STAC: Software Tuning Panels For Autonomic Control

Elizabeth Dancy and James R. Cordy

School of Computing,
Queen's University,
Kingston, Ontario, Canada
{dancy, cordy}@cs.queensu.ca

Abstract

One aspect of autonomic computing is the ability to identify, separate and automatically tune parameters related to performance, security, robustness and other properties of a software system. Often the response to events affecting these properties consists of adjusting tuneable system parameters such as table sizes, time-out limits, restart checks and so on. In many ways these tuneable parameters correspond to the switches and potentiometers on the control panel of many hardware devices. While modern software systems designed for autonomic control may make these parameters easily accessible, in legacy systems they are often scattered or deeply hidden in the software source.

In this paper we introduce Software Tuning Panels for Autonomic Control (STAC), a system for automatically re-architecting legacy software systems to facilitate autonomic control. STAC works to isolate tuneable system parameters into one visible area of a system, producing a resulting architecture that can be used in conjunction with an autonomic controller for self-maintenance and tuning. A proof-of-concept implementation of STAC using source transformation is presented along with its application to the automatic re-architecting of two open source Java programs. Use of the new architecture in monitoring and autonomic control is demonstrated on these examples.

1 Introduction

The time is approaching when the rate of software development so outweighs human expertise that there may be a software maintenance crisis. The domain of autonomic computing aims to proactively combat this problem by working towards the production of self-controlling and self-maintaining systems. However, it is futile to attempt to completely replace all existing software with autonomic equivalents because this just leaves us in a different race against time. It therefore seems pertinent to look at ways to automate the conversion from legacy systems to autonomic systems. Facilitating self-maintenance and self-tuning requires access to numerous tuneable system parameters. At present, these parameters are often scattered throughout source code and sometimes hidden, forcing each access to involve a costly search through the code.

In order to automate the re-architecture of programs so that the tuneable parameters are accessible without a search, we propose STAC, Software Tuning Panels for Autonomic Control, a system to automatically re-architect legacy source code with respect to marked-up tuneable parameters. The result is a system with identical functionality that provides the

maintainer, autonomic or otherwise, with localized access to these tuneable parameters for instrumentation, tuning and maintenance purposes. In STAC, tuneable parameters are limited to scalar types in order to bound the scope of the project. However, in practice tuneable parameters could be any system variables.

STAC has two main steps. First, tuneable parameters of interest are identified and references to them traced throughout the source code. Second, a new program module is generated that encapsulates all actions on the parameters during program execution. In STAC this new module is called the *Control Panel*.

The remainder of this paper is organized as follows. Section 2 describes the general approach and implementation of STAC. Section 3 provides examples of STAC re-architectures on two real world open source applications. Section 4 demonstrates three applications of the STAC Control Panels created for these systems. Section 5 provides a synopsis of related work and Section 6 analyzes the results of STAC and suggests future work in the area.

2 Approach

The prototype STAC re-architecture process for Java programs consists of a Java Front End in combination with a series of source transformations written in the TXL programming langauge [6]. These language choices were deliberate but not necessary as STAC could be implemented using other languages.

To begin the STAC implementation, the original source code is normalized for transformation. To do so, the original Java software package is first sent through a Front End File Filter, where it is sorted down into only source code files and organized into one contiguous text file to prepare the input for TXL transformation. Special XML tag delimiters separate each source file and other tags denote where each original source folder begins and ends so that the package can be re-created following re-architecture.

The merged source text file is run through a series of transformations which are responsible for the actual re-architecting of the source code and the generation of the Control Panel module housing the tuneable parameters of interest. Finally, the transformed contiguous source file is passed back into the Front End to be sorted into its original file and package structure, with the addition of the Control Panel. Figure 1 outlines this process graphically.

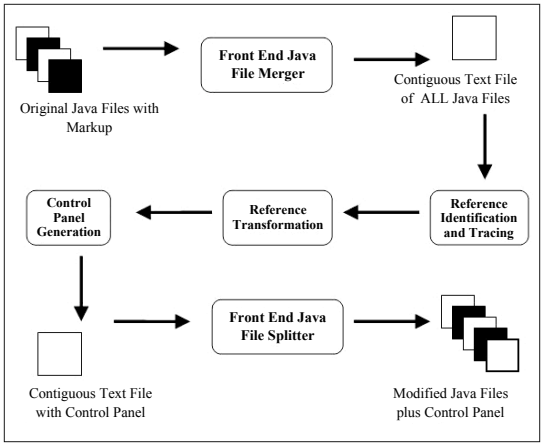

Figure 1: Overview of the STAC Process

The most challenging technical parts in Figure 1 are the TXL transformations which identify and transform the tuneable parameter references, as well as those transformations which create the Control Panel module. Details of these two main parts are explained below. It is pertinent to note that before this stage, a series of unique renaming transformations are applied to distinguish different variables with the same name.

In the STAC prototype the user must manually mark up the declaration of each tuneable parameter of interest directly in the source code using XML tags. After this step, the remainder of the transformation process is automated and hidden from the user.

The main architectural changes to a system are the addition of a new program module, the Control Panel, and the redirection of tuneable parameter references to the Control Panel. Figure 2 is a conceptual view of how a system is re-architected using STAC. The left-hand side shows the tuneable parameter references (represented by hollow circles) before the transformation. After the transformation, these references refer to the newly generated Control Panel module shown at the bottom.

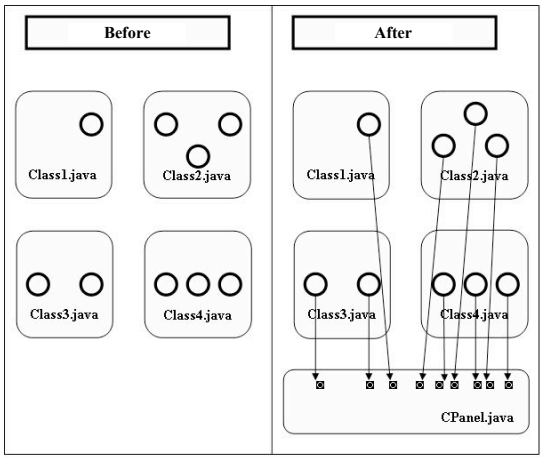

Figure 2: STAC Re-Architecting

2.1 Control Panel Generation

Creating the Control Panel class requires tracking of the tuneable parameters by searching for the XML markup around each tuneable parameter's declaration. Each marked up parameter is added to the Control Panel, one at a time. Each variable of interest is represented in the Control Panel as its own class, named for the tuneable parameter it represents. The left-hand side of Figure 3 shows an original Java class with a variable *numUsers* which has been marked up. The right-hand side of Figure 3 shows the resulting Control Panel for *numUsers*.

