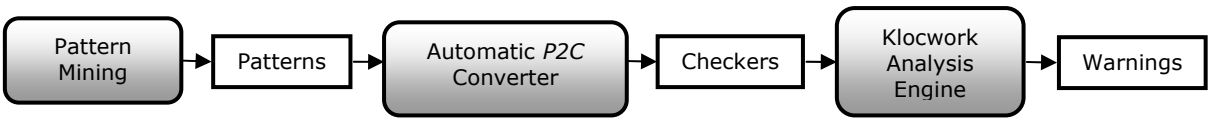


Figure 1. Framework

The rest of this paper is organized as follows. Section 2 presents background information on existing pattern mining techniques, as well as an overview of Klocwork and its rule checkers. Section 3 describes the automatic conversion of patterns into Klocwork checkers. Section 4 presents the experimental study and results. Our lessons learned are discussed in Section 5. Section 6 discusses related work and Section 7 concludes the paper.

II. BACKGROUND

In this section, we introduce the technique used to mine programming rules, which correspond to frequent code patterns in the source code. We then introduce Klocwork, the commercial static analysis tool used in this study, and describe how we use the mined rules to create new checkers.

A. Mining Frequent Code Patterns

Frequent pattern mining techniques have been used to discover project-specific bugs by several groups of researchers [2][4][5][9][21][22][23][30][32][33][34]. These approaches employ techniques for frequent itemset mining, frequent sequence mining, or frequent subgraph mining to discover recurring code patterns that may correspond to undocumented programming rules. It is assumed that sufficiently frequent patterns are likely to correspond to actual programming rules. The candidate rules are reviewed by developers to identify the real rules. Once programming rules are discovered and confirmed, techniques for approximate pattern matching can be used to find small deviations from the rules, which are called *rule violations*. Violations are examined by developers to determine if they are actual bugs.

To discover project-specific programming rules, we use the approach developed by Chang et al [4][5], which applies *frequent subgraph mining* to the System Dependence Graph (SDG) representation [10][13] of a project's code to discover neglected conditions and other types of project-specific bugs. The SDG is a directed labeled graph that represents data or control dependences among program statements. It captures semantically important ordering constraints among statements and omits inessential ones, so that statement reorderings that do not alter dependences also do not affect semantics. Thus, a given SDG subgraph can model minor (dependence preserving) variations of a programming pattern or rule. Mining rules from the SDG tends to be more precise than using frequent-itemset mining or frequent-sequence mining, even for rules with relatively few instances, because dependences between the statements comprising a rule are accounted for.

Chang et al's approach involves first extracting a collection of *dependence spheres*, which are dependence subgraphs of limited radius that are expanded from a central function call, say *f*(…). The spheres are pruned of irrelevant nodes and edges, and a frequent subgraph mining algorithm is applied to the collection of pruned spheres to find subgraphs that recur among them. Each such subgraph is examined manually to determine if it represents a programming rule that should be followed whenever the function *f*(…) is invoked. A basic rule can be expressed in the form:

$$\boldsymbol{f(\ldots)\ \text{is called} \Rightarrow \text{constraint } C \text{ is satisfied}}$$

where ⇒ means "implies". Three basic kinds of rules, involving different kinds of constraints *C*, are *precondition rules*, *postcondition rules*, and *call-sequence rules*. A precondition rule requires programmers to ensure that a certain input parameter is valid. To do that, a conditional check of the parameter's validity is usually performed before passing it to the function. Such a check can be omitted only if the value is certain to be valid, e.g., when a valid constant is assigned to the variable before it is passed to the function. Postcondition rules take one of two different forms. In a *Type 1* postcondition rule, one should perform a conditional check on the returned value before it is used elsewhere in the program, unless the return value is guaranteed to be correct. In a *Type-2* postcondition rule, the return value is a constant representing the completion status of the function call, which should be checked so that abnormal completion can be handled appropriately. A *call sequence* rule specifies that a set of functions should be called in a certain order. A *call-pair* rule is a simple call sequence rule requiring that when a given *f* function is called, another function *g* should be called before or afterwards.

If a programming rule consists of a single precondition rule, postcondition rule, or a call pair rule, we call it a *simple rule*. If a rule involves several pre or post conditions, more than two functions, or a combination of simple rules, we call it a *hybrid rule*. Table I shows examples of simple rules. The underlined code in each example implements the consequent (constraint part) of the corresponding rule.

Although we focus on frequent code patterns mined using the dependence-based approach developed by Chang et al [4][5], the issues addressed by our framework are quite common, and it could be used with other approaches to automatically discovering programming rules (e.g., [2][21][22][23][30][32][33][34]).

