Automating Discovery of Software Tuning Parameters

Nevon Brake, James R. Cordy
Queen's University
Kingston, Ontario, Canada
{brake, cordy}@cs.queensu.ca

Elizabeth Dancy, Marin Litoiu, Valentina Popescu
IBM Canada
Markham, Ontario, Canada
{lizdancy, marin, popescu}@ca.ibm.com

ABSTRACT

Software Tuning Panels for Autonomic Control (STAC) is a project to assist in the integration of existing software into autonomic frameworks. It works by identifying tuning parameters and rearchitecting to expose them as a separate control panel module. The project poses three distinct research challenges: automating the identification of tuning parameters, rearchitecting to centralize and expose them, and combining these two capabilities to facilitate the integration of existing software into autonomic frameworks. Our previous work focused on the second problem, automating the rearchitecture to expose and isolate tuning parameters. In this paper we concentrate on the first problem, automating the identification of tuning parameters. We begin with an empirical study of documented tuning parameters in a number of open source applications. From our observations of these known tuning parameters, we create a catalogue of different kinds and organize them into a taxonomy. Finally, we characterize a member of the taxonomy as a source code pattern that is used to find similar tuning parameters. We report our experience in applying this methodology in the context of a large, open source Java^{TM}system.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*software configuration management*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverese engineering and reengineering*; D.2.11 [**Software Engineering**]: Software Architectures; D.2.5 [**Software Engineering**]: Testing and Debugging—*monitors*

General Terms

Design, Management, Measurement

Keywords

autonomic computing, design recovery, rearchitecture, static analysis, tuning parameter

1. INTRODUCTION

Increased automation is essential to mitigate the complexity of managing software systems. Fundamental to achieving self-managing behavior is the ability to tune parameters that influence factors like performance and security. Traditionally, tuning is performed by human experts who have extensive knowledge of parameter names and value effects. However, manual tuning is costly and only contributes to the software maintenace problem. Software Tuning Panels for Autonomic Control (STAC) assists in the transition to more autonomic control by automatically identifying tuning parameters and rearchitecting software to isolate and expose them for monitoring and control by autonomic frameworks.

In our previous work [6] we describe an automated rearchitecture of legacy software to expose tuning parameters and isolate them to a separate monitoring and tuning module. In this paper we attack the related problem of automating the identification of tuning parameters using software analysis and pattern matching techniques. For this work we define a "tuning parameter" as a scalar field or property of a structured field in source code that measures or affects metrics like performance. A subset of these parameters are "tunable", meaning they are modifiable. The others, while not modifiable, are needed for making decisions about tuning. This definition simplifies the problem while still capturing the majority of practical tuning parameters in existing applications. We begin by analyzing the documented tuning parameters of a number of large open source systems to classify the kinds and characteristics of tuning parameters. We characterize each kind of tuning parameter as a pattern capturing the context and relationships with other parts of the code that distinguish it as a tuning parameter. We then demonstrate how both known and unknown tuning parameters can be identified in the code using static analysis and pattern matching techniques.

The paper is organized as follows. We begin with a brief overview of the STAC project and software analysis using design recovery. In Section 3 we present the approach taken to gather and analyze existing tuning parameters and how they are classified for pattern discovery. Section 4 demonstrates how the approach is applied to an open source Java application. We review related work in Section 5, and conclude with future work in Section 6.

2. BACKGROUND

The STAC project [6] was started in 2005 with the goal of automatically rearchitecting existing software for autonomic control by isolating and exposing their tuning parameters. This can be problematic as the tuning parameters are often scattered and sometimes hidden throughout source code. STAC works to overcome these issues by isolating tuning parameters into a single separate module. The result is maintainable, semantically equivalent software that provides localized access to tuning parameters.

Our previous work using a STAC prototype to isolate tuning parameters has shown promising results. Unfortunately, the prototype requires manual placement of markup around variable declarations of tuning parameters. This markup indicates which variables STAC should isolate. Manually searching for these variables and adding markup, even for a small application, can be tedious and error-prone. An automated identification mechanism is required to take full advantage of the STAC rearchitecture.

Design recovery [3] is a software static analysis method for analyzing source code by extracting an entity-relationship (ER) model of the entities (e.g., variables, methods, classes, etc.) and relationships (e.g., calls, comparisons, assignments, etc.) between the entities to form a software graph that can then be explored using relational algebra and graph pattern matching to discover deeper relationships. In this work we explore the application of design recovery to the problem of tuning parameter identification.