Not all Control Panel classes are as simple as that in Figure 3. In a typical program, several tuneable parameters would be marked up and each of those may have many references to it. Due to Java inheritance, tuneable parameters that are created in subclasses of other classes containing tuneable parameters may have other objects that refer to them. The Control Panel must create a separate reference class for each copy of a variable which is explicitly created in the code to accommodate for this inheritance property.

For example, in the class *InteractiveSystem* of Figure 3 we have marked up the global variable *numUsers* as a tuneable parameter before the transformation. If more than one instantiation of *InteractiveSystem* were created in the program, then the Control Panel must

```
public class
InteractiveSystem
{
<control_param>
      int numUsers =999;
</control_param>

public void startUp()
{
      break;
}
}//end class
```

```
public class Cpanel
{
 public static class numUsers
 {
   public static int numUsers;


   public static int create()
   {
      return numUsers;
   }
   public static int set (int value)
   {
      numUsers = value;
      return numUsers;
   }
   public static int get ()
   {
        return numUsers;
   }
  }//end class numUsers

}// end class Cpanel
```

Figure 3: *NumUsers* Markup and Control Panel Creation

contain a new copy of *numUsers* for each instance. Other extensions to our Control Panel generation handle other Java scenarios which would allow another pointer to the same *numUsers* variable. Figure 4 shows an example generated Control Panel with nesting to handle the situation where there is more than one copy of a tuneable parameter. In this case, an instance of *InteractiveSystem* named *InteractiveSystemInst* is created in the Java program and needs to have its own copy of *numUsers* in the Control Panel. If a second instance of *InteractiveSystem* were created, say *InterActiveSecondInst*, the Control Panel would add another class where *InterActiveSecondInst* would have its own *numUsers* class nested inside of it, and so on.

2.2 Reference Changing

While the above Control Panel serves its purpose, thus far the variables in the Control Panel classes are completely separate from those that were originally in the code. They must be linked with the code in order to serve any real use. The second major step in the TXL transformation involves changing all the original program references to the tuneable parameter variables to point to the Control Panel. The biggest challenge of this step is ensuring semantic equivalence.

The TXL transformation must therefore create a suitable, semantically equivalent reference, which is also syntactically valid, to insert in the place of all original references to the

```
public static class Cpanel
{
      public static class numUsers
      {
        public static int numUsers;
        public static int create()
        {
            return numUsers;
        }
        public static int set (int value)
        {
          numUsers = value;
          return numUsers;
        }
        public static int get ()
        {
          return numUsers;
        }
      }

    public static class InterActiveSystemInst
    {
      public static class numUsers
      {
        public static int numUsers;
        public static int create()
        {
            return numUsers;
        }
        public static int set (int value)
        {
          numUsers = value;
          return numUsers;
        }
        public static int get ()
        {
          return numUsers;
        }
      }
    }// end class InterActiveSystemInst
}// end Cpanel
```

Figure 4: Control Panel with Nesting

marked up parameters. For example, a constructor which is marked up must be replaced with a call to the constructor method in the Control Panel. Even though references may be of the same kind (such as constructors, assignment statements or variable calls), they may take on several different forms. For example, a constructor in Java can be either a constructor with an initialized variable or one without an initial value given. These can be further broken down into those which list several variable names in the same constructor, and those which declare only one. Each kind of reference may therefore require several different transformation rules to handle the different cases.

The left-hand side of Figure 5 shows two kinds of constructors marked up for transformation by the STAC tool. The right-hand side shows how they have been changed by the STAC transformation in the source code to refer to the Control Panel generated by STAC.

No matter how complex the reference, the TXL transformations must have rules which can match and transform all forms of references (within the scope of this project) to point to

Original Class

```
public class Example2{
 <control_param>
 int activeUsers;
 </control_param>

 <control_param>
 int idleUsers =999;
 </control_param>

 public void startUp()
 {
       break;
 }
}//end class
```

Class After TXL Transform

```
public class Example2{
 int activeUsers =
 Cpanel.activeUsers.create();

 int idleUsers =
 Cpanel.idleUsers.set (999);

 public void startUp()
 {
       break;
 }
}//end class
```

Figure 5: Reference Changing

the Control Panel. Furthermore, the transformation must be careful not to add any unnecessary source code, to ensure that the code remains maintainable. The next section explores this complicated issue through the use of two real-world examples.

3 Examples

Once a system has been transformed by STAC, a user (or autonomic controller) has complete control over the tuneable parameters that have been marked up. This section details the entire re-architecting process through the use of a two real-world open source examples. Each original architecture is first outlined and then a STAC transformation is stepped through.

In addition to the examples detailed below, the STAC transformation has been run on other Java programs with varying architectures. Tests were run on STAC throughout the process including regression tests and rule-based functional tests on the TXL code to ensure that a variety of cases can be handled. Tests of functionality were carried out on the Java source code both before and after the transformation in order to check semantics preservation.

Because TXL rules are encoded as a set of formal re-write rules, a more formal validation of semantics preservation could also be carried out independently using an an inductive proof. However, this would be time-consuming and is well beyond the scope of the STAC proof of concept presented in this paper.