B. Static Analysis Tools and Custom Checkers

For this study, we selected *Klocwork* [14], a popular static program analysis tool that can detect security vulnerabilities, quality defects, and architectural issues in C, C++, Java, and C# programs. Klocwork has default checkers [16] which detect common issues in C and C++ programs, such as possible buffer overflow vulnerabilities, null-pointer

TABLE I SIMPLE RULES

Rule type	Rule	Example	XML RuleSpec	Checker
Precondition	$f(x)$ ⇒ ensure x is valid	`if(x <= MAX)` `f(x);`	Value constraint	Data flow

Postcondition	**Type 1:**
	$y = f(x)$ ⇒ ensure y is valid
		`y = f(x);`
		`if(y == null)`
		`return;`
			Value constraint	Data flow

	Type 2:
	$status = f(x)$ ⇒
	ensure $status$ is checked to handle different completion statuses
		`status = f(x);`
		`if(status == FAIL)`
		`//do something`
			Conditional constraint	Control flow

Call pair	$f(x)$ ⇒
	ensure function $g(..)$ is called before or after $f()$
		Example-1:
		`y = f(x);`
		`g(y);`
			Call constraint	Control flow

		Example-2:
		`g(x);`
		`f(x);`		

dereferences, and memory leaks. Klocwork was selected for this study since it is used by development teams at ABB.

Klocwork provides the Klocwork Extensibility Interface [15][18] for creating custom checkers [17]. Other static analysis tools also support the creation of custom checkers. For example, Coverity provides a Static Analysis SDK [7] for this purpose. Each of these tools provides two types of checkers: AST checkers and path checkers. *AST checkers* are used for code-style analysis, while *path checkers* are used to perform control flow and data flow analysis. Writing effective checkers requires skill, even when the rules to be checked are well specified. We transform mined programming rules, corresponding to SDG subgraphs, into path checkers so that we can employ Klocwork to find violations across the entire code base.

Klocwork path checkers perform automatic source-to-sink data flow and control flow analysis. The source and sink are program elements of interest, such as program variables. Users specify them using the Klocwork C/C++ Path API, so each checker is a piece of C++ code. Once the source and the sink are specified, the path checker analysis engine checks whether there is a data flow or control flow from the source to the sink, and, if one exists, it reports an issue. A path checker may also employ the following capabilities provided by the Klocwork C/C++ Path API:

- Control-flow traversal – the checker can traverse the control flow graph of a function.
- Value tracking – the checker can determine whether two variables contain the same value or point to the same memory block
- Value constraints – the checker can determine if certain constraints hold on values in memory, such as upper and lower bounds.

III. GENERATING CHECKERS FROM MINED PATTERNS

In this section, we describe the *Pattern-to-Checker* (*P2C*) *Converter*, which is a prototype tool we have implemented to automatically transform frequent code patterns, mined using Chang et al's approach [5], into Klocwork path checkers. The P2C Converter consists of two main components, which carry out the transformation illustrated in Figure 2. The *Rule Extractor* extracts programming rules from the mined patterns, which are in the form of generic SDG subgraphs. To obtain rules of the forms shown in Table I, the Rule Extractor analyzes the dependence graph structure of a pattern and the abstract syntax trees (AST) of its statement instances. The Rule Extractor generates XML rule specifications, called *XML RuleSpecs*, conforming to an XML schema that we defined.

The other main P2C component, called the *Checker Generator*, transforms XML RuleSpecs into Klocwork checkers. This design make our framework more flexible, since programmers can use XML RuleSpecs to directly specify other programming rules they wish to enforce and then use the Checker Generator to transform them into checkers, obviating the need to write the checkers themselves. We have provided a web demo to illustrate the transformation of an XML RuleSpec into Klocwork checkers [35].

In the remainder of this section, we describe the Rule Extractor and Checker Generator in detail. We use the following code example throughout this section to illustrate how a pattern is transformed into checkers:

Example 1:

```
1.  if(x <= MAX)
2.  {
3.        status = f(x);
4.        if (status == FAIL)
5.              return with error;
6.  }
```

A. Rule Extractor

Since the mined patterns do not represent programming rules directly in the forms shown in Table I, we need a means to automatically extract programming rules of those

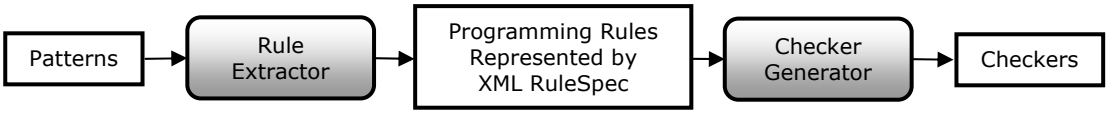


Figure 2. Transformations made by *P2C* Converter components

forms from the patterns. The rules are expressed using XML RuleSpecs. We first describe how programming rules are extracted from patterns, then we describe how the rules are represented using XML Rule Specs.

1) Extracting Programming Rules from Patterns: Mined patterns can be either simple patterns or hybrid patterns. A simple pattern involves only one precondition or postcondition, or it involves only one pair of functions. Thus a precondition, postcondition, or call pair rule can be extracted directly from a simple pattern. A hybrid pattern is a more complex pattern that involves multiple conditions and/or functions. By analyzing the dependence graph structure of a hybrid pattern, it can be decomposed into a conjunction of simple rules:

$$[\boldsymbol{f_1}(\ldots) \Rightarrow \boldsymbol{C_1}] \wedge [\boldsymbol{f_2}(\ldots) \Rightarrow \boldsymbol{C_2}] \wedge \ldots \wedge [\boldsymbol{f_n}(\ldots) \Rightarrow \boldsymbol{C_n}]$$

Such a rule conjunction is satisfied if each constituent rule is satisfied. If any of the latter rules is violated, then the conjunction is violated.