3. APPROACH

We study the documented tuning parameters in existing applications to understand how they are manifested in source code. The result is a taxonomy of different kinds. The tuning parameters are classified based on observed patterns in their use across multiple applications and application domains. These patterns, or idioms, are somewhat analogous to the design patterns [9] often employed in object-oriented software design. While design patterns address functional design issues, tuning parameter patterns address non-functional issues such as how to maintain a record of performance over time. The patterns are formalized and iteratively refined so that the necessary information can be automatically extracted from the source code and the patterns automatically detected.

3.1 Empirical Study

Thus far we have studied four applications (Table 1) that were chosen based on the following criteria:

1. Must be open source, so that we can study and report on source code patterns;

2. Must be implemented in the Java programming language, to be consistent with the STAC prototype;

3. Should be server-oriented, so that tuning parameters are realistic and relevant;

4. Should have industrial relevance, so that our results can have practical impact.

We focus on open source Java systems so that our results can be reported and easily attached to the STAC rearchitecture prototype, and server software, as opposed to client software, since tuning is more established and important in the server context.

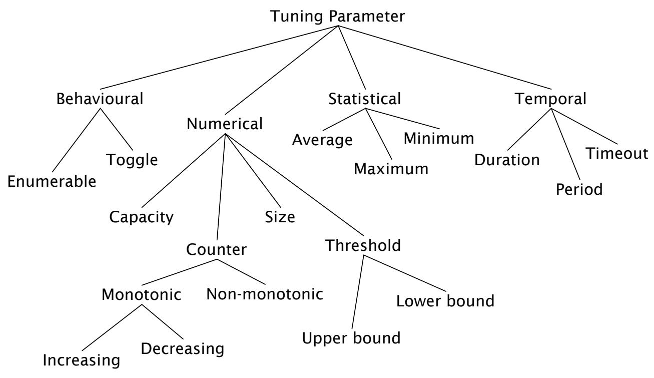

Figure 1: Tuning parameter classification by source code patterns of use.

The user documentation is consulted for each application as it commonly contains a performance tuning or management section. However, this section may only provide guidelines without mentioning the actual tuning parameters. This means falling back to the configuration section, where a list of parameters and their descriptions is often provided. This kind of search finds one type of tuning parameter–those that are used to influence the behaviour of the application. Another equally important type are those that provide observations about the application. It is impossible to know what to tune if you are not aware of how the application is performing. Fortunately, three of the applications in the study support Java Management Extensions (JMX) [18]. This support provides a set of standardized components that enable querying and modifying system properties. The properties specified by these components include many of the latter kind of tuning parameter.

3.2 Classification

Once a catalogue of tuning parameters is compiled from the documentation and JMX components of each application, the tuning parameters have to be mapped to corresponding fields in the source code. The mapping process involves a syntactical search of the source code for field declaration names that match the tuning parameter names. There are three possible outcomes for the search result:

- Exactly one matching declaration is found;
- More than one matching declaration is found;
- No matching declarations are found.

For the applications in this study, the first case is the most common and requires only a brief manual inspection to validate the match. The second case occurs less frequently than the first, but is more prevalent than the third. The additional matches in this case are often caused by data transfer classes, instances of which are data transfer objects. Data transfer objects act as containers in object-oriented languages like Java to transfer data either as a single argument to a method invocation or as a return value from a method invocation. The data transfer class declares a set of fields whose names are similar to the names of those fields for which the data is being transferred. More specifically,

Name	Version	Type	Open Source License	Lines of Code
Apache Tomcat	6.0.10	Web/Servlet	Apache 2.0	125,366
Apache Derby	10.2.2.0	Database	Apache 2.0	469,811
Jetty	6.1.3	Web/Servlet	Apache 2.0	35,723
Oracle Berkeley DB Java Edition	3.2.23	Database	OSI Approved	67,842

Table 1: Overview of the applications in the study.

for these cases, the value of the tuning parameter is being copied to a data transfer object to be passed to some other component such as a log manager. These matches are spurious and contribute nothing to the classification. Additional matches also result when tuning parameters with the same names exist for more than one component. These are not spurious and represent separate instances of a tuning parameter that happen to be identified by the same name. However, as with the first case, these can be easily validated through manual inspection. The final case is the most difficult since either the parameter no longer exists, it has been named differently in the source code, or it has been implemented indirectly using a hashed key/value pair. The latter case is confirmed by weakening the search to be lexical. The former cases are indistinguishable without additional clues and a more in-depth validation.

Having established a mapping between the tuning parameters and field declarations, all references to the identified fields are traced. These references are necessary to answer the question of how tuning parameters are used within the source code, or more to the point, whether patterns emerge that can be used to identify them. By observing the data transfers and comparisons involving the fields some similarities and differences become apparent. These differentiating characteristics are used to classify the different kinds of tuning parameters into a taxonomy (Figure 1):

Enumerable - Controls the behavior of the system through some finite set of states (e.g., thread priorities, algorithm selection).

