Stony Brook University

Implementing Tracecuts in the InterAspect Program Instrumentation Framework

A Thesis Presented

by

Ketan Dixit

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

December 2010

Stony Brook University

The Graduate School

Ketan Dixit

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Professor Scott Smolka – Thesis Advisor
Department of Computer Science

Professor Erez Zadok – Chairperson of Defense
Department of Computer Science

Professor Scott Stoller
Department of Computer Science

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

Implementing Tracecuts in the InterAspect Program Instrumentation Framework

by

Ketan Dixit

Master of Science

in

Computer Science

Stony Brook University

2010

As software grows in complexity, there is a need to check the runtime behavior of programs for potentially hazardous runtime states, and take the appropriate action. The tracecut mechanism, which allows one to match sequences of runtime events against a property specification given as a regular expression, provides us with this functionality.

In this thesis, we show how tracecut functionality can be applied to C programs by making use of InterAspect, an aspect-oriented instrumentation framework. InterAspect is a GCC compiler plug-in that performs runtime instrumentation at the GIMPLE level, GCC's intermediate representation. Our approach interprets a tracecut specification given as a regular expression as a finite state machine, and generates the code needed to

perform the state machine transitions.

The utility of our approach is illustrated by two case studies, one involving a tracecut for a simple data-source iterator, and the other involving a tracecut specification of file open-close behavior. The latter tracecut is applied to the bzip2 file compression utility.

To

My Family and Teachers

Table of Contents

Acknowledgements

First of all, I wish to sincerely thank Prof. Scott Smolka for his guidance and support all through the project. I would also like to thank Prof. Zadok, Prof. Stoller, Prof. Grosu and Dr. Havelund for their guidance. I would also like to thank all the SSW project group colleagues Justin Seyster, Xiowan Huang and Tushar Deshpande. In particular, I would like to thank Mr. Justin Seyster for his insightful advice and help throughout my thesis work.

Chapter 1

1 Introduction

A Tracecut is a concept which involves matching to a specific trace of events in a program at runtime. Tracecut is also referred to as Tracematch and more generally as "typestate property".

In Aspect-oriented programming, pointcut can refer to only the current program state or the current join point but the tracecut or type state property does matching on the trace of events in the program [1]. This concept is necessary in detecting run-time properties of a program and take appropriate corrective actions. It is useful for specifying safety properties (safe iterator example [1]) to allow proper use of resources. Several implementations for detecting Tracecut exist in Java built over **AspectJ** framework. We referred to paper 'Adding Trace Matching with Free Variables to AspectJ' [1] in designing the system. This is an attempt to bring the Tracecut functionality to C by building over **InterAspect** [2] the C-based aspect-oriented instrumentation framework. Thus, it could be potentially used for checking run-time properties of complicated system software. In our implementation of tracecut, we take regular expression as input to define the pattern of events and we parse that expression to generate code for state machine described by the expression.

Our tracecut implementation is implemented as API similar to the InterAspect design. As tracecut makes extensive use of InterAspect, we include a special section that explains InterAspect terminologies and APIs in brief. Then we will provide overview of the tracecut architecture and the components involved. In the next section, we give the API design of the tracecut. In the API design we will explain the purpose of each function and the description of the parameters it takes. We will follow API design up with two working examples one that checks for iterator safety and the other that protects writes over a closed file handle. Then, we explain the design of the tracecut in detail. In design we discuss how we actually interpret the regular expression to state machine code, how do we create state machines, how do we search on them with subset of attributes etc. Then we discuss the limitations of the approach, the desired features from InterAspect

that would be useful for tracecut and also future work. Finally we end the document with results, conclusion and references.

Chapter 2

2 InterAspect Overview

As mentioned earlier InterAspect is an aspect-oriented instrumentation framework that is implemented as a GCC plug-in. It provides the functionality to instrument a program while hiding the implementation details of GCC. It does instrumentation at the GIMPLE level, which provides the user with detailed type information. InterAspect makes use of familiar vocabulary of Aspect-oriented programming to develop instrumentation plug-ins. We will explain the terms used from Aspect-oriented world are *pointcut*, *join point*, *advice* and *weaving* briefly.