To extract programming rules, we need to extract *constraints* by analyzing dependence graph structures and sometimes also the abstract syntax trees of the statements. We now describe three kinds of constraints involved in precondition, postcondition and call sequence rules, namely value constraints, conditional constraints and call constraints, and explain how they are extracted from a pattern. The constraint types for different programming rules are summarized in Table I.

A *value constraint* on the value of an input or output parameter of a function is a constraint that can be checked by the Klocwork engine (statically). For example, a value constraint can specify upper and lower bounds for an input or output parameter or specify that a pointer cannot be NULL. Precondition rules and Type 1 postcondition rules specify value constraints on input and output parameters, respectively. Value constraints are identified by analyzing the abstract syntax tree of branch conditions from instances of the pattern.

A *conditional constraint* requires the presence, in the program code, of a conditional check on an input or output parameter, before or after a given function is called. A Type 2 postcondition rule specifies such a constraint. Conditional constraints can also be used when value constraints cannot be checked statically. For example, to ensure that a parameter value is within a certain range that can be determined only at runtime, a conditional constraint can be written that requires the presence of a conditional check on the parameter value.

A *call constraint* requires the presence of another function call before or after a function *f*() is called. Call sequence rules specify call constraints involving two or more function calls. For call sequence patterns involving one pair of functions, it is necessary to decide which function should be on the left hand side of the rule and which should be on the right hand side. In such cases, we generate programming rules for both cases, and let programmer decide whether to keep both of the rules or remove one of them.

2) XML RuleSpecs: XML RuleSpecs are used to specify a programming rule in the forms shown in Table I. The schema for XML RuleSpecs supports specification of value, conditional, and call constraints on a call to a given function f(). The XML specification has two main elements. The first element, **`<Function>`**, indicates the function f() whose calls are subject to constraints. The second element, **`<Constraints>`**, specifies one or several constraints as described above. The three types of constraints are expressed by the following three types of specification elements:

`<ValueConstraint>`: This element indicates which input or output parameter is involved in the constraint, and it specifies value constraints on this parameter. Currently we support constraints on numerical values, including upper bounds, lower bounds, ranges, legal values, and illegal values (including NULL for pointers).

`<ConditionalCheckConstraint>`: This element indicates which input or output parameter should be checked and whether the check should appear before or after the call to the function.

`<CallConstraint>`: This element specifies a function *g*() that should be called together with *f*() and indicates whether it should be called before or after *f*(). Relationships between parameters and return values of the two functions can also be specified. For example, a call constraint for the call sequence rule ***y* = *f*() ⇒ ensure *g*(*y*) is called afterward** indicates that *g*() should use the return value of *f*() as its input parameter.

Illustration. From **Example 1**, two programming rules for invoking a function *f*() could be extracted as follows:

Rule-1:

$$f(x) \Rightarrow x \text{ should not exceed MAX}$$

Rule-2:

$$f(x) \Rightarrow \text{check completion status}$$

Rule-1 is a precondition rule that specifies a value constraint. It indicates that the value of parameter `x` should not exceed `MAX`, which is a constant value. Rule-2 is a Type-2 postcondition rule that specifies a conditional constraint. It indicates that the return value of *f*() should be checked to make sure that abnormal completion is handled appropriately. The two rules can be expressed by the XML RuleSpec shown in Figure 3, assuming that `MAX` = 10.

```
<root>
   <Function>
      <name>f</name>
      <return id="0">
         <type>int</type>
      </return>
      <param id="1">
         <type>int</type>
         <declname>x</declname>
      </param>
   </Function>
   <Constraints>
      <ValueConstraint>
         <param>
            <id>1</id>
         </param>
         <SmallerEquals>10</SmallerEquals>
      </ValueConstraint>
      <ConditionalCheckConstraint>
         <param>
            <id>0</id>
         </param>
         <Location>post</Location>
      </ConditionalCheckConstraint>
   </Constraints>
</root>
```

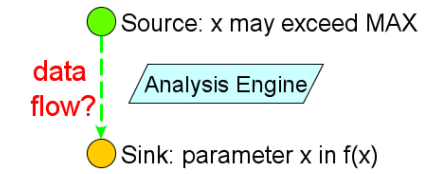
Figure 3. XML RuleSpec for **Example 1**

B. *Checker Generator*

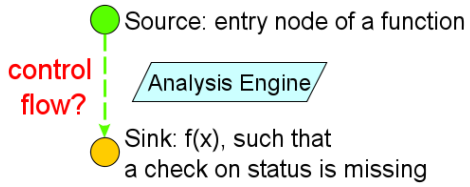
The Checker Generator takes an XML RuleSpec as input and produces Klocwork path checkers automatically. Since an XML RuleSpec may specify one or several constraints, a checker will be generated for *each* constraint to check whether it is violated.

As described in Section II.B, Klocwork path checkers perform source-to-sink data flow or control flow analysis, and an error is flagged at the sink if a data flow or control flow from the source to the sink is detected. For a value constraint, the checker generator produces a checker that flags an error if an illegal value propagates from the source to the sink via a data flow. Figure 4 (a) shows such a checker for Rule-1 from **Example 1**. If a variable `x` with a value possibly greater than `MAX` reaches the input parameter of *f*(), the checker flags an error.