Toggle - Enables or disables a behavior of the system (e.g., caching allowed).

Capacity - Limits how large (i.e., memory footprint) a resource can grow (e.g., buffer limits).

Counter - Measures how many times an event or action has occurred using discrete, constant increments (e.g., number of active connections).

Size - Measures how large a resource has grown (e.g., buffer usage).

Threshold - Limits the number of times an event or action can occur (e.g., prevent new connections).

Average - Measures the statistical average of some numerical or temporal parameter.

Maximum - Measures the statistical maximum of some numerical or temporal parameter.

Minimum - Measures the statistical minimum of some numerical or temporal parameter.

Duration - Measures the length of time some action or event has been occurring (e.g., system uptime)

Period - Controls the frequency with which an action or event is triggered (e.g., garbage collection).

Timeout - Limits the length of time an action has to complete before it will be interrupted (e.g., reading bytes from a network connection).

3.3 Design Model

In order to recognize usage patterns of tuning parameters, we need to know where and how data are transferred, and how data are compared. Traditional data-flow analyses can solve the where, but not the how. We instead use a static, reference-based analysis that is flow-insensitive, similar to that used by [7, 12]. With this type of analysis we are concerned with data transfers through assignments and method invocations. For assignments we are also interested in the particular operators used. Additionally, we need to determine which entities are compared to one another and using what operators.

Using observations made during the tuning parameter classification we construct a design model with which to reason about the relevant source code entities and their relationships (Figure 2). Source code entities are depicted as classes within an inheritance hierarchy, with relationships between them depicted as associations, or association classes. The source code entities are:

Expression - An Expression entity is an abstraction representing those entities that can be evaluated in some context, such as a field access or method invocation. Expressions can be subscripts of other expressions, can be compared to other expressions, can be returned by method invocations, can be actual parameters to a method invocation, and can appear on the right-hand side of assignments.

Variable - A Variable entity is an abstraction for variables such as fields of a class or interface, or local variables of a method. Variables can be all the things Expressions can be, and can also appear on the left-hand side of assignments.

Field - A Field entity represents a member variable of a class or interface whose scope does not extend beyond the class or interface in which it was declared. Fields can be all things Variables can be, and are also associated with the Type that declares them.

LocalVariable - A LocalVariable entity represents a variable whose scope does not extend beyond the method in which it was declared. Local variables can be all things Variables can be, and can also be formal parameters of method declarations.

Method - A Method entity represents a method of a class or interface. Methods can be all things Expressions can be, and can also override other methods.

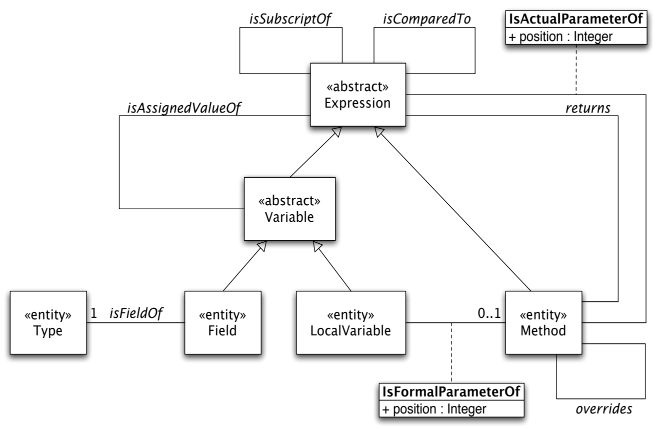

Figure 2: A model relating source code entities to one another to facilitate reasoning about data transfers and comparisons.

Type - A Type entity represents a class or interface.

Because our design model is similar to the reference-flow model used in design recovery [7], it can be populated using design recovery techniques on the source code to produce an instance graph representing the relationships between all of these entities in the program.

3.4 Design Recovery

In the design recovery step, the source code is parsed to generate a set of abstract syntax trees, one for each compilation unit. The abstract syntax trees are then traversed one-by-one to extract the data represented in the design model. For example, whenever a field declaration is encountered during the traversal, the association *isFieldOf* is generated between the field and the declaring type. Similarly, whenever a method declaration is encountered, the association *isFormalParameterOf* is generated between each of the formal parameters and their positions, and the declaring method. The extracted relationships are binary in nature and can therefore be thought of as directed graphs where the source code entities are nodes and the relationships are edges. Using this graph representation makes the relationships easier to visualize and facilitates both pattern building and matching.

