Report on study of research tools for embedded software development at Schlumberger

Kedar Swadi

March 14, 2006

1 Introduction

Embedded software is often executed on platforms that are safety-critical and where errors are difficult to detect and recover from in real time. It is therefore necessary to have a software development methodology that allows the detection and prevention of errors before it is deployed.

Errors in software can either arise from the incorrectness and incompleteness of requirement specifications, or due to the software violating some predefined safety policies that it must adhere to. For example, safety policies might prevent accessing null pointers or arrays at out-of-bounds indices, or casting pointers to incompatible types. Verifying that software correctly implements specifications is tedious for humans, and is automated verification techniques can be intractable (e.g. theorem proving). On the other hand, ensuring that software is safe can generally be performed statically (before software is executed). A safety-driven software development process would therefore allow the detection and prevention of as many errors as is practically possible.

This report describes my experience studying and using research tools for safety-driven software development in an industrial setting at Schlumberger Product Center, working with the Software Metier. The aims of this study were to 1. Determine the effectiveness and benefits of research tools in the Schlumberger setting, 2. Recommend tools and practices appropriate for the Schlumberger software development environment, and 3. Give a presentation to the software community on the findings.

2 Study Description

The study was carried out in a four-week period, and consisted of using four tools on a software project that involved the control of a motor. This software was chosen primarily because it was both small enough to be studied and worked with (given the short time period), and because it was real software that was being used in embedded settings in Schlumberger tools. The software had the following characteristics:

- It was written in Analog Devices C — a dialect of C that has a few nonstandard (non ANSI C) constructs and types
- It consisted of roughly 7000 lines of code
- It had no dynamic memory management, in line with the policy of the software group to avoid it because it is error prone, expensive, and requires OS support which is not always available in embedded environments
- It handled only one interrupt, and had no multi-threading
- It had low-level memory access code to interface with an external memory subsystem
- A majority of the code (and also the runtime) involved mathematical computations to determine a new state for the motor.

Using this software as the case study, four tools were considered for their benefits to the software development process. These are

- Devil : An interface definition language designed to specify a high-level interface for communication with memory-mapped devices.
- CCured : A memory safety analysis tool for C programs.
- Cyclone : A safe dialect of C.
- Blast : A model checker for C programs.

I describe below my experience using each of these tools in detail. These experiences are undoubtedly influenced by the choice of the case study software, and might have differed to if other software projects had been considered.

2.1 Devil

Tool description Programs that perform I/O communication must often rely on documentation provided by hardware manufacturers to determine the exact interface to I/O devices. Unfortunately, such documentations are often incomplete, ambiguous, and sometime, even incorrect. Devil [**?**, **?**] is an *Interface Description Language* designed to allow programmers specify a high-level interface for communication with hardware devices such that it is statically possible to verify critical safety properties of the device interface code. Thus, in addition to allowing the programmer to write a more maintainable code and ensuring that a hardware interface is semantically consistent, it also serves as a valuable documentation tool that can be reused in other applications.

Devil has three levels of abstractions at which communication with a hardware device can be viewed.

figure=devil.eps,height=5.5cm

1. *Ports* are the lowest level of interaction with a device, and represents the physical memory address which is read from or written to for interaction with a device. A device typically has several points of communication (or several memory addresses that it uses), and each of these is termed as a *port*.

2. *Registers* are the level at which interaction with a hardware device takes place, and allow related ports to be grouped. In particular, some devices assign different ports for reading and writing values, while other devices use only one port. A *register* allows a programmer to get a unified view so she can read from and write to the same register without having to know the port-level implementation.

3. *Variable* To minimize the amount of ports for communication with a device, registers often group multiple independent values into one data unit (byte or word). For example, a mouse might interact with a computer by setting three different bits in a single byte depending on which mouse button has been pressed. At the high-level, the programmer interacts with the device by accessing or modifying these values. In a Devil specification, a *variables* is used to represents each of these high-level values. A programmer using Devil only needs to refer to variables, while the code for all the low-level implementation details is generated by Devil from a given device specification.

The following example illustrates the levels of abstractions described above.