For a conditional constraint or a call constraint**,** it is necessary to check whether a conditional test or a call is *missing* when a certain function is invoked. Such constraints cannot be checked directly using either a data flow source-sink analysis or a control-flow source-sink analysis. To address such constraints, we devised the following scheme: the *source* is specified as the entry node to any function that may call *f*(), and the *sink* is a call to a function *f*(), such that either a required conditional check (for a conditional constraint) or required call to another function *g*() (for a call constraint) is missing**.** If there is a control flow from the source to the sink then there is a call to *f*() with a missing constraint. Figure 4 (b) shows a checker for Rule-2 of **Example 1,** which is a conditional rule. The *source* is specified as an entry node; the *sink* is specified as a call to *f*() such that a check on `status` is missing. An error will be flagged if there is a control flow from the source to the sink. To check whether a conditional test or a function call is



(a) Checker for **Rule-1**, which is based on data flow analysis



(b) Checker for **Rule-2**, which is based on control flow analysis

Figure 4. Checkers for **Example 1**

missing, we need to traverse the control flow graph backwards or forwards from the sink, and do possible value tracking using the Klocwork C/C++ Path APIs [15]. For conditional constraints, we traverse *backwards* from the sink to search for a conditional test on the input, and *forwards* to search for a test on return value. For call constraints, if *g*() should be called *after f*(), we traverse *forwards* from the sink to search for the call to *g*(). If *g*() should be called *before f*(), we traverse *backwards* from the sink to search for the call to *g*(). Value tracking will be used to address the relationships between parameters and return values of the two functions, as described in **`<CallConstraint>`** of Section III.A.2).

The checker generator needs to implement two functions to extract sources and sinks, namely *SourceExtractor* (*node*), and *SinkExtractor*(*node*). These two functions allow the path checker to interact with the backend path checker analysis engine. The engine traverses the control flow graph of the current function under analysis and passes the current node being traversed, represented below as *node*, to *SourceExtractor*() and *SinkExtractor*(). The functions extract expression(s) of interest from the node, which are returned to the engine. *Klocwork* provides a trick to allow the two functions to specify whether data flow or control flow analysis is to be performed, which we have learned from its support site [19]. Details are omitted here due to space.

We will use Rule-1 and Rule-2 of **Example 1** as examples to illustrate how *SourceExtractor*() and *SinkExtractor*() are written.

Rule-1. This rule specifies a value constraint on input parameter of *f*(), therefore data flow analysis is performed to check a violation to the constraint. The *SourceExtractor*() and *SinkExtractor*() are specified below:

***Expression* ← *SourceExtractor*(*node*)**
if a variable *v* is defined in *node*, and it is possible that *v* > MAX, **return** *v*

***Expression* ← *SinkExtractor*(*node*)**

if *node* is a call to *f*()
return first parameter of *f*()

Rule-2. This rule specifies a conditional constraint on *f*(), therefore control flow analysis is performed to check for a violation of the constraint. The *SourceExtractor*() and *SinkExtractor*() are specified below:

***Expression* ← *SourceExtractor*(*node*)**
if *node* is an entry to a function
return *node*

***Expression* ← *SinkExtractor*(*node*)**
if *node* is a call to *f*()
expr ← output parameter of *f*()
Traverse the control flow graph in the forward direction:
if the current node is a check of *expr*, **return** NULL;
return call to *f*()

IV. Experimental Study

We conducted an empirical study to assess the practicality and effectiveness of our automated framework for deriving project-specific rule checkers. The study addressed the following research questions:

R1: How general is our automatic framework to deriving checkers? Specifically,

- R1-1 ("Precision" of checker generation): what percentage of automatically generated checkers are actually valid?
- R1-2 ("Recall" of checker generation): how many rules (XML RuleSpecs) can be transformed into checkers automatically so that programmers do not have to write checkers for them manually?

R2: How effective are the generated checkers? Specifically,

- R2-1 (Precision of warnings produced): what percentage of checker-generated warnings are valid?
- R2-2: Compared to Klocwork default checkers, how often are new valid warnings produced using the generated checkers?

A. Preparing Patterns for Analysis

1) Mining Patterns from the Code Base: We applied our framework to the code bases of two industrial software systems, which we shall call System X and System Y, which were developed by separate product groups at ABB. Both systems contain a mixture of C and C++ code. System X is the larger system, with more than 5 million SLOC. Chang et al's pattern mining tool [5] was applied to one of its largest components, which contains 2.7 million SLOC and which we call Component A. The other software system, System Y, contains around 2 million SLOC, and we applied the mining tool to the entire system. We used CodeSurfer [11] to generate SDGs for the two systems. Chang et al's tool mined a total of 1112 frequent code patterns from Component A of System X, and it mined 748 patterns from System Y.