Using the extracted information as-is to reason about the source code is difficult because of the indirection induced by the object-oriented paradigm. While analyzing data transfers through assignments is somewhat straightforward, the situation is sufficiently complicated when dealing with method invocations that pass parameters that may themselves be method invocations. All the while, each of these methods may be overridden by subclasses, and so on. Instead we make inferences about the information to generate more direct relationships. For example, from an assignment that has a method invocation on the right-hand side we can infer a direct assignment to the expression returned by the method. If the method is overridden, we also infer assignments to the expressions returned by the subclasses' methods. Each of these inferences generates new edges in our graph representation.

The result of the design recovery step is a complete graph representing the transitive flow of data references through the software. The inference step localizes relationships such that all interactions for each individual variable or field are directly attached to its entity node.

```
<?xml version="1.0"?>
<mbeans-descriptors>
  <mbean name="StandardManager"
    description="Standard implementation of..."
    domain="Catalina"
    group="Manager"
    type="org.apache.catalina.session.StandardManager">
      ...
      <attribute name="sessionCounter"
        description="Total number of sessions created..."
        type="int" />
      ...
      <attribute name="sessionMaxAliveTime"
        description="Longest time an expired session..."
        type="int" />
      ...
      <operation name="listSessionIds"
        description="Return the list of active session..."
        impact="ACTION"
        returnType="java.lang.String" />
      ...
  </mbean>
</mbeans-descriptors>
```

Figure 3: Excerpt from *mbeans-descriptors.xml* in the package `org.apache.catalina.session`

3.5 Patterns

The design graph representation provides a high-level view of the variables and fields of the system and how they are used. Because inference has localized all relationships, we can encode the pattern of use representing each kind of tuning parameter as a local subgraph pattern centered on the variable or field entity itself. For example, a field representing a Maximum parameter might have a subgraph pattern that requires assignments to a field to be guarded by comparisons for greater than or equal to.

By observing the subgraphs surrounding known parameters, we can infer and tune subgraph patterns representing each kind of tuning parameter. These subgraph patterns can then be used to identify new tuning parameters by searching program design graphs for instances of each pattern. In the remainder of the paper, we demonstrate this entire process, using Apache Tomcat as an example.

4. AN EXAMPLE: STATISTICAL MAXIMA IN APACHE TOMCAT

To demonstrate, we apply the methodology to Apache Tomcat [1], a web container implemented in Java that includes a standalone HTTP server. Apache Tomcat is used in production by hundreds of thousands of websites [13] and serves as the reference implementation for the JavaServer Pages (JSP) [17] and Java Servlet [19] technologies.

For its management infrastructure, Apache Tomcat includes an extensive set of JMX [18] components, or *managed beans*. Managed beans are Java objects that define management interfaces as a set of attributes and operations that follow well-known naming conventions and design patterns. While the application documentation does not explicitly mention each of the managed beans and their capabilities, there are files named *mbeans-descriptors.xml* in some

```
package org.apache.catalina.session;
...
public abstract class ManagerBase implements Manager,
    MBeanRegistration {
  ...
  /**
   * The longest time (in seconds) that an expired session
   * had been alive.
   */
  protected int sessionMaxAliveTime;
  ...
  // Number of sessions created by this manager
  protected int sessionCounter=0;
  ...
}
```

Figure 4: Fields that correspond to the managed bean attributes

of the Java source packages. These files provide detailed descriptions of the managed beans along with their attributes and operations. For example, Figure 3 is an excerpt from the descriptor for a managed bean named *StandardManager*.

Recall that we have restricted our definition of tuning parameters to scalar fields and scalar properties of structured fields. These correspond to the scalar attributes of the managed bean. Continuing with the *StandardManager* example, we can map the managed bean attributes to fields in the source code. There is a class named `StandardManager` that corresponds to the managed bean; however, the fields corresponding to the attributes are actually inherited from the parent class, `ManagerBase` (Figure 4). The following discussion, therefore, refers to `ManagerBase`. All references to the fields must be traced to determine a classification for each of the tuning parameters. We shall consider one of the fields from the example, `sessionMaxAliveTime`.

4.1 Building a Pattern

Based solely on its name, the field `sessionMaxAliveTime` might be an upper bound on the length of time a client session is permitted to exist. However, as indicated by the developer's comment, the field is actually a statistical maximum taken over the lengths of all expired sessions. While such natural language descriptions are undoubtedly helpful to a human, it is difficult to automate this kind of reasoning for all cases. Instead, we can use characteristics of the data transfers and comparisons involving the field to determine how the tuning parameter should be classified.

There are two references to the field in the class `ManagerBase` (Figure 5). One is an accessor method for retrieving the value of the field, and the other is a mutator for changing the value of the field. Following the chain of references from these methods leads us to the class `StandardSession` (Figure 6) where both the accessor method and the mutator method are referenced indirectly through the interface `Manager`, which is implemented by `ManagerBase`. By observing the relationship between the accessor and the mutator established by the conditional statement, it becomes apparent that the mutator is only ever invoked if the current value of the field `sessionMaxAliveTime` has been exceeded by the value of the local variable `timeAlive`. This is precisely the semantics of a statistical maximum algorithm.