In figure **??**, 8-bit memory addresses 100 and 101 are the points of communication, or *ports* for a device. The device uses address 100 to read data (get input) and address 101 to write data (send output). A *register* abstracts away from this detail, and therefore, reading from the register would fetch the contents of memory at 100, and writing to the register would write into the memory location 101. This register however contains three independent values, with the first value taking up three bits, the second taking up two, and the last taking up three bits. The abstraction of a *variable* allows the programmer really must interact with the device without having to manually perform bit-level operations to get and set the values.

How did Devil affect the software development process?

In the absence of Devil, the writing software to interact with a hardware device was as follows

- Read and understand documentation from hardware manufacturer
- Express interface constants such as addresses and bit masks in header files
- Write interfacing code interspersed in C files on a by-need basis

In this process, maintenance of software is made difficult because understanding the device interface involves inspecting all the code. It also involves extensive testing to ensure that hardware interface is properly coded to be semantically consistent. This might not always catch all errors. Finally this process would have to be repeated for every new project (even if it used a similar hardware device), and this reduced the opportunity for code reuse.

On the other hand, software development process using Devil involves these steps:

- Derive hardware interface specifications from documentation provided by the manufacturer
- Encode the derived specification in Devil
- Use the Devil compiler to generate a header file that has macros to interact with device *variables*
- Write application code using macros in the header file generated in the previous step

How was Devil used? Devil was used in the project to reduce the programming overhead of bit manipulation involved in communicating with an external memory subsystem via memory-mapped registers.

In this project, the communication with the memory subsystem is through a set of registers that are memory-mapped onto a set of locations starting at two base addresses (0x3fe0-0x3fe6, and 0x3fef-0x3fff). Many registers have sets of bits assigned for different values or flags. For example, one 16-bit register (for the program flag data) has the following layout:

bits 15-8	unused
bit 7	ADC flag
bit 6	tp3 flag
bit 5	tp2 flag
bit 4	tp1 flag
bits 3-0	mux channel selector bits

In the original C header files, these flags are written using carefully computed constants for performing masking operations as shown below for setting/resetting the bit for the ADC flag

```
#define SET_PF7_ADC_CONVST      0x0080
#define RESET_PF7_ADC_CONVST    0xFF7F
```

The Devil specification for the device was obtained by inspecting the definitions of such constants, and their uses in the .c files. The above two lines are expressed in Devil code as

```
variable flag_data_adc =
    REG_PROG_FLAG_DATA[7] : {SET_ad <=> '1', RESET_ad <=> '0'};
```

Based on this Devil specification, two C macros (`get_flag_data_adc` and `set_flag_data_adc`) which read and write the value of this bit are generated by the Devil compiler. The original C code that involved communication with the external memory was then rewritten to use these macros. The resultant code was undoubtedly easier to read and is expected to be easier to maintain.

We now describe our experience using Devil.

How easy was writing Devil specifications? Using Devil-generated code and retrofitting it onto existing code in this project presented the following practical problems: The C code has one common header file that consists of the constants used for this device. However, the uses of registers used to communicate with the external memory subsystem are spread over many `.c` and other `.h` files. The specification for this subsystem had to be be inferred by carefully reading and documenting the exact use of each of the registers (and their bits) in all these files. Furthermore, it was necessary for me to talk to the original code author to ensure that the Devil specifications that I had derived from studying the code were correct.

How easily was Devil integrated with existing code? Rewriting the C files to use Devil-generated macros was fairly simple. However, some of the Devil-generated macros could not be used, and gave errors. I had to manually fix these macros to be usable, in particular the macro to set a whole structure is generated with a different name than that given in the documentation. Since Devil assumes the Unix environment, it used the `inw` and `outw` functions which are not available in the Analog Devices C compiler used at Schlumberger. I had to manually write macros to define `inw` and `outw`. Devil generated `.h` files also use the types `u16` and `u32`. These are not defined in the Analog Devices C compiler either and I had to redefine them too. After these small changes, the new code compiled and ran in the Visual DSP++ software development environment without any problems.