A pointcut denotes a set of well defined program points called join points where calls to advice function can be inserted by process of weaving. In simple terms a pointcut acts as a matcher over the entire source code and each point in the source that is matched by the matcher is called join point. An advice function is some action user wants to execute at each join point. The InterAspect provides a novel feature of join callbacks which allows customized weaving. We have made use of this feature in implementing Tracecut.

InterAspect uses an API based approach to define pointcuts. The pointcuts are created by calling `aop_match_*` functions. There are following types of pointcuts function call, function entry, function exit, and assignment. These pointcuts are further filtered by parameter types, return types (only in case of function call pointcut) etc. by `aop_filter_*` functions. After filtering we have to specify callback function which will get called for each join point. For a join point, various attributes such as parameters return values etc. can be captured with the help of `aop_capture_*` functions. Finally calls to advice functions can be inserted with the help of `aop_insert_advice` function. It is possible to pass both static parameters and dynamic parameters.

Chapter 3

3 Tracecut Concept and Overview

Aspect oriented programming provides ability to call an advice function at a specific point in the program. This means that it inserts instrumentation based on static information about the program at compile time. Tracecut provides the ability to match on a pattern in the sequence of events that take place at run-time. Tracecut are also referred to as “Tracematch” in [1] or “Stateful Aspects” in [3]. Tracematch is an implementation of history based pointcut matching on the top of AspectJ. We have similarly tried to build tracecut over InterAspect, the Aspect-oriented instrumentation framework.

A tracecut defines a pattern of events and an advice to be called if the current trace matches that pattern. Each tracecut declaration has the 4 parts: Declaration of tracecut symbols, declaration of tracecut parameters, binding tracecut parameters to the information available at each symbol such as parameters, return values etc, and finally specification of regular expression of the symbol and specification of the advice. The pattern of symbols is specified with the regular expression. A match occurs when a trace of events belongs to the regular language described by the regular expression.

Each event that occurs in a program is modeled by transitions on the state machine corresponding to the regular expression. Multiple state machines can exist in program at runtime. The tracecut parameters act as attributes of the state machine. At each join point described by the tracecut symbol we might have all or subset of these parameters. We use this list of parameters available at the join point to search for the appropriate state machine to make the transition on.

The following diagram describes the Tracecut architecture. Tracecut was designed to be built on top of InterAspect. It makes use of InterAspect API to achieve the instrumentation. The tracecut specification is interpreted to form appropriate pointcuts. The advice weaved into for these pointcuts consists of two parts one is static `tracecut-helper.c` and other is auto-generated `statemachine.c`. The `tracecut-helper.c` contains the helper functions to do state machine operations. The state machine is initialized at a symbol with a set of attributes. At occurrence of other

symbols/events there may be all or subset of these parameters available. Now these parameters have to be used to find to which state machine they belong. Thus the code for all these operations is in `tracecut-helper.c`. Also, the regular expression specification is converted to code simulating the state machine. Such code exists in `statemachine.c`. The specification compiler thus creates weave module which would insert calls to do state machine transitions in compilation process.