This example illustrates the process of manually classifying a single known tuning parameter. In practice, we use the knowledge gained through classifying many parameters to automatically discover previously unknown tuning parameters. We use a formal description of the relationships between the field, and its data transfers and comparisons. These relationships can then be automatically extracted from the source code and analyzed for the pattern.

```
public abstract class ManagerBase implements Manager,
    MBeanRegistration {
  ...
  public int getSessionMaxAliveTime() {
    return sessionMaxAliveTime;
  }

  public void setSessionMaxAliveTime(
      int sessionMaxAliveTime) {
    this.sessionMaxAliveTime = sessionMaxAliveTime;
  }
  ...
}
```

Figure 5: References to the field `sessionMaxAliveTime` in the class `ManagerBase`

```
package org.apache.catalina.session;
...
public class StandardSession
    implements HttpSession, Session, Serializable {
  ...
  protected transient Manager manager = null;
  ...
  public void expire(boolean notify) {
    ...
    long timeNow = System.currentTimeMillis();
    int timeAlive = (int) ((timeNow - creationTime)/1000);
    synchronized (manager) {
      if (timeAlive > manager.getSessionMaxAliveTime()) {
        manager.setSessionMaxAliveTime(timeAlive);
      }
      ...
  }
  ...
}
```

Figure 6: References to the accessor and mutator methods of the class `ManagerBase` in the class `StandardSession`

Our formal description is taken from the design model described in Section 3.3. The source code is parsed into abstract syntax trees that are traversed to extract instances of the relationships defined in the design model. These relationship instances form a directed graph, where the nodes are source code entities (e.g., variables, types) and the edges are the relationships (e.g., assignment, inheritance) between the entities (Figure 7):

(1.1) The field `sessionMaxAliveTime` is declared by the class `ManagerBase`;

(1.2) The field `sessionMaxAliveTime` is assigned the value of a local variable of the same name by the method `setSessionMaxAliveTime()` (i.e., mutator method);

(1.3) The local variable `sessionMaxAliveTime` is a formal parameter of the method `setSessionMaxAliveTime()` at position 0 (i.e., the first parameter);

(1.4) The formal parameter at position 0 of the method `setSessionMaxAliveTime()` in the class `ManagerBase` overrides the formal parameter at position 0 of a method with the same name declared by the `Manager` interface;

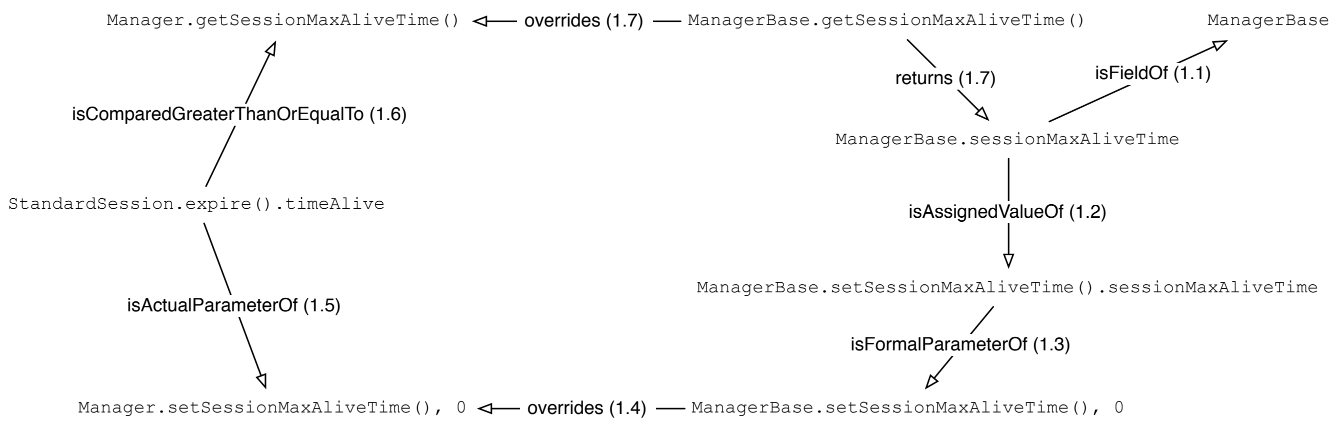
Manager.getSessionMaxAliveTime()
overrides (1.7)
ManagerBase.getSessionMaxAliveTime()
ManagerBase
returns (1.7)
isFieldOf (1.1)
isComparedGreaterThanOrEqualTo (1.6)
ManagerBase.sessionMaxAliveTime
StandardSession.expire().timeAlive
isAssignedValueOf (1.2)
ManagerBase.setSessionMaxAliveTime().sessionMaxAliveTime
isActualParameterOf (1.5)
isFormalParameterOf (1.3)
Manager.setSessionMaxAliveTime(), 0
overrides (1.4)
ManagerBase.setSessionMaxAliveTime(), 0