How applicable was this tool? While there are many lines of code that involve bit-manipulation, the overall percentage of efforts spent in writing this code is not significant in comparison to the whole project. Similarly, most of the debug time was spent in removing errors in arithmetic calculations. There were also no chances to try out the advanced features of Devil (such as register serialization, automata-based addressing mode, block transfer) given the relative simplicity of the device interface that this program used. I was told, however, that other projects have very similar interfaces with external memory subsystems, and the specifications for this project can be fruitfully reused in the other projects.

How did it affect performance? Performance was measured using the changes to the size of the code. The code sizes compared thus:

Memory Segment	Original	Devil	Difference
mem_code	0xd92	0xe0b	121
mem_data1	0x252	0x25e	12

Devil thus came at a code size cost. In this particular project, however, most of the communication with the external memory subsystem is done in an initial

setup phase, and not in the main program loop, and so it does not affect the runtime significantly.

2.2 CCured

CCured [**?**, **?**] is a tool for analysis and re-engineering of C programs to guarantee runtime safety. It achieves this using a pointer-based analysis, which works by classifying pointers by observing the context in which they are used, and then ensuring that all the uses of pointers are safe. It optionally inserts code to ensure the safety of potentially unsafe pointer operations, and offers better analysis and smaller runtime overhead than some other current commercial tools.

The pointer analysis classifies pointers into three categories, *safe*, *sequence*, and *wild*. Pointers in the *safe* category are those that are not subjected to either arithmetic or casting. It is therefore sufficient to only ensure that such pointers are non-null when they are dereferenced. *Sequence* pointers are those that are used to iterate through arrays. In addition to requiring them to be non-null, we must also ensure that they are restricted to the bounds of the array. Sequence pointers that are used to traverse the array in only the forward direction require an upper-bound check, while those used in a bidirectional manner (either traverse forward or backward in the array) require both upper and lower bound checks. Finally, the *wild* pointers include all other pointer uses (e.g. casting), and in addition to nullity and bound checks, they also require knowledge of the data that is being pointed to. This is necessary to ensure that any casting that is performed will result in pointing to data that is compatible with the type that it is cast to.

A small example illustrates the use of CCured. Consider a function `addAll` (as shown below) that returns a sum of all elements of an array of length `len` and starts at `p`.

```
int addAll(int * p, int len){
   int sum = 0;
   for(;len >= 0; len--, p++)
   {sum += *p; }
   return sum;
}
```

In this function, we must ensure (a) that the increment `p++` in the `for` loop is within array bounds , and (b) that the dereference operation `*p` is safe. This is achieved by CCured by making a new structure to represent forward-traversing pointers such as `p`, given by `fseq_int` shown below.

```
typedef struct fseqp_int {
   int  * _p ;
   void * _e ;
} fseqp_int;
```

In this structure, _p is used to point into the array, and _e acts as a sentinel to check that _p never points to a location beyond what _e points to. Using this structure, CCured *cures* the function addAll to give the function addAll_f shown below. This function is guaranteed by the type system of CCured to be safe and free from runtime memory errors.

```
int addAll_f(fseqp_int p, int len)
{
  int sum;
  sum = 0;
  while (len >= 0) {
    CHECK_FSEQ2SAFE
       (p._e, p._p,
        sizeof(int),sizeof(int),1,0);
    sum += (*(p._p));
    len--;
    CHECK_FSEQARITH
       (p._p, sizeof(int), (p._p + 1));
    p._p = p._p + 1;
  }
  return sum;
}
```

Notice the insertion of two checks CHECK_FSEQ2SAFE and CHECK_FSEQARITH that ensure that the pointer dereferencing is safe, and that the pointer arithmetic is safe respectively.

How did CCured affect the development process?

In the absence of CCured, there was no compiler-supported technique to guarantee safe uses of pointers. It was therefore the programmers responsibility to ensure that all pointer uses and manipulations were safe. In this process, errors that arise due to pointer misuse cannot be found statically, and debugging them is difficult and time consuming. Commercially available tools such as Purify are sometimes used to detect memory errors, but this is not a part of the standard software development process.