3.1 JHotDraw

JHotDraw [4] is an open source graphical user interface Java application that serves as a tool for creating graphical editing programs. One can also use it as a graphics editor itself. The source code for JHotDraw is openly available and comes packaged in several layers. In total, the JHotDraw version used for STAC contains eighteen Java source code packages, including class files, jar files and required images. It contains approximately 32,500 (total) lines and 16,000 logical lines of code after it has been filtered in preparation for the STAC transformation. JHotDraw is chosen as a proof of concept system for STAC because it is a widely used medium-sized Java application which is also available as open source.

3.1.1 Tuneable Parameters

Once JHotDraw has been selected and analyzed for its main properties, the discovery of tuneable parameters can begin. In the present prototype this identification is performed manually. Once found, the constructor of each tuneable parameter is marked up by hand. In the STAC proof-of-concept prototype, candidates for parameters of interest must meet the condition that they are either stored as a scalar type or in a Vector.

Since the class *DrawApplication.java* is a class included in every application in JHotDraw, we start there. One possible candidate is the *int winCount*, declared as a static global variable, representing the number of open windows in the application at one time, and initialized to zero in *DrawApplication*. This will serve as the first example because *winCount* is contained in one class and keeps track of several different components during runtime. The declaration for *winCount* is shown in Figure 6 and its references are also highlighted.

3.1.2 Transformation

The XML tags in Figure 6 mark *winCount* as a tuneable parameter. After the transformation, the class *DrawApplication.java* is changed in the following ways: it imports the Control Panel (stored in a new package in the same home directory as the entire source package),

```
public class DrawApplication extends JFrame
       implements DrawingEditor, PaletteListener, VersionRequester
{
      private Tool      fTool;
      private Iconkit   fIconkit;

      private JTextField      fStatusLine;
      private DrawingView     fView;
      private ToolButton            fDefaultToolButton;
      private ToolButton            fSelectedToolButton;

      private String                fApplicationName;
      private StorageFormatManager  fStorageFormatManager;
      private UndoManager           myUndoManager;
      protected static String       fgUntitled = "untitled";
      private final EventListenerList listenerList = new
      EventListenerList();
      private static final String   fgDrawPath ="/CH/ifa/draw/";
      public static final String    IMAGES = fgDrawPath + 'images/";
      <control_param>
      protected static int          winCount = 0;
      </control_param>

      public static final int             FILE_MENU = 0;
      /* The index of the edit menu in the menu bar.       */
      public static final int             EDIT_MENU = 1;
      /* The index of the alignment menu in the menu bar.  */
      public static final int             ALIGNMENT_MENU = 2;
      /* The index of the attributes menu in the menu bar. */
      public static final int             ATTRIBUTES_MENU = 3;

      /**
       * Constructs a drawing window with a default title.
       */
      public DrawApplication() {
            this("JHotDraw");
      }

      public DrawApplication(String title)
      {
            super(title);
            setApplicationName(title);
            winCount++;
      }
      public void exit()
      {
            destroy();
            setVisible(false);      // hide the JFrame
            dispose();   // tell windowing system to free resources
            winCount--;
            if (winCount == 0)
            {
                  System.exit(0);
            }
      } . . .
}// end class DrawApplication
```

Figure 6: *WinCount* Declaration and References

and all the references to *winCount*, including the constructor, have been changed to refer to the Control Panel (Cpanel) class. In addition, the XML markup has been removed. Figure 7 shows this change, while Figure 8 shows the new Cpanel class. Notice the constructor, accessor and mutator methods for *winCount* within the class *Cpanel.winCount*. These are generated for every tuneable parameter which is marked up, giving a maintainer the power to modify and create variables within the Control Panel.

3.2 Babylon Chat

Babylon Chat [1] is an open source Java instant messaging system which can be used for internet conferences or for group meetings on

```
import CH.ifa.draw.Control_Panel.*;

public class DrawApplication extends JFrame implements DrawingEditor,
PaletteListener, VersionRequester {
  . . .
   protected static int winCount = Cpanel.winCount.set (0);
    public DrawApplication () {
        this ("JHotDraw");
    }

    public DrawApplication (String title) {
        super (title);
        setApplicationName (title);
        Cpanel.winCount.set (Cpanel.winCount.get() + 1);
    }

    public void exit ()
    {
          destroy ();
          setVisible (false);
          // hide the JFrame
          dispose ();
           // tell windowing system to free resources
           Cpanel.winCount.set (winCount - 1);
           if (Cpanel.winCount.get () == 0)
          {
                System.exit (0);
          }
    }
  . . .
}// end class DrawApplication
```

Figure 7: *WinCount* References After Transformation

```
package CH.ifa.draw.Control_Panel;

public class Cpanel
{
    public static class winCount {
        public static int winCount;

     public static int create ()
     {
         return winCount;
     }

     public static int set (int value)
     {
            winCount = value;
            return winCount;
      }

     public static int get ()
     {
            return winCount;
     }

 }// end class Cpanel
```

Figure 8: Cpanel Class For *WinCount*

a local host. It boasts special white board capabilities to allow all parties to view an image during a group conference. It also allows for public and private chat rooms and user authentication. Babylon Chat can be run as a Java applet within a web page or as a Java application. It is simple to set up a local host on one computer.

3.2.1 Tuneable Parameters

The same features of Babylon Chat which differentiate its architecture from that of JHotDraw also lend it to the use of different styles of tuneable parameters. Because it is not as clearly modularized, Babylon Chat uses many global lists and shared global variables to keep track of system parameters. Examples include variables which keep track of the online traffic and chat room variables. For instance, the variables representing window height and width for each application are both stored globally in the *Babylon.java* class. These can be used in a similar way to how the *winCount* variable was used in the last section.

There are two main complications associated with the Babylon Chat code in comparison with the more structured JHotDraw code. The first is that the most useful global attributes are stored in Vectors. Because our proof of concept of STAC was originally designed to monitor only scalar types, Vectors were of little use but realistically of a lot of interest. Therefore, the STAC transformations (in TXL) were altered during the implementation phase to include the ability to mark up a Vector. In keeping with the scalar type limitation however, only the attribute *Vector.size()* is actually monitored within the Control Panel. This allows the program to track the number of elements currently inside of a Vector (by using the *trimToSize()* and *size()* methods in the Vector class) and watch as it grows and changes. In a way this creates a second reference to the Vector which can be thought of as a copy.