Figure 7: Graph of relationships for the field `sessionMaxAliveTime`. Nodes in the graph use the qualified name of the entity, so the field `sessionMaxAliveTime` is denoted `ManagerBase.sessionMaxAliveTime`.

(1.5) The value of the local variable `timeAlive` is passed as a parameter to the method `setSessionMaxAliveTime()` of the interface `Manager` by the method `expire()` in the class `StandardSession`;

(1.6) The value of the local variable `timeAlive` is compared to the return value of the method `getSessionMax-AliveTime()`;

(1.7) The method `getSessionMaxAliveTime()` in the class `ManagerBase` overrides a method of the same name in the interface `Manager`;

(1.8) The method `getSessionMaxAliveTime()` of the class `ManagerBase` returns the value of the field `session-MaxAliveTime` (i.e., accessor method).

Unfortunately, the object-oriented paradigm induces numerous levels of indirection. This makes it difficult to detect the statistical maximum relationship between the field `sessionMaxAliveTime` and the local variable `timeAlive`. To simplify the task of detecting the pattern, the graph is strengthened using relationships inferred through relational algebra (Figure 8):

(2.1) The method `getSessionMaxAliveTime()` of the interface `Manager` could return the value of the field `sessionMaxAliveTime` since the class `ManagerBase` has overridden the method;

(2.2) The value of the local variable `timeAlive` could be compared to the value of the field `sessionMaxAlive-Time` since it is compared to the return value of the method `getSessionMaxAliveTime()` which, according to the previous inference, returns the value of the field;

(2.3) The local variable `timeAlive` could be an actual parameter of the method `setSessionMaxAliveTime()` of the class `ManagerBase` since it is an actual parameter of the interface method overridden by `ManagerBase`;

(2.4) The local variable `sessionMaxAliveTime` could be assigned the value of the local variable `timeAlive` since `sessionMaxAliveTime` is a formal parameter to which, according to the previous inference, the value of `timeAlive` could be an actual parameter;

(2.5) The field `sessionMaxAliveTime` could be assigned the value of the local variable `timeAlive` since it is assigned the value of the local variable `sessionMaxAlive-Time` which, according to the previous inference, could be assigned the value of `timeAlive`.

The strengthened graph now has direct relationships between the field `sessionMaxAliveTime` and the local variable `timeAlive`. By observing that the field is only ever assigned a value (2.5) to which it has been compared as being greater than or equal to (2.2), the statistical maximum pattern has been explicated. However, another formalism is still needed to phrase such an observation so that the pattern detection process can also be automated. The problem is isomorphic to finding subgraphs that: must contain certain edges, may contain certain edges, or must not contain certain edges. These kinds of graph queries can be represented in the visual query language GraphLog [5]. Intuitively, the query resembles the subgraphs of interest. Nodes in the query serve as variables to which nodes in the graph are bound based on whether their edges match the corresponding edge constraints in the query.

The statistical maximum query (Figure 9) consists of three constraints:

(1) The field **F** *must* be declared by some type **T**;

(2) The field **F** *must* be assigned some expression **E**;

(3) The expression **E** *must* be compared to the field **F**;

This query results in **T** being bound to `ManagerBase`, **F** being bound to `ManagerBase.sessionMaxAliveTime`, and **E** being bound to `StandardSession.expire().timeAlive`. The thicker edge in the query represents a new relationship between **F** and **E** that establishes **F** as the statistical maximum of **E**. Querying the graph of the entire application results in similar bindings for all other subgraphs matching this one. Each suitable binding for **F** represents a statistical maximum tuning parameter.