CCured was used to find any possible unsafe pointer uses in the project code. The process of using CCured involved first preprocessing all the source files in the code and merging them. CCured then uses its analysis to determine any possibly unsafe pointer uses. If such uses are found, CCured instruments the code to make these uses safe. The instrumented code is then compiled to produce the final executable binary. CCured was used in the software development process by simply replacing the Visual DSP++ compiler that was originally used for this project by the CCured compiler. Output given by the CCured compiler was inspected to determine if there were any erroneous or possibly unsafe pointer uses. The original code was then changed to take into account these uses.

We now present our experience using CCured.

How easy was it to use CCured? Using the CCured system is easy since it requires no new language to be learnt. I started by writing a makefile manually. This step is not necessary in the Schlumberger environment, because the Visual DSP++ build function performs dependency analysis and automatically generates its makefile equivalent. Therefore, knowledge of writing makefiles is necessary to use this tool. While there is some documentation on using CCured, it does not explain in sufficient details the process of writing makefiles so that a user can understand what exactly all the options do and how to correctly invoke them. I had considerable problems being able to specify the options for browser feature that allows an easier inspection of the CCured analysis output. On Unix, CCured outputs a xxx.cured.c and xxx.cured.i from file xxx.c which shows the cured output. On Windows (under Cygwin) these files are not generated (using the same makefile options)

How easy was it to integrate CCured with existing code? The original C code has some inlined assembly code. While CCured understands (to some extent) `asm` directives, it assumes x86 assembly. I therefore commented out this code (rather, wrote `#define asm(x) /* nothing */`). The Schlumberger Visual DSP++ compiler also makes references to some non standard types. For example, the type `wint_t` is not understood by CCured. I defined this type to `int`. The system also makes some preprocessor definitions (e.g. `__ADSP21XX__` to define the processor type) that are not made explicit in the .c or .h files. This prevents the correct preprocessing of some of the system header files (e.g. sysreg.h) It required a manual inspection of the code and I had to determining all such variables that needed to be defined that would allow the system files to be correctly processed.

How applicable was this tool? I noticed that pointers were used sparingly in the code. There is no dynamic memory management (all arrays are fixed-sized and statically declared). This works well for the Schlumberger applications because the sizes of data to be processed are known before hand, and all subsystems only interact through fixed sized messages. Therefore, the code did not present sufficient opportunities to investigate the full potential of the CCured system. On the other hand, it was instructive to compare the warning messages given by CCured with those given by the Visual DSP++ system. In the comparison, I found that CCured was able to infer safe uses of pointers that Visual DSP++ was not (and therefore gave no warning messages for the same), but also flagged some uses of constant values as pointers (for memory mapped registers) as potentially unsafe. The fact that these assembly instructions were not for x86 architecture meant that CCured could not be used to inspect the assembly statements. CCured now also has the ability to work with C++ files. For larger projects which might involve more dynamic memory management uses, CCured will be of much more help.

How was the performance affected? Compiling with CCured as opposed to Visual DSP++ takes similar amount of time. Since some code was instrumented by CCured, the runtime performance will be slowed. This might be problematic since many applications involve interrupt handlers that must finish execution in some predetermined time period. CCured therefore is better used more as

a tool to point out potential problems than to fix them. This is also necessary because of the fact that the programmer must inspect the generated assembly code sometimes to ensure some conditions (such as the optimal assignment of variables to registers for execution speed) and link it to the source code which he can inspect.

2.3 Cyclone

Cyclone [**?**] is a general-purpose, C-based, statically typed, safe language. Cyclone is intended to be a safe substitute for C, and allows C programmers to use many of the popular (though potentially unsafe) C idioms in a safe manner.

Cyclone's type system ensures that programs are safe by imposes some restrictions on uses of pointers. For example, it ensures that pointers are never null if they are dereferenced. Pointer usage must be disciplined in that it must ensure that pointers are initialized, and that pointer arithmetic must be restricted to obey array bounds. Casts must be performed in a manner that can be shown to be safe. No `goto` into scopes is allowed, and the use of functions `setjmp` and `longjmp` is not supported.