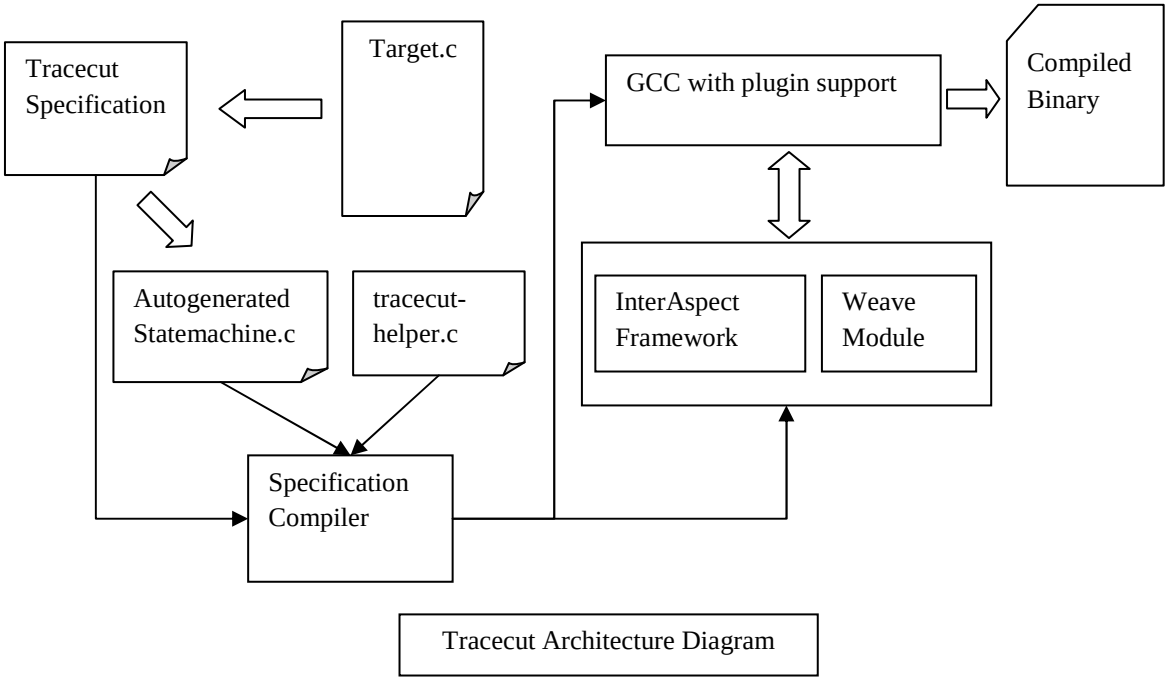


Figure 1.1 Tracecut architecture

Chapter 4

4 Tracecut API Design

The Tracecut is basically a regular expression over the sequence of events that occur in the program. These events are modeled by Tracecut symbols. Tracecut symbols refer to specific event in the program. Tracecut symbols refer to well defined points in the program which are described by a pointcut in InterAspect. Now a Tracecut also has several parameters that are common to all symbols so these parameters are modeled by defining Tracecut parameters. These Tracecut parameters are the attributes of the state machine they are defining with the help of regular expression. A collection of these parameters is used as a key for identifying the right instance of the state machine. Moreover these parameter need to be mapped with function parameters or return values associated with each symbol.

We will first present the Tracecut API design followed by the working example.

4.1 Steps in defining tracecut using APIs

The Tracecut API was designed with the goal of making it easy and intuitive to the user. The Tracecut will be written as a GCC plug-in based on the InterAspect framework. But the API is designed such as the user need not bother about the details of the InterAspect framework.

For understanding the use of the API, it is necessary to describe the following steps in creating a Tracecut.

- Creating a Tracecut: This step will create a pointer to Tracecut in which Tracecut symbols and Tracecut parameters are stored.
- Adding Tracecut symbols to Tracecut: In this step we will create a Tracecut symbol corresponding to a function call, function entry, function exit.
- Adding Tracecut parameters to Tracecut: We need to specify parameters to the Tracecut to define the attributes of the state machine we are going to describe.

- Mapping Tracecut parameters to the Tracecut symbol parameters: Each of the Tracecut parameter has to be mapped to at least one Tracecut symbol parameter or return value.
- Specifying the advice parameter and name: User can specify what function to call and what parameter can be passed to the function.
- Specifying a regular expression of symbols: A regular expression involving symbol names has to be specified in the tracecut. It is basically input for creating a state machine.