TABLE II DISTRIBUTION OF MINED PATTERNS

	Useful patterns	Common usage scenarios	False Positives
System X	103 (68.7%)	34 (22.7%)	13 (8.7%)
System Y	127 (63.5%)	39 (19.5%)	34 (17%)

2) Sampling from the Mined Patterns: To reduce the amount of effort needed for our study (which includes reviewing the patterns, generated checkers, and reported warnings), we did not apply our framework to the entire pattern set. Instead, we selected random samples of 200 patterns from System X and 150 patterns from System Y, respectively. To obtain a diverse and representative set of patterns, a stratified random sampling scheme was employed. We partitioned the entire pattern set into precondition patterns, postcondition patterns, call sequence patterns, and hybrid patterns, and we sampled a number of patterns from each category according to its size relative to the entire set.

3) Filtering the Sampled Patterns: We manually examined each of the patterns and put them into one of the following three categories: Useful patterns are those from which a rule or a set of rules like those shown in Table I could be extracted. Common usage scenarios are patterns that do not satisfy the criteria for useful patterns but that still represent common coding scenarios. One example is that a get function is often called after a set function. Although this is an interesting pattern, it is not a useful call pair rule, since get and set can be invoked independently. False positive patterns are those with no apparent practical significance. Checkers were generated for just the useful patterns.

To make sure that we obtained a fair categorization of the patterns, the first two authors of the paper reviewed disjoint subsets of the sampled patterns, each comprising 45% of the total, and they *both* reviewed the remaining 10% of the sampled patterns independently, to see whether they generally agreed on the categorization. They disagreed on only one of the latter patterns. The distribution of the patterns is shown in Table II.

To better evaluate the precision of the mined patterns, we conducted a small survey of the developers of *System Y*. We created three surveys, each containing 10 randomly selected patterns, and we asked three different groups of developers to classify the patterns. The first group of developers classified all patterns to be common usage scenarios; the second group of developers classified 5 patterns as useful, 4 as common usage scenarios, and 1 as a false positive; the third group of developers classified 7 patterns as useful, 2 as common usage scenarios, and 1 as a false positive. It seems that the first group of developers was more conservative, since their comments on some patterns indicated that they

TABLE III CHARACTERIZATION OF CHECKERS FOR SYSTEM X

		Value Constraints	Conditional Constraints	Call Constraints	Total	*CG* Prec	*CG* Recall
Generated	Precise	34	70	22	126	78.3%	96.9%
	Imprecise	0	13	0	13		
	False Positive	1	0	21	22		
Manually Created		1	0	3	4		
Total		36	83	46	165		

TABLE IV CHARACTERIZATION OF CHECKERS FOR SYSTEM Y

		Value Constraints	Conditional Constraints	Call Constraints	Total	*CG* Prec	*CG* Recall
Generated	Precise	70	36	14	120	81.1%	96.8%
	Imprecise	0	9	0	9		
	False Positive	4	1	14	19		
Manually Created		0	0	4	4		
Total		74	46	32	152		

are useful under certain circumstances. The second and third group of developers thought the majority of patterns were useful ones, and the proportions of these patterns (50% and 70%) roughly agree with that shown in Table II (63.5%).

B. R1: Generality of Checker Generation

In total, 161 and 148 checkers were generated for System X and Y, respectively. To assess the generality of checker generation, we reviewed each of the checkers to decide whether it was precise, imprecise, or a false positive. A checker for a rule was considered *precise* if it would check exactly the required constraint(s), whereas an *imprecise* checker could produce false positive warnings. A checker for a rule was considered a *false positive* if it did not actually check the constraints in the rule. That is, the rule cannot be addressed using our current XML RuleSpec framework, though might be possible to create a suitable checker manually. Based on the above definition, we can define the *precision* and *recall* of the checker generator (*CG*) as follows:

$$CG\ \text{Precision} = \frac{|\text{precise checkers}|}{|\text{all generated}|}$$

$$CG\ \text{Recall} = \frac{|\text{precise checkers}|}{|\text{precise checkers \& manual}|}$$

The generated checkers are characterized in Table III, for System X, and Table IV, for System Y. For both projects, around 80% of the generated checkers were precise checkers, and we needed to manually create only four checkers for each project. Next, we discuss the sources of false positives and imprecision:

Sources of false positives: (1) *Incomplete constraints.* Some patterns could not be fully specified using the three types of constraints described in this paper. For example, we observed some patterns expressing the following programming rule: ***f*(…) is called and *f*(…) returns a value *V* ⇒ *g*(…) should be called.** This rule could not be fully expressed by our call constraint, since it has an additional requirement on the returned value of *f*(). For such rules, we must manually create checkers to fully express the constraints. (2) *Reversed call constraint.* If the mined pattern is a call pair rule involving two function calls *f*() and *g*(), there are two possible rules: ***f*() called ⇒ ensure *g*() called afterward**; or ***g*() called ⇒ ensure *f*() called before**. There is no way to decide which one is the correct programming rule, or whether both of them are correct. Therefore, we generated checkers expressing both of the rules automatically, and let programmers decide which one they want to use. If only one of them is correct, the other one will be a false positive.