Some of the extensions in Cyclone include: "Fat" pointers that involve runtime bounds checking for safe pointer arithmetic, tagged unions for type varying arguments, polymorphism to replace uses of `void *` so that they are safe, implementation of variable-argument functions using fat pointers instead of the potentially unsafe `vararg` mechanism used in C, use of exceptions replace some uses of setjmp and longjmp, and an optional automatic memory management scheme for applications that can use garbage collection without any problems arising due to the associated runtime overheads.

Cyclone allows a programmer to specify different kinds of pointer types with its extended syntax. For example, to specify that a function `getc` takes a non-null file pointer `fp` and returns an integer, we write `int getc (FILE @ fp);` (in contrast to just `int getc(FILE * fp);` in C). This ensures that `getc` can only be called for pointers that can be statically shown to be non-null. As a result, `fp` itself can be guaranteed to be non-null in the body of `getc`. Array pointers are written using fat pointers. For example, a function `strlen` that takes a string implemented as a pointer to characters and returns its length is written in Cyclone as `int strlen (char ?);` (instead of just `int strlen (char *);` in C). A fat pointer holds information about the length of the array, and performs bounds check at each array access.

The following example illustrates how the `strlen` function would be written in C and in Cyclone.

The C implementation shown below takes a pointer argument `char *s`, assigns it to a local pointer `p`, which is then used to iterate through the array incrementing `i` which holds the length, until the end-of-string character is found. Since this character might lie outside array bounds, this code is not safe.

```
int strlen(const char *s){
   const char *p = s;
```

```
   int i = 0;
   for (i=0;*p;i++,p++){
   }
   return i;
}
```

In contrast, the Cyclone implementation shown below takes a fat pointer `char ?s` and ensures that `p` the local pointer is never accessed beyond the array bounds (using its array-length information assigned to the variable `n`).

```
int strlen(const char ?s) {
   const char *p = s;
   int i, n;
   n = numelts(s);
   for (i=0; i<n && *p; i++,p++) {
   }
   return i;
}
```

The tagged unions in Cyclone allow us to ensure that the variants in the union types are never used in an unsafe manner.

Consider the C code below:

```
struct arg {
  union {
     int i;
     char * s;
  } t;
  int tag;
};
void pr (struct arg x) {
   switch (x.tag) {
      case 0:
        printf (``%d'', x.t.i);
        break;
      case 1:
        printf (``%s'', x.t.s);
        break;
   }
 }
```

In this code, the structure `arg` has a union `t` (of an integer and a char pointer), and uses a `tag` to keep track of which variant it currently has. The programmer must first implicitly make an assumption that a tag of 0 implies that the union carries an integer value, and that a tag of 1 implies that the union carries a char pointer value. Furthermore, while writing code such as the function `pr`, which uses the union, the programmer must be careful to ensure

that (in the `switch` statement) the tag value is properly matched with the variant that is used. It is easy to make a mistake such as

```
      case 1:
        printf (``%s'', x.t.i);
        break;
```

where the tag (= 1) and the variant (= i) do not agree. Instead, in Cyclone, one would write

```
tunion t {
   Int(int);
   Str(char ?);
};

void pr(tunion t x) {
   switch (x) {
     case &Int(i):
       printf("%d",i);break;
     case &Str(s):
       printf("%s",s); break;
   }
}
```

Here, each variant in the union is explicitly tagged (using tags `Int` and `Str`). The switch statement can then pattern match using these variants as shown above which would make it impossible to make the error shown above.

How did Cyclone affect the process?

The existing development environment used C which did not ensure that pointer usage was safe. Therefore, the programmer was burdened with the responsibility to ensure that all pointer uses and manipulations were safe. This is tedious, and any programmer errors cannot be found statically, and debugging them is difficult and time consuming.

Using Cyclone would allow us to be able to have a safe version of the motor-control software. This involved rewriting all the sources in Cyclone (and changing the pointer types as necessary) to ensure that the program is safe to run. The Cyclone compiler has a command-line porting option (cyclone -port) which was used to first convert the C files to Cyclone files, and then the Cyclone compiler was invoked instead of the C compiler to compile these files.