The second pertinent complication associated with the source code structure of Babylon Chat is that because many variables are stored and passed globally, there are many more references to tuneable parameters scattered throughout the program. This presents the opportunity to test STAC's true capabilities at changing all types of references. As a consequence of the large number of references, there are also many different references to the same variable name. This requires an added layer of transformation by STAC to ensure that each reference to the Control Panel represents a unique instance and consequently, unique renaming and updating of these names across the source code is required. This is a major technical challenge of STAC and is detailed in the examples later in this section.

3.2.2 Transformation

Consider the variable *clientId*, declared as a global in the class *babylonPanel*. Initially, the value of *clientId* is set to 0. The purpose of this variable is to assign a new *clientId* to each

client thread so that they can be referenced by this index. *ClientId* may not seem very useful for monitoring or tuning but this example illustrates how one scalar type can require dozens of reference replacements as well as unique renaming by the STAC transformation in order to work properly. It should be noted that on a larger scale this example poses a referencing challenge because of interface implementation within *babylonPanel*. This is discussed in more detail in Section 6.2.

To understand the renaming challenge of this transformation, consider that many unique classes refer to a *clientId*. For example, the class *babylonClient* declares a *mainPanel* which is of type *babylonPanel* and therefore must have its own copy of *clientId*. If the Control Panel were to create just one copy of *mainPanel* and within that, one variable for *clientId*, there would be just one copy when in fact there should be one for each *mainPanel*.

To resolve this conflict, the STAC transformation has a detailed process that checks the names of all marked up parameters as well as the Objects which may reference their own copy of each parameter. In this case, this means renaming each *mainPanel* as a unique string called *mainPanel#* where the number sign is filled in with a number from 1-9 (Figure 9). Such name changes must be propagated throughout each class. This works well in Babylon Chat because the Panel declarations are local to each class. Thus, it is a matter of finding the unique renaming associated with a class and then changing all other *mainPanel* references within that class to the appropriate *mainPanel#*. Note that this form of Control Panel incorporates nested if-statements as a way to decipher which Object called a Control Panel method at runtime. This also requires adding a parameter representing the calling Object's name to the methods in the home class (class where the variable is originally declared and marked up). After the initial method uses the parameter representing the calling Object's name, it is replaced with null in the subsequent method calls.

Now that a difficult example of scalar types has been discussed, it is pertinent to show some Vector transformations. Consider the variable *userList*, which is declared as a Vector in the

```
package cp.Control_Panel;

public class Cpanel
{
    public static class clientId {
    public static int clientId;

    public static int create (int InitialValue)
    {
        return clientId;
    }

    public static int set (int value, String callingObj)
    {
        clientId = value;
        return clientId;
    }

    public static int get (String callingObj)
    {
          if (callingObj.equals ("mainPanel9"))
               return mainPanel9.clientId.get (null);
          if (callingObj.equals ("mainPanel8"))
               return mainPanel8.clientId.get (null);
          if (callingObj.equals ("mainPanel7"))
               return mainPanel7.clientId.get (null);
          if (callingObj.equals ("mainPanel6"))
               return mainPanel6.clientId.get (null);
          if (callingObj.equals ("mainPanel5"))
               return mainPanel5.clientId.get (null);
          if (callingObj.equals ("mainPanel4"))
               return mainPanel4.clientId.get (null);
          if (callingObj.equals ("mainPanel2"))
               return mainPanel2.clientId.get (null);
          if (callingObj.equals ("mainPanel2"))
               return mainPanel2.clientId.get (null);
          if (callingObj.equals ("mainPanel1"))
               return mainPanel1.clientId.get (null);
          if (callingObj.equals ("panel1"))
               return panel1.clientId.get (null);
          else return clientId;
    }

}
. . .
public static class mainPanel9 {}

public static class clientId {
   public static int clientId;
   . . .
}
```

Figure 9: Control Panel with Renaming

class *babylonClient.java*. This variable stores information about the active users online during any given time. By using STAC to create a Control Panel capable of tuning the *size()* attribute of the *userList*, it is possible to monitor the number of active users online at one time.

The first step in this process is to mark up the tuneable parameter declaration. Because this tuneable parameter is a Vector type rather than a scalar type, special *<vector_param>* tags are used, to indicate to the transformation rules that the *size()* attribute of this particular Vector is to be added to the Control Panel. Figure 10 shows the initial markup in the *babylonClient* class. This example is interesting because the *babylonClient* class serves as a very central Object in the Babylon Chat system. In particular, each *babylonPanel* has a specific client associated with it, declared as a

```
public class babylonClient extends Thread
{
    protected Socket socket;
    protected DataInputStream istream;
    protected DataOutputStream ostream;
    protected boolean stop = false;

    protected double protocolVersion = 2.1; // by default
    protected babylonPanel mainPanel;
    <vector_param>
    protected Vector userList = new Vector();
    </vector_param>
    protected Vector roomList = new Vector();
    protected Vector ignoredUsers = new Vector();

    private Class thisClass = babylonClient.class;
. . .
}
```

Figure 10: Tuneable Parameter Initial Markup of Bablylon Chat Client Class

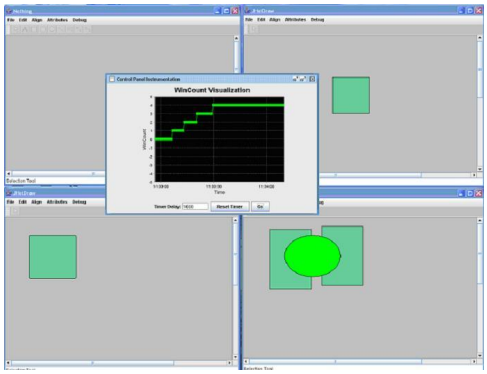

Figure 11: Visualization of JHotDraw WinCount

global variable in the *babylonPanel* class. Furthermore, many of the other classes create their own Panel to reference when adding attributes to the system. Each of these panels abstractly has its own copy of a client, as discussed in the previous example using *clientId*. Therefore, this example contains three levels of references which must be changed to either point to the Control Panel or be renamed to preserve semantics.