4.2 Tracecut API description

The `create_tracecut` function creates a tracecut instance which is further filled with more information later. This structure acts as a container to all the tracecut related information.

```
struct tracecut *tc = create_tracecut();
```

The `add_tracecut_symbol_call` function tells the tracecut to catch each call of function having name `func_name` and generate the state transition before the call to the function. The symbol name will be used to refer to this Tracecut symbol later.

```
add_tracecut_symbol_call (
     struct tracecut *tc,  /* Tracecut pointer */
     const char *sym_name, /* Symbol name */
     const char *func_name /* Function name */
     AOP_BEFORE            /* Advice insertion location*/);
```

The `add_tracecut_symbol_entry` function tells the tracecut to catch event of entry in the specified function and generate the state transition at the entry to the function. The symbol name will be used to refer to this tracecut symbol later.

```
add_tracecut_symbol_entry (
     struct tracecut *tc,  /* Tracecut pointer */
     const char *sym_name, /* Symbol name */
     const char *func_name /* Function name */ );
```

Similarly we have symbol for capturing function exits.

```
add_tracecut_symbol_exit (
     struct tracecut *tc,   /* Tracecut pointer */
     const char *sym_name,  /* Symbol name */
     const char *func_name  /* Function name */ );
```

The `add_tracecut_param` function adds a parameter to the tracecut specifying the tracecut parameter name and the type respectively. The type specifier `aop_type` is provided by the InterAspect framework.

```
add_tracecut_param (
     struct tracecut *tc,          /* Tracecut pointer */
     const char *tc_param_name,    /* Tracecut Param name */
     struct aop_type t            /* Type of the param */ );
```

The `bind_to_param` function does the mapping of tracecut parameter to the parameter of a tracecut symbol specified by the index.

```
bind_to_param (
    struct  tracecut *tc,      /* Tracecut pointer */
    const char *sym_name,      /* Tracecut symbol name*/
    int index,                 /* Symbol parameter index */
    const char *tc_param_name /* Tracecut parameter name*/);
```

Similarly, `bind_to_return` function does the mapping of tracecut parameter to the return value of a particular symbol. Note that this applies only to Tracecut symbols corresponding to function call pointcuts.

```
bind_to_return (
   struct  tracecut *tc,        /* Tracecut pointer */
   const char *sym_name,        /* Tracecut symbol name*/
   const char *tc_param_name /*Tracecut parameter name */);
```

The function `set_initial_symbol` allows user to specify at what tracecut symbol the state machine instance should be created.

```
set_initial_symbol (

     struct tracecut *tc,     /* Tracecut pointer */
     const char *sym_name,    /*Tracecut symbol name*/);
```

The function `set_advice_parameter` sets the tracecut parameter that needs to be sent to the advice.

```
set_advice_parameter (
     struct tracecut *tc,        /*Tracecut pointer */
     const char *tc_param_name   /* Tracecut param name*/);
```

The `create_tracecut_rule` is the final step in tracecut creation. It sets the regular expression on the symbols defined. This regular expression provides a pattern on the sequence of events. It also specifies the advice function to be called.

```
create_tracecut_rule (
     struct tracecut *tc,      /* Tracecut pointer */
     const char *regex,        /* Regular expression */
     const char * advice_name /* Advice name */);
```

Chapter 5

5 Tracecut examples

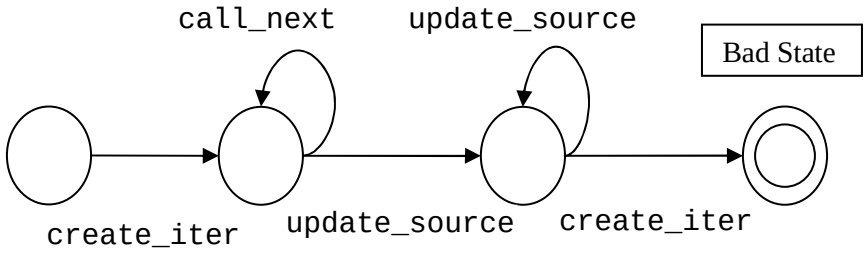
We give the following two examples to explain the use of the iterator. The first example is referred from [1].
The first example detects a condition of invalid iterator use. The second example detects write to a closed file handle and open the file handle in an advice.