How easy was it to use Cyclone? The Cyclone language is mostly easy to learn and use. The difficult parts seem to involve the region annotations for dynamic memory management. However, since there was no dynamic memory management in this project, I had no opportunity to use regions. The porting option did not output too many changes. In fact, the only change required me to annotate some functions with an extra region polymorphism argument. There were also some bugs with the Cyclone system that I managed to get fixed by communicating with the author.

How easily did the C code to convert to Cyclone Not all the C code could be translated to valid Cyclone code. The C code has some inlined assembly code. Cyclone does not understand the `asm` directive, and therefore functions containing inlined assembly could not be translated. Cyclone also cannot handle some of the pointer uses in the C code. As a concrete example, the project requires some absolute addresses to be written to and read from. Cyclone deems this to be unsafe and therefore functions involving such code could not be translated to Cyclone. Cyclone also uses its own header files and libraries. This might prove to be a problem when trying to convert C code which relies on preexisting libraries to Cyclone.

How applicable was this tool? As with CCured, the absence of dynamic memory management and (relatively) sparse pointer usage means that the code does not present any significant opportunities to investigate the full potential and benefits of Cyclone. Furthermore, the fact that Cyclone does not allow low-level absolute memory location access would be an impediment to writing embedded software where such operations are routinely used. A balance might be found by developing a part of the software in Cyclone, while the low-level operations might be written in a (throughly tested and debugged) C library. The absence of multi-threading in the test application also prevented the use of Cyclone to statically check for safe locking for shared variables. I was informed that other embedded software does use multi-threaded programs, and the use of Cyclone in the development of such programs would significantly benefit from the use of Cyclone.

How did it affect performance? It was not possible to compile the code using Cyclone because some critical parts of the code are not allowed as legal by Cyclone. It is not clear to me as to how this can be overcome (mainly the low-level memory access). Even assuming that these parts could be made to work, Cyclone would still slow down the performance since it inserts extra safety checks into the code.

2.4 Blast

Blast[**?**] is a C-based model checking tool. A Blast user first specifies program properties to be checked. The model checker ensures that program respects these properties. If not, it shows where problems might lie. A small example shows the basic idea behind model checking with Blast.

Consider the two implementations of `foo` shown below. They are identical except for implementation (1) having `x = x - y`, while implementation (2) has `x = y - x`. Assume that we would like the program to have property that `x` should be greater than zero after the subtraction. In both of them, we check this property using the statement `assert(x > 0);`. In model checking, the basic idea is to statically check that assert statements do not fail for any execution path taken by the program. In implementation (2), if the `if` branch is taken, then we encounter a false assertion, and therefore this implementation would be marked as erroneous by a model checker.

```
#include <assert.h>
int f1(int x, int y) {
   if (x > y) {
     x = x - y;
     assert(x > 0);
   }
}

#include <assert.h>
int f2(int x, int y) {
   if (x > y) {
     x = y - x;
     assert(x > 0);
   }
}
```

In Blast, we use this same basic idea to check for much more complicated properties of programs. Consider, for example, the following function `f`.

```
 fract16 f (void) {
  ...
  disable_intr();
  ... ...
  enable_intr();
  ...
  while(cond2) {
    ...
    switch (x) {
      ...
      case (2): ...
                disable_intr();
                ... ... ...
                enable_intr();
                ...; break;
       ...
    }
    ...
    disable_interrupts();
    ... ... ... ...
    enable_interrupts();
    ...
  }
}
```

We would like to to satisfy the property that it starts with interrupts being enabled, and only has alternating calls to `enable` and `disable`. This ensures that we do not enable (or disable) an already enabled (or disabled) interrupt.

This property can be captured using the state machine shown below: In Blast, we encode this property using the following specification:

```
global int enabled = 1;

event {
   pattern {enable_intr (); }
   guard {enabled == 0 }
   action { enabled = 1; }
}

event {
   pattern {disable_intr (); }
   guard {enabled == 1 }
   action { enabled = 0; }
}
```

The variable `enabled` says which state we are in (0 or 1). The `event` construct specifies the transitions. For example, if we are in state 0 (`guard{enabled == 0}`), and we encounter the statement `enable_intr();`, then we transition to state 1 (`action{enabled = 1}`). Using this specification Blast model checks the function `f` to ensure that it satisfies the required property.