These necessary reference changes take place during the reference tracing and identification phase of the STAC process. Similar to the *mainPanel* example, each client needs to be renamed and then these renamings must be propagated correctly to avoid confusion in other classes where the client variable is used. For example, if the *client* variable in class *babylonChat* becomes *client1*, then every class that extends babylonChat or instantiates a new *babylonChat* Object must propagate this change to its *client* reference.

4 Applications

Once the STAC tool has been applied, the resulting Control Panel provides complete isolation of tuneable parameters and acts like the harness under the hood of a car, allowing localized tuning and complete control in a central location. The uses of the Control Panel are many and include variable monitoring and both static and dynamic variable tuning.

Monitoring of tuneable parameters can be implemented as a visualization of parameters at runtime. To demonstrate this for STAC, we have chosen to use JFreeChart [3], a graphics package freely available that has the capability to create dynamic time-based plots at runtime. With JFreeChart, the Cpanel class can be instrumented and then visualized dynamically. Going back to the JHotDraw example, where the *winCount* variable was marked up and isolated in the Control Panel, only a few simple lines of instrumentation code are needed in the Control Panel to implement dynamic visualization. The *winCount* variable simply keeps track of the various JHotDraw application windows, because in a drawing editor it may be necessary to have several windows open at one time. Figure 11 illustrates a screen shot of the dynamic visualization of several JHotDraw windows running simultaneously. The four windows are all separate applications of JHotDraw. The dynamic plot shows time increasing as each new window is opened, resulting in a stepwise function because they were opened at corresponding intervals. If several more were opened, the value of the *winCount* variable would increase accordingly. If any of the windows were closed, it would decrease and this would be visualized in real time.

Figure 12 shows the minimal instrumentation necessary inside the Cpanel class to extract this information. Inside the *set()* method for *winCount*, all that is needed is the simple writing of values to a buffer which is read by the JFreeChart graphing program as data. Every time the value of *winCount* is updated, this value is written out, and then

```
public class Cpanel
{
   public static int set(int value)
   {
      try
      {
         FileOutputStream out = new FileOutputStream ("JFreechart location ");
         PrintStream prStream = new PrintStream(out);
         prStream.print(value);
         winCount = value;
         prStream.close();
      }
      catch (FileNotFoundException e)
      {
         System.out.println("ERROR: could not find data file to write to");
      }
      return winCount;
   } // end set method for winCount
   . . .
} // end class Cpanel
```

Figure 12: Control Panel Instrumentation for JHotDraw WinCount Visualization

```
public static int get() throws AWTException{
    try
    {
        int threshold = 4;
        if (winCount > threshold)
        {
              Robot r = new Robot();
              r.mouseMove(1500, 1024);
              r.mousePress(InputEvent.BUTTON3_MASK);
              r.mouseRelease(InputEvent.BUTTON3_MASK);
              r.delay(20);
              r.mouseMove(1200, 900);
              r.mousePress(InputEvent.BUTTON1_MASK);
              r.mouseRelease(InputEvent.BUTTON1_MASK);
        }
    }
    catch (Exception e)
    {
        System.out.println(e.getMessage());
    }
    return winCount;
}
```

Figure 13: Adding Threshold Checking in the Control Panel

displayed by the dynamic plot shown above. Note that originally there were four references to *winCount* within the *DrawApplication* class. Without the STAC re-architecture, the system would require four different references every time it wanted to change the *winCount* variable. Keeping in mind that this is a notably simple example, scaling this up would provide a tremendous increase in control as well as a decrease in necessary maintenance time.

In addition to parameter visualization, the STAC Control Panel can be used to tune each parameter of interest either retroactively, after a visualization, or on-the-fly at runtime. For very accurate tuning of specific values, or for tuning which requires some sample data before analysis, static tuning can be applied to the Control Panel after run-time. The isolation of tuneable parameters aids in this process, eliminating multiple program accesses, although for co-dependent variables some program manipulation may be required.

The dynamic tuning possibilities enabled by STAC are also intriguing. Through simple value comparison, a tuneable parameter can be monitored until it goes beyond a certain threshold and then reset to a more desirable value. This does not apply well to *winCount* because resetting the *winCount* would not serve a useful purpose for system tuning. However, sanity-checking the values in the Control Panel and then redirecting the program to correct an error is also possible and can be nicely illustrated using the *winCount* variable.

Suppose that during program execution, one wanted to minimize overcrowding of the workspace with respect to JHotDraw application windows. To do so, one could dynamically tune the Control Panel to monitor the *winCount* variable and react if the number of open JHotDraw Windows becomes larger than a certain threshold size by tiling the rest of the Windows equally and setting the current Window on top. To do so, all that is needed are some simple additions to the Control Panel.

Within the Control Panel, a simple *Robot* object can be created. The *Robot* class, found in the AWT package, allows a *Robot* Object to be created which can itself create native events in Java at runtime. Therefore, this allows us to mimic a user, in keeping with the autonomic undertone of this experiment. The *Robot* can then be instantiated and can perform a specific autonomic action whenever the *winCount* variable goes above its threshold. Figure 13 shows the Control Panel with a *Robot* being created for a threshold of 4. Figure 14 details the Desktop view both before,when the number of open windows is 4 , and Figure 15 shows after, when there are five open windows.