5.1 Iterator Example

We will explain how the condition of using a bad iterator can be detected by using tracecut. Iterator provides the functionality to iterate through each element in the data source. An instance of iterator is created from a datasource. But when the datasource is changed, the instance of iterator is invalid and thus should not be used. By using Tracecut we can detect this situation and can call any user specified advice.
This situation could be specified as regular expression as follows

```
create_iter call_next* update_source+ call_next
```

This regular expression says that call to `create_iter` followed by zero or more calls to `call_next` followed by one or more calls to `update_source` followed by call to `call_next`. The state transition diagram corresponding to the above regular expression is as follows.



We describe a ‘safe iterator’ example where we will detect the condition when an illegal use of iterator happens. We use the a sample target program which make uses comma separated integer values as a datasource and uses an iterator structure to loop over the elements of the datasource.

The target program makes use of the following key functions.

The `create_source` function instantiates the new datasource from a string of comma separated values.

```
struct datasource * create_source (char * csv);
```

The `update_source` updates the datasource pointed to by ds with new comma separated values.

```
void update_source (struct datasource *ds, char *csv);
```

The `create_iterator` function creates the iterator which points to the first element of the list of values.

```
struct iterator * create_iterator (struct datasource *ds)
```

The `call_next` function returns the current value pointed to by the iterator and advances the iterator by one element.

```
int call_next ( struct iterator * it)
```

Hence we write the tracecut for the above target program as follows.

```
/*   Initialize the Tracecut. */
struct aop_tracecut *tc = create_tracecut();

/*
This function tells the Tracecut to catch each call of
create_iterator.
The parameter AOP_AFTER tells that state transition is to
be done after the create_iterator call.
*/
```

```
add_tracecut_symbol_call(tc,
“create_iter","create_iterator”, AOP_AFTER);

/*
 add_tracecut_param adds paramaters iterator and
datasource of any pointer type.
*/
add_tracecut_param(tc, "iterator",  aop_t_all_pointer());

add_tracecut_param( tc, "datasource", aop_t_all_pointer());

/*
bind_to_return binds the Tracecut parameter ‘iterator’ to
the return value of the ‘create_iter’ Tracecut symbol.
*/
bind_to_return( tc, "create_iter", "iterator");

/*
bind_to_param binds the Tracecut parameter ‘datasource’ to
the first parameter of ‘create_iter’ Tracecut symbol.
*/
bind_to_param(tc, "create_iter", 0, "data_source");

/*
This function specifies that ‘create_iter’ corresponds to
the initiation of the state machine.
*/
set_initial_symbol (tc, "create_iter");

add_tracecut_symbol_call(
tc,"update_source","update_source", AOP_BEFORE);
bind_to_param(tc, "update_source", 0, "data_source");

add_tracecut_symbol_call( tc,"call_next","call_next",
AOP_BEFORE);
bind_to_param(tc, "call_next", 0, "data_source");



/*
 This is the final step in Tracecut creation. It specifies
the regular expression for statemachine. It calls
‘handle_bad_iterator’ advice on satisfying the regular
expression.
*/
```

```
create_tracecut_rule (tc, "create_iter call_next*
update_source+ call_next"," handle_bad_iterator");
```

5.2 Detection of write to closed file handle example.

This example detects the condition of a write to a file handle after it has been closed. It is referred from [4]. It also calls the advice to reopen the file so that further write could be safe. The regular expression for detecting the condition is

```
open write* close write
```