Sources of imprecision: (1) *Runtime behavior*. Some constraints could not be checked statically by the checkers. For example, the following rule requires a parameter to be within the range of an array: ***sort*(*list*, *size*) called ⇒ ensure *size* is within the range [0, length(*list*)]**; however, the range of the array could not be determined statically. In this case, we can write a checker based on a conditional constraint instead of value constraint, that is, check for a runtime test on the value of ***size***. This could produce false positive warnings if the parameter is sure to be within the range and no check is needed. (2) *OR rules*. Sometimes two constraints work as alternatives to each other. For example, we found the following *OR* rule in our experiment: ***speed* returned by *f*(…) ⇒ ensure *speed* does not exceed limit OR *speed*time* does not exceed limit**. In this case we have to write two separate checkers to check the two constraints "***speed* does not exceed limit**" and "***speed*time* does not exceed limit**" when *f*() is called. Therefore, false positive warnings will be produced when code satisfies only one of the constraints but not the other. (3) *Value constraints on structure fields or class members*. We found some rules that require value constraints on a field of a structure or a member of a class. Our current framework implementation does not support such constraints, which require traversing

TABLE V CHARACTERIZATION OF WARNINGS IN SYSTEM X

	Value Constraints	Conditional Constraints	Call Constraints	Total	%
Valid	36	16	42	94	**94%**
Invalid	3	0	3	6	**6%**
Total	39	16	45	100	100%

TABLE VI CHARACTERIZATION OF WARNINGS IN SYSTEM Y

	Value Constraints	Conditional Constraints	Call Constraints	Total	%
Valid	123	19	27	169	**84.5%**
Invalid	23	8	0	31	**15.5%**
Total	146	27	27	200	100%

the memory blocks of a structure or a class. We will implement this feature in the future.

C. R-2: Effectiveness of the Generated Checkers

To assess the effectiveness of the generated checkers, we estimated the precision of the reported warnings, and we compared the generated checkers with the Klocwork default checkers to see how many new valid warnings were produced.

1) Precision of the Warnings: Ideally, precision of the warnings should be the proportion of warnings indicating actual bugs among all bugs reported. This proportion could be obtained only by submitting all warnings to programmers and asking them to decide whether or not each warning is a bug they should fix. This was infeasible, due to the large number of warnings produced, so we estimated precision by the proportion of valid warnings. Valid warnings represent true violations of the corresponding rule, namely: if the rule is **f(…) is called ⇒ constraint C is satisfied**, true violation should be **f(…) is called ⇒ constraint C is NOT satisfied .** For each warning, we manually determined whether it was valid or invalid. Our estimate is likely to be an overestimate of the true precision. However, the valid warnings at least indicate dubious programming practices that programmers should look into.

Our approach produced altogether 691 warnings for System X and 2335 warnings for System Y. To limit our effort, we selected a random sample of 100 warnings from System X and 200 warnings from System Y, and we examined each of them carefully to decide whether they were valid. Table V and Table VI characterize the precision of the warnings issued for the two systems. System X and Y achieved precision of 94% and 84.5% respectively, which are both high levels of precision. The precision measures varied different types of checkers. Call constraint checkers seem to have slightly better precision than the other two types of checkers.

Sources of the imprecision that did occur were as follows: (1) *Imprecise data flow analysis*. For example, an invalid warning concerning the validity of the return value of *f*() was as follows:

```
1:    x->a = f(…);
2:    g(x); //Warning reported
```

The warning was reported because Klocwork detected a data flow from `x->a` to `x`. This indicates that the analysis engine could not distinguish a structure variable from its fields. (2) *Interprocedural analysis*. Klocwork provides interprocedural data flow analysis, but the control flow analysis is intraprocedural. Since checking of conditional constraints and call constraints is based on control flow analysis, an invalid warning may be produced due to lack of interprocedural analysis. For example, although *g*() is not called after *f*() in a function *h*(), it might be called in a caller or callee of *h*(). A false positive warning will be reported in either case.

2) Comparison between Warnings Produced by Default Checkers and Those Produced by Generated Checkers: We ran default checkers against the code bases, and 109 warnings and 1146 warnings were produced for System X and System Y, respectively. We compared them with the set of warnings produced by the generated checkers. There was no overlap between the two sets of warnings for System Y, and there were only 4 warnings for System X produced by both kinds of checkers. These 4 warnings were produced by the default checker NPD.FUN.MUST, which checks whether the return value of a function is NULL; and by generated checkers which check value constraints on the return values of functions to make sure that they are not NULL. One thing to note here is that NPD.FUN.MUST can check only for functions that explicitly return a NULL pointer, such as "`return 0`", while generated value constraint checkers can check any functions that could possibly return a NULL.

V. LESSONS LEARNED

At the outset of this work, our approach was to manually write checkers for mined rules. The first author sampled

around 40 mined patterns and wrote checkers for them. We found that this task requires expertise with both the mining tool and the static program analysis tool. First of all, one needs to understand the mined patterns and be able to translate them into programming rules. This requires one to also have knowledge about the underlying dependence graph structure of the mined patterns. Secondly, one also needs to know the checker extension functionality of the static analysis tool very well, since many details need to be handled. The following are some examples of the subtleties that arise when writing checkers: (1) traversing the abstract syntax tree of a statement to obtain a certain parameter; (2) extracting value constraints for an arbitrary parameter, including value constraints for a memory block referenced by a pointer; (3) deciding whether two parameters are the same, or whether they point to the same memory block; (4) handling cases in which the return value of a function is omitted. We concluded that either a special static analysis specialist or an automated framework was needed to do the work. In our current automated checker generation framework, we handled all subtleties mentioned above, so that programmers do not need to understand a lot about the mined rules. Therefore, we think the framework will greatly improve the practicality of discovering and checking project-specific rules and will also save companies the cost of hiring an expert to do the work.