As an extension to the above example, the Control Panel can continue in a loop, monitoring the current value of the *winCount* variable the entire time the application windows are open. If the number of Windows gets beyond seven, then the tiling could be considered too crowded and all but the current Window will be minimized. Because the loop will continue monitoring the number of Windows, if the user closes a JHotDraw application window later, the tiling will re-appear. Figure 16 shows

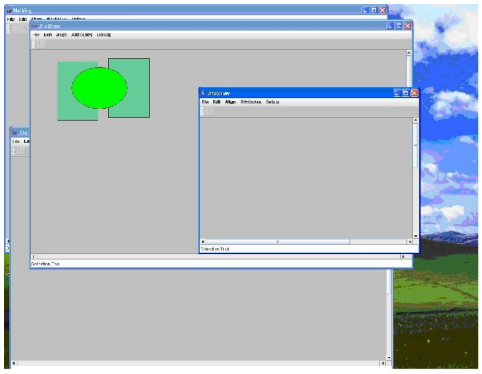

Figure 14: Before Dynamic Tuning of Desktop

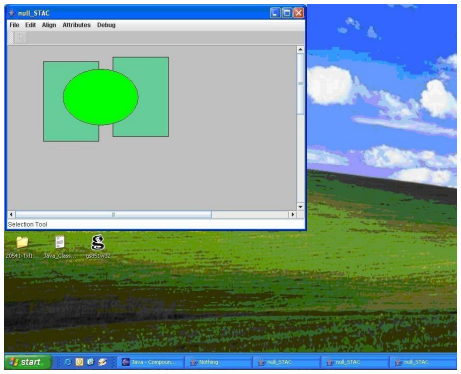

Figure 16: Minimized Windows

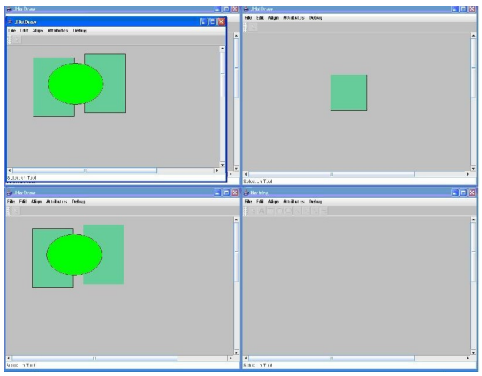

Figure 15: After Dynamic Tuning of Desktop

Figure 17: After Re-tiling

the Desktop when the number of open Windows has reached seven and all but the current Window have been minimized. Figure 17 shows later on in the application, when one window has been closed, and the tiling has been reset.

Although there are times when allowing a user to change or create a variable within the Control Panel could be dangerous and potentially allow corruption of parameters, a solution to this actually presents itself from within the Control Panel. Consider the case where the Control Panel is used to check for and handle corruption of variables caused by remote access. To implement this is strikingly simple. Each time a legal *set()* is called for a tuneable parameter, this value is stored within the Control Panel. Every time a *get()* is called after this, its return value is compared with this record of the last legal *set()*. If the two values differ, the value is reset to the last legal call. This presents a simple, elegant solution to the same security threat which is implied by allowing users access to the variables within the Control Panel.

Some of the potential of STAC has been investigated and confirmed in this section using examples of transformations on both JHotDraw and Babylon Chat. However, there is still much work to be done. Section 6.2 examines what limiting assumptions have been made to this point, and what future projects may evolve from the current STAC project.

5 Related Work

The STAC project draws on three main areas of current research in computing. These are autonomic computing, automated refactoring, and source transformation.

Although the domain of autonomic computing is relatively new, there are a few projects

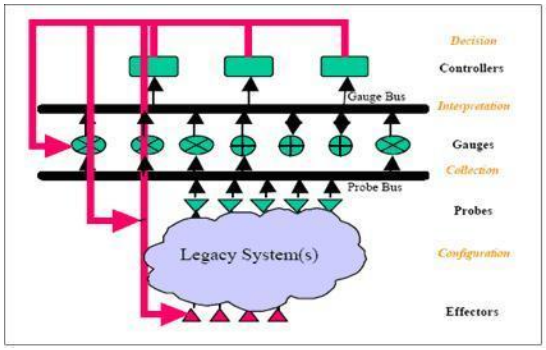

Figure 18: Autonomizing Legacy Systems Approach (from [8])

which parallel STAC in their attempts to implement self-monitoring and tuning behaviours in existing systems. One of the most closely related projects is Kaiser et. al's work on Autonomizing Legacy Systems [8]. In their system, Kaiser et. al develop a protocol for adding a series of probes to legacy systems which then deliver instrumentation information to a series of gauges that perform monitoring functions based upon this information. The implementation is not trivial. It relies upon a layered architecture with the above mentioned probes collecting information, the gauges interpreting it, and the effectors performing reconfigurations and system adaptations as necessary. This system is shown in Figure 18.

The system is noteworthy for its detailed instrumentation of a legacy system, transforming it into a system which allows for autonomic control. The ideas behind this particular example draw many parallels with the implementation of STAC. However, the STAC solution is able to add only one module to any software system, which encapsulates the idea of the probes, gauges and effectors in the work of Kaiser et. al. STAC can be thought of as combining these layers into one layer, the Control Panel. In a way STAC is working on a subset of the problem tackled by Kaiser et. al.

The reference identification and tracing of STAC can be likened to the aspect-oriented style of programming, but produces the re-architecture without the separation of code into cut points, thus maintaining the original code structure and making the maintenance of the result much simpler. Also, there is no need to recognize underlying scattering or distribution of the tuneable parameters in STAC, as one must do to identify the join points in aspect-oriented programming, because the re-architecture is completely automated based solely on the markup of the declaration of variables of interest.

The technical challenges of implementing STAC can also be compared to those in aspect-oriented programming, which could in principle be used to implement the rearchitecture done by STAC. However, the problem of locating and tracing every possible reference to a parameter of interest is complex, making it difficult to identity appropriate syntactic cut-points. Moreover, once rearchitected, STAC's implementation of the Control Panel as a separately linked module rather than an aspect has the additional advantage that the Control Panel can be maintained independently of the program itself, with no dependence on source.

Automated refactoring techniques tackle technical challenges similar to those in STAC, as they work to re-structure code to improve performance or maintainability and STAC has an analagous goal of re-architecture. Srinivasan [10] provides a Modularity Toolkit to automatically re-modularize legacy source code. The system first standardizes the code, eliminating discrepancies, much like the automatic renaming in STAC. After this, the Toolkit breaks apart the existing modules. Next, it carefully analyzes the scope and localization of specific references and variables, using clustering to propose and create new, more effective modules. This, again, is followed in STAC during the reference tracing phase. Once these modules are put into place, the code is searched for dependencies within and between modules, resulting in a re-modularized piece of code. Much of the implementation of the Modularity Toolkit relies upon source transformation, particularly with TXL, which leads into the final domain of related work for this project.