Blast can be used to specify and check for various properties including those that concern thread safety, reachability, dead-code analysis, secure function call sequences, and consistency of parameters in function calls.

How did Blast affect the software development process? The existing software development process did not make use of model checkers to verify software; C sources were compiled into binaries, and all verification was performed by manually testing this binary. Using Blast, the software development proceeded as follows: First, it was necessary to identify what properties or invariants were desired at runtime. (In our case, the property ensured that the calls to the enabling and disabling of interrupts were properly sequenced.) A Blast specification file was then written to encode these properties. Then, a program in the Blast software (*spec.opt*) was used to instrument the C code to facilitate checking of the desired properties. Finally, the Blast model checker is run on this instrumented code, and flags violations of the properties if any are found. If Blast is unable to generate any errors, then the C source code is guaranteed to comply with the properties. The uninstrumented C source code then can be processed further using the existing tools.

How easy was it to use Blast? Using Blast involved many problems. Although I was able to install Blast on Windows under Cygwin, it did not work correctly. I was not able to reproduce the results for tutorial examples that come as a part of the Blast distribution. Furthermore, attempts to contact the authors to resolve this problem were futile. These examples did run on the Linux installation as described in the manual. Since Blast considers undefined (but declared) functions as black boxes that do not effect the property under

consideration, it was possible to verify only parts of the project. Blast did not scale up when more files were added (into the instrumented code). Blast gives an error message and then quits even when the new files do not at all affect the property under consideration. The documentation did not have any detailed descriptions of the error messages. My attempts to overcome this failure by using and changing Blast command-line options were futile. The options are not well-documented, and many are not implemented. Furthermore, a large part of the Blast query language does not seem to have been implemented in the Blast-1.0 version that is available for download. It might also be difficult for programmers not used to writing logical formulae that the Blast query language uses to express property specifications. This may present a steep learning curve for Blast, and any other model checkers, in general. Since the Blast graphical user interface did not work for me, it was very difficult to parse the error traces and link it to the erroneous parts in the source code. I see this as another potential difficulty in using Blast.

How easy was it to integrate Blast with the project code? The Blast distribution contains a `spec.opt` binary which generates an instrumented C file from the property specification and C sources does not work when nonstandard headers are used. C compilers have a `-I` option to allow the specification of paths which must be searched for header files. This option does not work with `spec.opt`, and as a result, the user must manually copy all headers (not in the standard include path to the current path). No other special effort was necessary while integrating the existing C source code with the Blast framework.

How applicable is this tool? Since model checking can be used to ascertain that the code respects a wide variety of properties, there are many applications for Blast in the Schlumberger program development process. For example, properties such as function reachability, dead code elimination, (non) occurrence of certain safety critical function call sequences, safe multi-threading, and correct interrupt handling are all relevant to the embedded software programs being developed at Schlumberger.

Does it affect performance? Since Blast does not involve modifying source code, it does not affect the runtime performance.

3 Conclusions

The following two tables summarize the experience using each of the tools with respect to a variety of concerns.

	Purpose	Applicability	Learning Curve	Ease of Integration	Effect on Performance
Devil	Device Interface	High	Medium	Easy	Slowed
CCured	Memory Safety	Low	Low	Medium	Slowed
Cyclone	Memory Safety	Low	Low	Medium	Slowed
Blast	Model Checker	High	Medium	Easy	N/A

	Level of Support	Level of Maturity	Level of Automation	User Expertise	Environment Required
Devil	N/A	Medium	Low	Medium	Cygwin
CCured	N/A	Medium	High	Low	Cygwin
Cyclone	Good	Medium	Low	Low	Cygwin
Blast	Bad	Low	Low	High	Linux

We noticed that there were more opportunities to exploit the advantages that Devil and Blast offer than CCured and Cyclone. This was primarily because the embedded software developed at Schlumberger avoids the use of dynamic memory management (not so much because of the complications involved as much as because of the fact that either the hardware is often not sophisticated enough to support it, or because memory requirements can be determined statically). On the other hand, CCured and Cyclone were easier to learn and required lower user expertise, because Blast and Devil both required the user to be familiarized with new languages and notations. CCured and Cyclone also proved to be more difficult to integrate with the existing framework because they affected the main C sources directly (Cyclone involved using another language while CCured involved instrumenting the C source code). On the other hand, using Devil required a preprocessing step (the generation of Devil macros) and using Blast was a post-processing step, both of which did not interact with the main C software development process. Other than Blast (which did not have any effect on the final executable), all other tools involved the insertions of additional runtime checks to ensure safety. This impacted negatively on the runtime performance.