We have also learned important lessons from the survey we conducted to access the quality of the mined rules. We learned from the programmers' responses that it was hard for them to draw a fine line between a programming rule and a common usage scenario, even if they fully understand the semantics of the pattern. One explanation for this is that programmers need stronger evidence to decide the usefulness of a pattern than is provided by a couple of code examples. Therefore, we think that a better way is to let programmers decide the usefulness of patterns over the course of development. Mined patterns could be placed on a "watch list", which is updated automatically when new evidence about a pattern's status is obtained. For example, the priority of a pattern can be elevated if a defect is found in code that matches the pattern.

VI. Related Work

To the best of our knowledge, this study is the first research which integrates a commercial static analysis tool with a tool that discovers programming rules by mining frequent patterns. We summarize previous work on defect mining and application of static analysis tools in this section.

A. Mining Frequent Patterns as Programming Rules

In this work we applied Chang et al's technique [4][5] to produce frequent code patterns. There are other pattern mining techniques that could also be used to detect rules and extend this approach. Engler et al were among the first to propose the idea of treating bugs as violations to programming rules [9]. In their approach, programmers are required to write rule templates to find bugs. Livshits and Zimmermann [22] developed a tool called *DynaMine* to discover coding rules involving correlated method calls from a source code revision history. Li and Zhou proposed an approach to finding programming rules and rule violations using frequent itemset mining [21]. In a later work, they developed a tool called MUVI [23] to detect bugs involving inconsistent updates of correlated global variables or structure fields. Acharya et al presented an interprocedural approach to discovering client-side API usage patterns [2] by mining execution traces generated by an adapted model checker. Wasylkowski et al developed a tool called JADET to detect ordering patterns among method calls of objects by mining finite state automata. Thummalapenta et al [32] mines exception handling rules by mining conditional association rules. They also developed a tool *Allatin* [33] which is used to find alternative rules.

B. Applying and Augmenting Static Analysis Tools

Both open source and commercial static analysis tools have been applied for early bug detection, and researchers have reported their experiences in various work. Engler et al [12] applied the *Coverity* commercial tool to the Linux kernel and indicated that such a static analysis tool can help programmers by focusing on source code with high bug density. Krishnan et al [20] applied the *Klocwork Inforce* tool to enforce secure coding standards in Motorola's products. Nagappan et al [25] reported experience in applying two static analysis tools, *PREFix* and *PREFast* at Microsoft, to predict pre-release defect density. Zheng et al [36] applied a set of static analysis tools, including *Klocwork*, *FlexLint* and *Reasoning*, to an industrial software system developed at Nortel Networks, and showed that the tools provide affordable means for software fault detection.

Some other research work addresses augmentation of static analysis tools. Csallner et al [8] combined static analysis with a concrete test-case generation tool to eliminate spurious warnings. Nanda et al [26] addressed a tool *Khasiana* which was developed in IBM to combine functionalities of three static analysis tools: *FindBugs*, *SAFE* and *XYLEM*. Phang et al [27] created a tool called *Path Projection* on top of a static analysis tool *Locksmith* in order to visualize and navigate program paths. Ruthruff *et al* [29] propose a logistic regression model to classify both accurate and actionable static warnings that are produced by the *FindBugs* tool. All the above mentioned work augments static analysis tools by enhancing the accuracy or understandability of the warnings produced. However, none of the work addresses the issue on expanding the tools to find project-specific defects, as was done in our work.

VII. Conclusions

We have developed a novel approach to obtain checkers for project-specific programming rules, which may be run with the help of advanced static program analysis tools such as Klocwork. We implemented an automated framework to transform rules discovered by a prototype rule mining tool into Klocwork path checkers. The results of our analysis indicate that the approach is practical and generalizable, as it is able to transform most patterns into checkers precisely. The results also indicate that our technique enhances the defect detection of the static analysis tool by generating new

and precise warnings. Our framework applies to function precondition and postcondition rules as well as call sequence rules, regardless of how they are mined from the code base. We also plan to extend the current work to address more types of programming rules in the future.

ACKNOWLEDGMENT

We would like to thank ABB Corporate Research for supporting this work. We also thank the National Science Foundation for supporting this work with awards CCF-0702693 and CCF-0820217. Moreover, we would like to thank the *Klocwork* support team for their help, and Mithun Acharya for his help in building CodeSurfer projects.

REFERENCES

[1] Automatic Klocwork Checker Generator from XML RuleSpec Web Demo: http://selserver.case.edu:8080/autochecker/index.htm

[2] M. Acharya, T. Xie, J. Pei, and J. Xu. "Mining API Patterns as Partial Orders From Source Code: from Usage Scenarios to Specifications," in Proceedings of The 6th joint meeting of European Softw. Eng. and ACM SIGSOFT Symp. Found. of Softw. Eng., Dubrovnik, Croatia, pp. 25-34. 2007.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak and D. Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," Communications of the ACM, Vol. 53 Issue 2, pp. 66-75, Feb. 2010.