TXL [6] is a programming and rapid prototyping language designed to address source transformation problems of a structural nature. It uses a hybrid rule-based and functional programming style to transform input data which must be represented in tree format.

TXL has a wide spectrum of possible uses. Key to the STAC experiment are TXL's capabilities in syntactic manipulation of source code, specifically uses which aim to select and manipulate only parts of source code with a particular property or of particular interest. Also critical is TXL's potential for agility and adaptability.

Many projects involving source restructuring must be able to focus in on one part of the code, essentially working around the rest to leave the semantics unscathed. In STAC, it is the tuneable parameters and their references which are of the most importance. It is therefore desirable to investigate how TXL can be used to create such a localized focus and ignore the rest of the source.

This idea has been explored in the use of TXL to identify clones in source code [7]. The term *clone* in this sense refers to identical or near-identical copies of source code dispersed through a program or web page. In this case, an *island grammar* was crafted to detect clones. An extractor written in TXL is then used to "pretty print" the source code *islands* (containing possible clones) to improve the accuracy of a comparator. Specifically, the transform works by separating source code features by lines so that the use of the Unix *diff* command will be most effective when comparing code. This promotes the goal of matching clones and near-miss clones. It is interesting how easily *islands* can be isolated from the rest of the source code.

Another TXL technique of particular interest to our work is *agile parsing*. In this sense, agile means "the ability to use a customized version of the input grammar for each particular analysis and transformation task" [5]. Specifically, grammar overrides are used to allow for specific cases to be addressed in a transformation. The article also uses an island grammar similar to that used in [11], again showing how TXL can be used to focus in on certain aspects of source code while leaving less interesting bits out.

While we have exploited the use of TXL in the STAC prototype, other source transformation systems such as ASF+SDF [12], Stratego/XT [13] or ANTLR [9] could easily serve as well.

6 Summary and Future Work

The previous several sections have outlined the STAC transformation tool in detail and followed several example experiments run with STAC. In doing so, some key strengths and benefits of the STAC tool have been highlighted and important applications have been discussed. However, there are limitations and limiting assumptions which must be adhered to at this time. Such limiting assumptions naturally suggest future work in the area.

6.1 Parallel Projects

STAC can be looked at as part of a larger package of projects which, when combined, could be used to automate the entire autonomization of a system, with respect to tuneable parameters. The first step of this process is the markup of tuneable parameters, which in STAC is done by hand. However, as a parallel project, the identification and markup of interesting tuneable system parameters would fit into the package as the first step. In addition, this kind of tool would ensure that STAC receives meaningful input to exploit its capabilities and produce the maximum benefit.

Another critical part of the process which has only been touched upon in the Examples section of this paper is the generation and implementation of autonomic controllers. Such controllers are currently being implemented for other projects, but generation of controllers built to work specifically with Control Panels created by STAC has yet to be explored.

6.2 Current Scope and Natural Extensions

The scope of STAC presently includes local and global scalar variables which may or may not extend across classes and which may also necessitate several distinct copies at runtime. The first obvious limitation imposed is that the variables must be scalar types (with the exception of Vector objects which may be instrumented for their size attribute). A natural extension of this transformation would be to look at more general objects as tuneable

parameters. Although more complicated, this has actually been started with the addition of the Vector markup and isolation in the Control Panel. This would be a very interesting follow-on research project we hope to be able to pursue.

A second, less obvious, limitation is that STAC can currently miss some kinds of subtle references. Section 2.1 uncovered the possibility of having many different objects sharing one Control Panel as well as the more simple case of having just one object using the tuneable parameters at runtime. For the time being, this suffices as a proof of concept as STAC works well to capture most program references to tuneable parameters of interest. However, the transformation at present can miss some references which change when they are passed as parameters to other objects and methods. At this time, STAC also does not handle aliasing. This is particularly limiting when a tuneable parameter gets passed into an object constructor as a reference and then is changed during the constructor method and reassigned. The handle is effectively lost and this affects the usage of this parameter later in the program. To remedy the loss of these references, it would be necessary to add a pointer to the object which currently is passing this reference. This has been successfully done in the transformation to solve another problem, namely that of having multiple objects which share several copies of a tuneable parameter, as in the final Babylon Chat example. Although this is difficult in Java, it may be less cumbersome in other languages.

Java interfaces also presently pose a problem for STAC whereby the methods which must be present in a class when it implements an interface could be automatically changed by the transformation when extra parameters are added to keep track of the calling object's name. Currently, rules have been added to exclude specific methods from having parameters automatically added to them and this could be continued on a case by case basis or alternatively solved exhaustively with rules to handle all possible interfaces in Java.

Finally, as parallelism is increasingly part of Java applications, it would be useful for STAC to have functionality to handle the case where multiple threads have access to the same variables and create a race condition. This is considered future work and would require the use of Java synchronization mechanisms to guarantee mutual exclusion during tuneable parameter access.

As an additional extension, it is future work to integrate the entire STAC transformation into the Eclipse IDE as a plug-in. Along these same lines, it would be interesting to implement the STAC tool for languages such as C or C++, allowing for more systems to be re-architected and potentially solving some of the current limitations imposed by using Java. Because of the choice of TXL as a transformation language, this should not be too challenging.

6.3 Side Effects on Performance

It is critical to address the question of possible side effects a STAC transformation may introduce to a system. Of paramount importance is the potential increase in run time caused by the numerous code changes and TXL transformations involved in STAC. To investigate this, a STAC transformation was run on a simple command-line Java interest calculator [2]. The compound interest calculator takes in three arguments from the command line. The first is the number of years the money will be invested, the second is the monthly investment amount (in dollars), and the third is the initial amount deposited (in dollars). The program then calculates the investment growth and total amount at a monthly interest rate of eight percent with thirty percent deducted for taxes.