From the software engineering point of view, since the tools were freely available implementations of research ideas, they had a low level of support and were not as mature as the commercially available tools. It would require some time for these tools to mature (remove existing bugs and allow easier integration with popular commercial tools) and be practically usable. Other than CCured, all other tools necessitate significant changes to the software development cycle that involve manual input, and therefore affect the level of automation of the process.

3.1 Response to this study

I presented the results of this study to members of the embedded software development group at Schlumberger. A brief discussion following this presentation, where members voiced their opinions and concerns.

A primary concern was regarding the maturity of these tools and the level of support that would be available. Since many of these tools had incomplete documentation, and did not always behave as expected, there was a concern that the current and future use of these tools could be hampered due to the lack of support for them.

Another concern was how easily the tools could be used with the existing systems at Schlumberger. As a concrete example, I was told that most software

projects have an automatic generation of makefiles which do not allow analysis tools to be conveniently pipelined into the software development process. Similarly, software is sometimes dependent on particular versions of development tools (e.g. the VC++ compiler versions), and it is necessary for any newer tools to conform to the requirements of these tools.

Several members also felt that memory safety was not an important enough issue to justify either the use additional software or a different language in their development process. I was informed that some of the compilers that are used at Schlumberger already provide options to insert checks for null pointers and array bounds, and that most applications are written to avoid dynamic memory management, which reduced the opportunities to get much benefit from using these tools. I was also informed that it is their practice to use the commercial tool Purify to ensure the memory safety of their programs.

There was also a strong preference for analysis tools that do not modify the code in any manner. Some members told me that they would not like to lose control over their code by using tools that perform automatic instrumentation of code, since this makes it harder to debug, analyze, and maintain the code.

In relation to Devil, I was informed that some companies such as Texas Instruments already provide programmers the type of high-level C interfaces that are generated by Devil. However, a large fraction of hardware that is used comes with no such interfaces, and that Devil would certainly be of some interest in projects using such hardware.

3.2 Future Collaboration

Based on the feedback I received on my presentation, I feel that the embedded software group members were most excited about using tools such as Devil and model checkers (though not Blast specifically). I have identified the following next steps in our collaboration with Schlumberger.

- Study other software projects at Schlumberger that might involve more opportunities for exploiting the benefits that these tools can yield. Since other software developed at Schlumberger involves advanced features such as dynamic memory management, multiple interrupt handling, and multi-threading, they could provide an ideal test suite against which the research tools considered could be evaluated to their full potential.

- Study other research and commercial tools to examine how they can aid the software development process at Schlumberger. Research tools such as NDL and Spin offer benefits similar to Devil, while other commercial tools such as SVM could also be considered for their benefits in model checking.

- Concentrate on interface definition languages and model checking tools to come up with implementations of these tool that are specialized to the software development process and environments at Schlumberger. In addition to a research component, this will also involve an engineering

component that involves working out the details of a seamless integration with the existing development environments.

4 Acknowledgments

I would like to acknowledge Thierry Simien and Shyam Mehta for their help in initiating the collaboration and this study, to Maheswar Gattupalli and Sridhar Sana for their help with many matters (technical or otherwise) throughout the study, and Walid Taha for all his inputs and direction.

References

[1] J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. Ccured in the real world. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 232–244, San Diego, CA, June 2003.

[2] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[3] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C, 2002.

[4] F. Merillon, L. Reveillere, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming, 2000.

[5] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[6] L. Reveillere, F. Merillon, C. Consel, R. Marlet, and G. Muller. A DSL approach to improve productivity and safety in device drivers development, 2000.