This code is captures calls to fopen, fwrite and fclose. It defines a parameter called fp which corresponds to the file handle. It also binds to the file name so that it is available to pass as parameter to advice. Following is the code snippet.

```
char regex[] = "open write* close write";

tc = create_tracecut();

add_tracecut_symbol_call (tc,"close","fclose", AOP_AFTER);
add_tracecut_symbol_call ( tc,"open","fopen", AOP_AFTER);
add_tracecut_symbol_call ( tc,"write","fwrite",AOP_BEFORE);

add_tracecut_param (tc, "fp", aop_t_all_pointer());

bind_to_return(tc, "open", "fp");
bind_to_param(tc, "open", 0, "name");
bind_to_param(tc, "close", 0, "fp");
bind_to_param(tc, "write", 3, "fp");

set_initial_symbol (tc, "open");
set_advice_param (tc, "name");
create_tracecut_rule ( tc, regex,"reopen_file_handle");
```

In the advice implementation we have made an assumption that the file handle is shared between the main code and the advice code. In this case, ideally we would have needed to pass pointer to the file handle in order to store the new file handle in the same address and the following write could run successfully. But as of now InterAspect lacks this facility and thus an assumption had to be made.

Chapter 6

6 Implementation Details

The implementation of tracecut requires instrumenting state transition code at specific points in the program. We use the InterAspect framework for this purpose. As mentioned earlier the tracecut is specified as regular expression over tracecut symbols. Each Tracecut symbol corresponds to a well defined event in the program. These well defined events are modeled by using pointcuts in InterAspect. So the basic idea is to insert advice making the state transition at each join point satisfying the pointcut corresponding to the symbol. Also we pass the mapped parameters or return values of the join point to the advice in order to decide on what instance of state machine the transition should be made. So we describe the steps in generating the tracecut as follows.

6.1 Storing the Tracecut specification in a structure:

All the Tracecut related information is stored in a '`struct tracecut`'. It contains information of tracecut symbols, tracecut parameters.

```
struct tracecut
{
     /* Regular expression */
     char * regex;

     /* List of all symbols */
     struct tracecut_symbol *tc_sym_list;

     /* List of all parameters*/
     struct tracecut_param *tc_param_list;
};
```

As shown in the above snippet the Tracecut stores regular expression of Tracecut symbols and lists of the Tracecut symbols and Tracecut parameters.

As mentioned earlier the `tracecut_symbol` is actually implemented by creating a corresponding pointcut. In `tracecut_symbol` structure we store the type of the pointcut that corresponds to it.

```
struct tracecut_symbol
{
     /*Tracecut symbol type */
     enum tracecut_symbol_type tc_type;

     /* Function name */
     char *name;

     /* List of parameters associated with symbol */
      struct tracecut_symbol_param *tc_sym_param_list;

     /* Next pointer for the list */
     struct tracecut_symbol *next;
};
```

Moreover it stores the function name if it is associated with a function entry, function exit or function call pointcut. Now each program location satisfied by pointcut (also called join point) of the symbol has useful information such as parameters, return values that needs to be captured. In order to store what needs to be captured for a `tracecut_symbol`, we use a structure called `tracecut_symbol_param` and we maintain its list in the `tracecut_symbol` structure.

The `tracecut_symbol_param` structure holds the information of a symbol parameter.

```
struct tracecut_symbol_param
{
     /* -1 for return value */
     int param_index;

     /* The tracecut_param bound to this symbol
     parameter */
     struct tracecut_param *tp;

     struct tracecut_symbol_param *next;
};
```

It has a `param_index` field that refers to the parameter or return value of the symbol to be captured. The `struct tracecut_param` refers to the tracecut parameter the symbol parameter is mapped to. This field is set when the user calls bind_to_param or bind_to_return function. Note that we are not storing the type information in this structure; it is contained in the `tracecut_param` field instead.

To store the tracecut parameters we have the `tracecut_param` structure which stores the name of the parameter and the type information. The type `aop_type` is from InterAspect framework.

```
struct tracecut_param
{
     /* Name to be used as identifier */
     char * name;

     /* Type of the parameter */
     aop_type type;

     struct tracecut_param *next;
};
```

As mentioned earlier we provide APIs to initialize the above structures. The name field is used as a key to identify tracecut symbols and tracecut parameters. This name should be used by the user to refer to a particular tracecut parameter or tracecut symbol in the regular expression as well as all the API functions.