A small experiment was run on a time-shared Unix system, with 8GB of memory, and 4 750 Mhz processors in order to investigate the effect of a STAC transformation on system performance in terms of run time. The experiment compared the run time of the original code, to that of the code which had undergone a STAC transformation. The STAC code had been marked up to isolate the variable *amount*, which is representative of the initial amount deposited. Each program was then given the same input arguments, growing progressively larger. The results, given in the two tables below, show that for small amounts there is virtually no difference. However, when very large

Input (Years, Monthly, Initial)	User Seconds	System Seconds
1000,1000,1000	2.0	0.0
5000,5000,5000	6.0	2.0
10,000 , 10,000, 10,000	9.0	4.0
20,000, 20,000, 20,000	17.0	9.0
50,000 50,000, 50,000	43.0	24.0

Table 1: Original Source Code Run Times

Input (Years, Monthly, Initial)	User Seconds	System Seconds
1000,1000,1000	2.0	0.0
5000,5000,5000	6.0	2.0
10,000 , 10,000, 10,000	10.0	5.0
20,000, 20,000, 20,000	18.0	9.0
50,000 50,000, 50,000	47.0	23.0

Table 2: Run Times After STAC Transformation

calculations are carried out, there is a significant difference in the two times, with the STAC code performing at a slower rate. However, one must consider that this program calculates interest every month. Therefore, an input value of 1000 represents 12,000 calculations. As such, further investigation is needed to determine exactly what causes the slowdown introduced by STAC, and if this is replicated or exaggerated in programs which involve user interaction.

In addition, there is a lingering sub-problem of variable dependencies. During a STAC re-architecture, a certain parameter may seem to be isolated within the Control Panel but throughout the program it may be part of a chain of variables which are dependent upon one another. Therefore, changing one of these parameters within the Control Panel, as can be done in tuning, may inadvertently cause changes or errors in the chain of dependent variables. A challenging future problem would be to pinpoint this chain of variable dependencies and use it to propagate the changes that happen from within the Control Panel, to the variables that depend upon the isolated parameter. This however, is a very complex problem and would have to be done in stages.

6.4 Conclusion

STAC has been shown through proof-of-concept to be capable of isolating tuneable parameters inside an automatically generated Control Panel specific to the given application. In doing so, it preserves semantics and makes minimal local changes in such a way that the re-architected code remains maintainable. The technical challenges of identifying all of the references to a tuneable parameter are many. They include recognizing the numerous different categories of references and replacing them with syntactically and semantically correct replacement references to the Control Panel. These challenges are compounded by having multiple objects sharing a tuneable parameter and having to uniquely rename each Object throughout the source code. The creation of the Control Panel methods is no less challenging, requiring a nested class for every object instantiation in the source code as well as a global class where references can be redirected when necessary.

A number of applications of STAC have been identified and illustrated by example. Real-time visualization of the Control Panel activities allows for comprehensive system analysis of a tuning variable which can be used retroactively to make appropriate changes. Furthermore, dynamic monitoring of variables can be effectively accomplished by the Control Panel itself, and using simple instrumentation can be extended to dynamic tuning, mimicking a human-user. These examples demonstrate our goal of facilitating independent autonomic control through automatic rearchitecture.

Acknowledgments

This work is supported by an IBM Faculty Innovation Award and by the Natural Sciences and Engineering Research Council of Canada.

About the Authors

Elizabeth (Liz) Dancy is a software developer at the IBM Software Laboratory in Markham, Ontario and a former Master's student working on the STAC project in the Software Technology Laboratory of Queen's University under the supervision of James Cordy. She completed her Bachelor of Computing at Queen's as well, with a subject of specialization in Software Design. Her research interests include source transformation, autonomic computing and software architecture.

James Cordy is the Director of the School of Computing and Professor of Computing and Electrical and Computer Engineering at Queen's University. From 1995 to 2000 he was Vice President and Chief Research Scientist at Legasys Corporation, a software technology company specializing in legacy software system analysis and renovation. Dr. Cordy is a founding member of the Software Technology Laboratory at Queen's University and winner of the 1994 ITRC Innovation Excellence award and the 1995 ITRC Chair's Award for Entrepreneurship for his work there. He serves on a range of software engineering conference committees and has recently co-chaired several conferences and workshops including CASCON 2005. Dr. Cordy is an IBM Faculty Fellow and has been awarded IBM Faculty Innovation Awards in both 2004 and 2005.

References

[1] BabylonChat. http://visopsys.org/andy/babylon/.

[2] Compound9 Interest Calculator. http://www.ping127001.com/java/Compound9.java.

[3] JFreeChart. http://www.jfree.org/jfreechart/.

[4] JHotDraw 6.0. http://www.jhotdraw.org.

[5] J.R. Cordy. Generalized Selective XML Markup of Source Code Using Agile Parsing. In *International Workshop on Program Comprehension*, pages 144–153, 2003.

[6] J.R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[7] J.R. Cordy, T.R. Dean, and N. Synytskyy. Practical Language-Independent Detection of Near-Miss Clones. In *Proc. CASCON'04: 2004 IBM Centre for Advanced Studies International Conference*, pages 1–12, 2004.

[8] G. Kaiser, P. Gross, G. Kc, J. Parekh, and G. Valetto. An Approach to Autonomizing Legacy Systems. In *Proc. ACM Workshop on Self-Healing, Adaptive and self-MANaged Systems(SHAMAN 2002)*, June 2002.

[9] T.J. Parr and R.W. Quong. ANTLR: A Predicated LL(k) Parser Generator. *Software, Practice and Experience*, 25(7):789–810, 1995.

[10] R. Srinivasan. Automatic Software Design Recovery and Re-Modularization Using Source Transformation. Master's thesis, School of Computing, Queen's University, 1993.

[11] N. Synytskyy, J.R. Cordy, and T.R. Dean. Robust Multilingual Parsing Using Island Grammars. In *Proc. CASCON '03: 2003 IBM Centre for Advanced Studies International Conference*, pages 266–278, 2003.

[12] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling Language Definitions: the ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[13] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*. Spinger-Verlag, 2004.