4.2 Using the Pattern

Now that the statistical maximum pattern has been constructed and validated against the tuning parameter on which

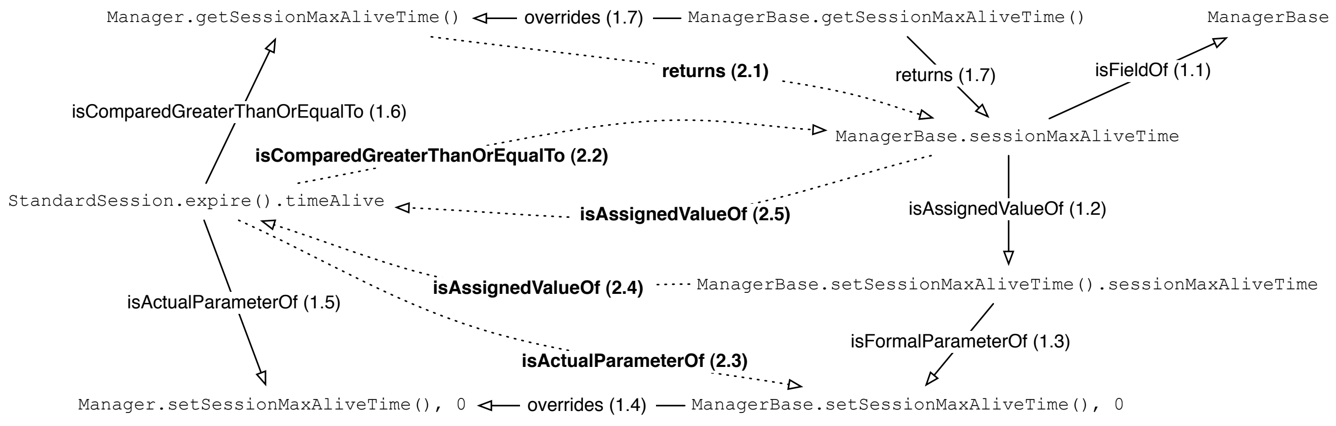
Manager.getSessionMaxAliveTime()
overrides (1.7)
ManagerBase.getSessionMaxAliveTime()
ManagerBase
returns (2.1)
returns (1.7)
isFieldOf (1.1)
isComparedGreaterThanOrEqualTo (1.6)
isComparedGreaterThanOrEqualTo (2.2)
ManagerBase.sessionMaxAliveTime
StandardSession.expire().timeAlive
isAssignedValueOf (2.5)
isAssignedValueOf (1.2)
isAssignedValueOf (2.4)
ManagerBase.setSessionMaxAliveTime().sessionMaxAliveTime
isActualParameterOf (1.5)
isFormalParameterOf (1.3)
isActualParameterOf (2.3)
Manager.setSessionMaxAliveTime(), 0
overrides (1.4)
ManagerBase.setSessionMaxAliveTime(), 0

Figure 8: Graph of relationships with edges generated by inference. New edges are denoted with dashed lines and labeled in bold typeface.

it was based, it is time to automatically discover other, previously unknown instances of the pattern. Applying the query to the graphs extracted from the Tomcat source code yields numerous results. Many of these are already documented in the managed bean descriptors. But there are also some that are undocumented. An example of one such parameter is `deadMaxTime` in the class `RequestGroupInfo` (Figure 10).

Each time a request processor is removed by the method `removeRequestProcessor`, the field `deadMaxTime` is updated with the maximum time the request processor took to process a request. But only if that particular request processor took longer than any of its predecessors. In other words, the field `deadMaxTime` is the statistical maximum over all requests processed by all request processors in that particular group. It should be noted that the conditional expression in this example is written differently from that of the example in the previous section. The syntax used here has the field on the left-hand side of a less-than operator as opposed to the right-hand side of a greater-than operator. These syntactical variances do not affect the pattern since they are made to be semantically equivalent by the analysis before the query is applied.

In this paper we only have room for a short demonstration of how our method works with one particular pattern. In practice we have inferred and applied patterns for a number of kinds of tuning parameters in our taxonomy, and we are in the process of applying these patterns to find tuning parameters in several other open source software systems in addition to the four used to create the taxonomy.

5. RELATED WORK

The purpose of our work is to find and expose potential tuning parameters for monitoring and autonomic control. To the best of our knowledge, software tuning parameter identification has not been previously studied.

Our prototype implementation uses Eclipse's JDT representation [8] and custom code to implement our design graphs and pattern matching. While convenient for working with the applications in the study, our prototype might not scale well, nor does it handle languages other than Java. A production implementation of our technique might use more mature tools of the software design recovery community such as Rigi [16] to extract and infer model graphs for multiple languages, and CrocoPat [2] or Grok [10] to implement pattern matching on them. Other techniques from the design recovery community might also be applicable to the discovery problem. One can imagine for example casting tuning parameter discovery as a concept location [15] or aspect mining [4] problem.

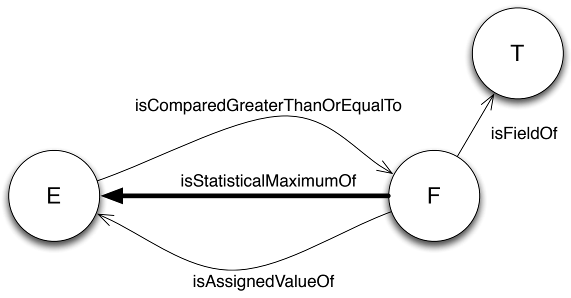

Figure 9: A GraphLog query for the statistical maximum pattern.