6.2 Creating pointcuts according to the stored information:

Once we have all the information we can create the pointcuts related to the Tracecut symbols.

We make use of parameterized weaving provided by InterAspect to pass the `struct tracecut_symbol` pointer to `join_on` callback function at each join or weave location. The structure stores a list of `tracecut_symbol_parameter` to capture the required parameters at each join point. (`struct tracecut_symbol_parameter` holds information about the index of the parameter

and type of the parameter). These captured parameters will be passed to state machine transition functions from `tracecut-helper.c`. The auto generated code is written in `statemachine.c`. Both these files can be compiled separately with any GCC version (with or without plug-in support) to object files and these object files have to be linked with the target program in the linking process to get the advice code.

6.3 Managing state transitions:

In `tracecut_symbol` structure we have information about how to create the pointcut and what parameters to capture at its join points. We need to make use of the captured information to make the state transitions. From the function name we know what transition to make and from captured parameters we need to infer on what state machine the transaction should take place.

For achieving this, we have to keep track of state machines at the run time. Thus, a structure describing the state machine is required. The state machine consists of list of name-value pairs of attributes. The state machine is initialized at the symbol which specified as init symbol by the user. The need for name-value pairs arises from the fact it is not possible at every Tracecut symbol location that all the parameters are available (For example in `call_next` only the iterator is available). So in that case partial matching has to be done and to achieve that the name should be used to apply partial matching of attributes (that means attribute name is compared first and then its value). The state machine related data is stored in following structures.

```
struct state_machine
{
      int state_id;
      struct attribute *attrib_list;
      struct state_machine *next;
};

struct attribute
{
      char *name;
      void *value;
      struct attribute *next;
};
```

```
/* Global list of all state machine instances*/
struct state_machine sm_list = NULL;
```

The `state_id` field in the `struct state_machine` refers to the current state of the state. Additionally the structure has list of attributes with name-value pairs.

The file `tracecut-helper.c` provides the helper functions to create state machine, to add new attribute to the state machine, to make transitions on the state machines. We will explain the functions as follows.

```
void create_state_machine ()
```

This function creates a new state machine that is added to the global list. As we cannot capture return value from one advice function to the advice immediately following it, we have to store the pointer to the state machine as a global variable. Also as we have to add variable number of attributes, we call `add_attribute_sm` successively.

```
void add_attribute_sm (char * name, void * value);
```

This function adds the attribute to the newly created state machine.

```
void add_attribute_to_attrib_list (char * name,
                                   void * value);
```

Similar to a global state machine, a global pointer to query attribute list is maintained. This list is used to query the state machine list in `make_transition` function. The above function adds a new attribute to the global list.

```
void make_transition (int sym, int is_create_sm , const
char *advice_param);
```

The `make_transition` function does the task of finding the state machine corresponding to the current global attribute list and making transition on each of them. If the state machine is newly created which is denoted by `is_create_sm` parameter is set to 1 and the state transition is only made on the newly created state machine.

The function assumes presence of function get_next_state which has the following signature.

```
int get_next_state (int state_id, int sym ,
                    int * is_final);
```

The state_id parameter refers to the current state of the state machine. The sym parameter refers to the character denoting the event. Each of the tracecut symbols is assigned a representative single character. This character is used as a key in the code generating the state machine. Also the single character makes the `get_next_code` simple. The third parameter `is_final` is passed by address to know whether the next state is a final state.

This function contains the code corresponding to the state machine. This code is generated by parsing the regular expression and converting it to DFA (Deterministic Finite Automata) in memory and writing the corresponding switch case for the memory. We have used source code of regular expression to DFA converter library developed by Russ Cox (Refer http://swtch.com/~rsc/regexp/regexp1.html [5]). We have modified the source code of the library to write the in-memory DFA as a C source code. So with each new regular expression the code for `get_next_state` function is dumped in the file `statemachine.c`.