[4] R. Chang, A. Podgurski, J. Yang, "Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software," in Proceedings of the 2007 International Symposium on Software Testing and Analysis, London, UK, pp. 163-173. 2007.

[5] R. Chang, A. Podgurski, J. Yang, "Discovering Neglected Conditions in Software by Mining Dependence Graphs," in IEEE Transactions on Software Engineering, IEEE Computer Society. 2007.

[6] Coverity: http://www.coverity.com/

[7] Coverity Extend: http://www.coverity.com/products/static-analysis-extend.html

[8] C. Csallner and Y. Smaragdakis, "Check'n Crash: Combining Static Checking and Testing," in Proceedings of 27th International Conference on Software Engineering, ACM. 2005.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. "Bugs as Deviant Behavior: A General Approach to Inferring Errors in System Code," in Proceedings of 18th ACM Symp. on Operating Systems Principles, Banff, Canada, pp. 57-72. 2001.

[10] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and its Use in Optimization," ACM Trans. Prog. Lang. Syst., vol. 9, pp. 319-349. 1987.

[11] Grammatech, CodeSurfer, www.grammatech.com/products/codesurfer/overview.html.

[12] Guo, P. J. and Engler, D. 2009. Linux kernel developer responses to static analysis bug reports. Proceedings of the 2009 USENIX Annual Technical Conference, pp. 285‑292.

[13] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," ACM Trans. Program. Lang. Syst. 1s2, 1, pp. 26-60. Jan. 1990.

[14] Klocwork: http://www1.klocwork.com

[15] Klocwork C/C++ Path API Reference: http://www1.klocwork.com/products/documentation/barracuda/images/b/b1/Klocwork_C_Cxx_Path_API_Reference.pdf

[16] Klocwork detected C/C++ Issues: http://www1.klocwork.com/products/documentation/Insight-9.1/Detected_C/C%2B%2B_Issues

[17] Klocwork KAST and Path checkers: http://www1.klocwork.com/products/documentation/Insight-9.1/Writing_custom_C/C%2B%2B_checkers

[18] Klocwork path checker tutorials: http://www1.klocwork.com/products/documentation/Insight-9.1/Creating_C/C%2B%2B_Path_checkers

[19] Klocwork Support: https://my.klocwork.com/

[20] R. Krishnan, M. Nadworny and N. Bharill, "Static Analysis Tools for Security Checking in Code at Motorola," ACM SIGAda Ada Letters. 2008.

[21] Z. Li, and Y. Chou, "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code", ESEC-FSE '05, Lisbon, Portugal, pp.306-315. Sept 2005.

[22] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories", The joint meeting of European Softw. Eng. and ACM SIGSOFT Symp. Found. of Softw. Eng., pp. 296-305.

[23] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. "Muvi: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs," SOSP07, Stevenson, Washington, USA, pp. 103-116. Oct. 2007.

[24] G. J. Meyers. "The Art of Software Testing," Wiley, 1979.

[25] N. Nagappan and T. Ball. "Static analysis tools as early indicators of pre-release defect density," Proceedings of the 27th international conference on Software engineering, pp. 580–586. 2005.

[26] G. M. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt and P. Balachandran, "Making detect-finding tools work for you," in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Cape Town, South Africa, pp. 99-108. 2010.

[27] K. Y. Phang, J. S. Foster, M. Hicks, and V. Sazawak, "Path projection for user-centered static analysis tools," in PASTE'08: Proceedings of the 8th ACM workshop on Program analysis for software tools and engineering. 2008.

[28] PC-lint. http://www.gimpel.com/html/pcl.htm

[29] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach," in 30th International Conference on Software Engineering, Leipzig, Germany, pp. 341-350. May 2008,

[30] S. Shoham, E. Yahav, S. Fink and M. Pistoia, "Static Specification Mining using Automata-based Abstractions," in Proceedings of ISSTA, pp. 174-184. 2007.

[31] B. Sun, A. Podgurski, S. Ray. "Improving the Precision of Dependence-Based Defect Mining by Supervised Learning of Rule and Violation Graphs," in Preceedings of 21th International Symposium on Software Reliability Engineering, San Hose, CA, USA, pp. 1-10. Nov. 2010.

[32] S. Thummalapenta and T. Xie, "Mining Exception-Handling Rules as Conditional Association Rules," The 31th ICSE, Vancouver, Canada, pp. 496-506. May 2009.

[33] S. Thummalapenta and T. Xie, "Mining alternative patterns for detecting neglected conditions," in Proceedings of 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), pp. 283–294. 2009.

[34] A. Wasylkowski, A. Zellerand and C. Lindig, "Detecting Object Usage Anomalies," Proceedings of ESEC/FSE, Dubrovnik, Croatia, pp. 35-44. 2007.

[35] Web Demo of Automatic P2C Converter: http://selserver.case.edu:8080/autochecker/index.htm

[36] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "On the Value of Static Analysis for Fault Detection in Software," in IEEE Transactions on Software Engineering, vol. 32, no. 4, pp. 240-253. 2006.