The results of our work can be used with management frameworks such as JMX [18], ATMA [11], or WSDM [14] to implement autonomic controllers. A JMX (Java Management Extensions) specification defines an architecture, design patterns, API, and services for management and monitoring of applications written in Java. New tuning parameters discovered by our process can be encoded as JMX managed beans. Once rearchitected by STAC, tuning parameters could also be easily attached to ATMA or WSDM-based control frameworks.

6. CONCLUSION

We have explored the problem of how to automate the discovery of software tuning parameters and motivated our work in the context of a broader autonomic project, Software Tuning Panels for Autonomic Control. We provided results from our empirical study of documented tuning parameters

```
package org.apache.coyote;
...
public class RequestGroupInfo {
  ...
  private long deadMaxTime = 0;
  ...
  public synchronized void removeRequestProcessor(
      RequestInfo rp) {
    if( rp != null ) {
      if( deadMaxTime < rp.getMaxTime() )
        deadMaxTime = rp.getMaxTime();
      ...
      }
  }
  ...
}
```

Figure 10: Declaration and references for the field `deadMaxTime` in the class `RequestGroupInfo`

and described how they were classified into a taxonomy. In addition, we characterized a known tuning parameter as a source code pattern and showed how it could be used to find other, undocumented tuning parameters. This was demonstrated using an example from an industrially relevant, open source application, Apache Tomcat.

We are presently in the process of evaluating the effectiveness of the approach more formally using three applications not previously studied: a web server, a database server and a messaging framework. This evaluation will measure the accuracy of the technique when applied to application domains that are both similar and different from those on which it was trained.

In the future we plan to refine the tuning parameter taxonomy based on expert feedback and study of a broader range of application domains. We also plan to study an orthogonal tuning parameter classification scheme based on the kinds of resources the tuning parameters influence. Finally, we hope to integrate with management frameworks based on standards such as WSDM.

7. ACKNOWLEDGMENTS

This work was supported by the IBM Toronto Centre for Advanced Studies (CAS), the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Ontario Graduate Scholarship (OGS) program.

8. TRADEMARKS

9. REFERENCES

[1] Apache Software Foundation. *Apache Tomcat Documentation, Version 6.0*, 2006.

[2] D. Beyer. Relational programming with CrocoPat. In *International Conference on Software Engineering*, pages 807–810, 2006.

[3] T. J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, 1989.

[4] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. A qualitative comparison of three aspect mining techniques. In *International Workshop on Program Comprehension (IWPC)*, pages 13–22, 2005.

[5] M. P. Consens and A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *9th ACM Symposium on Principles of Database Systems (PODS)*, pages 404–416, 1990.

[6] E. Dancy and J. R. Cordy. STAC: Software tuning panels for autonomic control. In *CASCON'06, 16th IBM Centre for Advanced Studies International Conference on Computer Science and Software Engineering*, pages 146–160, 2006.

[7] T. R. Dean, J. R. Cordy, K. A. Schneider, and A. J. Malton. Using design recovery techniques to transform legacy systems. In *International Conference on Software Maintenance (ICSM)*, pages 622–631, 2001.

[8] Eclipse Foundation. *Eclipse Java Development Tools (JDT) Subproject*, 2008.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[10] R. Holt. Binary relational algebra applied to software architecture, 1996.

[11] IBM AlphaWorks. *IBM Autonomic Task Manager for Administrators (ATMA)*, 2005.

[12] K. Kawabe, A. Matsuo, S. Uehara, and A. Ogawa. Variable classification technique and application to the Year 2000 problem. In *2nd Euromicro Conference on Software Maintenance and Reengineering*, pages 44–51, 1998.

[13] Netcraft. *Netcraft Web Server Survey*, December 2007.

[14] OASIS. *Web Services Distributed Management: Management of Web Services (WSDM-MOWS), version 1.1*, August 2006.

[15] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *International Conference on Program Comprehension*, pages 37–48, 2007.

[16] M.-A. D. Storey, K. Wong, and H. A. Müller. Rigi: A visualization environment for reverse engineering. In *International Conference on Software Engineering*, pages 606–607, 1997.

[17] Sun Microsystems, Inc. *JavaServer Pages Technology - Documentation, version 2.0*, 2004.

[18] Sun Microsystems, Inc. *Java Management Extensions (JMX) Specification, version 1.4*, November 2006.

[19] Sun Microsystems, Inc. *JSR-000154 Java Servlet 2.5 Specification*, 2006.