Chapter 7

7 Results

7.1 Iterator Example:

We ran our iterator example having the following code snippet. Note that the last line contains the call_next done after the source is updated. Thus our tracecut catches this situation and calls `handle_bad_iterator.`

```
struct iterator *it = create_iterator (ds);
while ((data = call_next(it)) >= 0)
{
      printf("Data : %d\n",data);
}
printf ("Updating the source .....\n");
update_source (ds, str2);
call_next(it);
```

7.2 Detection of write to closed file handle example

We ran this example on bzip2 program. In this program we introduced fclose call before an fwrite and detected the hazardous write and exited the program. (Otherwise the program gives a segmentation fault if it writes on a closed file handle.) With instrumentation on we ran the program to compress and decompress an 8.1 MB PDF document and following are the results.

Running Time:

With Instrumentation		Without Instrumentation	
Compress	Decompress	Compress	Decompress
4.319	3.608	4.18	2.613
4.182	3.488	4.214	2.606
4.334	3.24	4.186	2.599
4.233	3.5	4.161	2.593
4.267	3.294	4.202	2.637
4.205	3.456	4.171	2.608
4.192	3.272	4.216	2.674
4.186	3.415	4.163	2.599
4.254	3.225	4.185	2.662
4.258	3.284	4.175	2.631
4.243	3.3782	4.1853	2.6222

Binary Size:

With Instrumentation	Without Instrumentation
168870	162710

Memory Usage Size (in KB):

With Instrumentation		Without Instrumentation	
Virtual	Resident	Virtual	Resident
9060	7014	9060	7000

Chapter 8

8 Conclusion and Future Work

Thus we have presented the tracecut concept that takes a pattern on program trace in form of regular expression and successfully matches it on the sample programs. However, there are some limitations to this approach and here are some features we would like to add to current implementation.

1. Support for keeping track of scope of variables. By using this we can keep track of open file handles which are not closed when all references to those go out of scope.
2. As a result of implementation of this project, it was realized that some more features are needed from InterAspect. For example, we need to be able to capture the address of parameter captured not just their value. This will enable us to take corrective actions more elegantly. This feature can be useful in our example of file handle where if we could catch the file handles address then we could replace that closed file handle with an opened file handle in the memory address; and the program would run unharmed.
3. Taking different state transitions for the different ranges of the same parameter of the same function. For example, setuid(0) sets effective user id to 0 that means it enters a privileged mode and non zero parameter means non privileged mode. We want to have different state transitions for call the same function.

Bibliography

[1] Chris Allan , Pavel Avgustinov , Aske Simon Christensen , Laurie Hendren , Sascha Kuzins , Ondřej Lhoták , Oege de Moor , Damien Sereni , Ganesh Sittampalam , Julian Tibble, Adding trace matching with free variables to AspectJ, Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, October 16-20, 2005, San Diego, CA, USA

[2] Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A. Smolka, Scott D. Stoller, and Erez Zadok Aspect-Oriented Instrumentation with GCC In Proceedings of 1st International Conference on Runtime Verification 2010, St. Julians, Malta

[3] Thomas Cottenier, Aswin van den Berg , Tzilla Elrad, Stateful aspects: the case for aspect-oriented modeling, Proceedings of the 10th international workshop on Aspect-oriented modeling, p.7-14, March 12-12, 2007, Vancouver, Canada

[4] Clara: a framework for Statically Evaluating Finite-state Runtime Monitors (Eric Bodden, Patrick Lam, Laurie Hendren), In 1st International Conference on Runtime Verification (RV),pages 74–88, Volume 6418 of LNCS, Springer, 2010.

[5] Regular Expression Matching Can Be Simple And Fast by Russ Cox http://swtch.com/~rsc/regexp/regexp1.html