# Exploring the Acceptability Envelope [*]

Martin Rinard
MIT CSAIL
Singapore-MIT Alliance
Massachusetts Institute of Technology
Cambridge, MA 02139
rinard@lcs.mit.edu

Cristian Cadar
Computer Systems Laboratory
Stanford University
Stanford, CA 94305
cristic@stanford.edu

Huu Hai Nguyen
MIT CSAIL
Singapore-MIT Alliance
Massachusetts Institute of Technology
Cambridge, MA 02139
nguyenh2@cag.csail.mit.edu

## ABSTRACT

An *acceptability envelope* is a region of imperfect but acceptable software systems surrounding a given perfect system. Explicitly targeting the acceptability envelope during development (rather than attempting to minimize the number of errors, as is the current practice) has several potential benefits. Specifically, leaving acceptable errors in the system eliminates the risks and costs associated with attempting to repair the errors; investing fewer resources in less critical regions of the program and more resources in more critical regions may increase acceptability and reduce the overall investment of development resources.

To realize these benefits, the acceptability envelope must be both sizable and accessible. We present several case studies that explore the acceptability envelopes of the Pine email client and the SurePlayer MPEG decoder. These studies show that both Pine and SurePlayer can tolerate the addition of many off-by-one errors without producing unacceptable behavior. This result suggests that current systems may be overengineered in the sense that they can tolerate many more errors than they currently contain.

Our SurePlayer case study also shows that SurePlayer has *unforgiving regions* of code that must be close to perfect for the system to function at all. To effectively exploit the acceptability envelope, developers must be able to distinguish forgiving and unforgiving regions so that they can appropriately prioritize their development effort. In SurePlayer, the unforgiving regions occur in code that uses metadata to parse the input stream; the forgiving regions tend to access the data within each image. This result suggests that developers may be able to use relatively simple indicators to effectively prioritize their development effort.

**Categories and Subject Descriptors:** D.2.5 [Software/Testing and Debugging]: Error Handling and Recovery, D.2.4 [Software/Program Verification]: Reliability

[*]This research was supported in part by the Singapore-MIT Alliance, DARPA Cooperative Agreement FA 8750-04-2-0254, NSF Grant CCR-0086154, NSF Grant CCR-0341620, NSF Grant CCF-0209075, and NSF Grant CCR-0325283.

**General Terms:** Experimentation, Reliability

**Keywords:** Failure-oblivious computing, acceptability-oriented computing

## 1. INTRODUCTION

Given a specification, there may be multiple perfect software systems that implement that specification. The goal of much research in software engineering, program verification, and program analysis is to find ways of bringing the implemented software system as close to perfection as possible. The results of this research include program specification and verification systems [17, 24, 11, 5, 20, 19], program analysis systems [7, 4, 26, 3, 21], software development processes [25, 16, 12] and, more recently, bug finding systems [10, 28, 15]. In spite of this effort, it appears that, in practice, effectively *all* large software systems contain significant numbers of programming errors [14].

This fact is often taken as an indictment of current software development practices. But when one examines the evidence, it is hard to miss a clear indication that aspiring to perfection in software systems is of limited utility and may even be actively counterproductive. Consider that, despite their many errors, most deployed systems do a perfectly acceptable job of satisfying the needs of their users. Consider also that attempting to develop a perfect system places extreme demands on the developer, and that developers may be able to be more productive and efficient if their goal is merely an acceptable, rather than perfect, system. Finally, consider that one common way of attempting to move the system closer to perfection (removing programming errors) can involve significant costs, complications, and risks. Botched repairs can be a substantial problem in practice [8]; the system may even reach a point in which virtually *any* change causes more problems than it solves [18]. In effect, each system may have a finite modification budget, and spending this budget repairing errors that are not crucial to the system's acceptable execution may crowd out the ability to make other, potentially more worthwhile, modifications.

### 1.1 The Acceptability Envelope

We propose an alternate conceptual framework. Instead of evaluating a software system by considering how much it deviates from perfection, we propose to instead consider whether the system delivers acceptable (even if flawed) service to its users. The foundation of this approach is the concept of an *acceptability envelope*.

Every perfect software system is surrounded by a constellation of similar but imperfect software systems. It is conceptually (but not practically) possible to construct this constellation by starting with the perfect system, then injecting errors. Some of the software systems in this constellation will acceptably satisfy the needs of the users. We call the collection of all such systems the *acceptability envelope* of the perfect software system. Note that almost all successful deployed software systems are contained in the acceptability envelope of the perfect system that the developers initially set out to build.

## 1.2 Exploring the Acceptability Envelope

In principle, the acceptability envelope should present a much larger development target than the perfect system at the core of the acceptability envelope. Whether the developer is able to exploit the flexibility and additional options that this larger development target provides depends on the size and accessibility of the acceptability envelope target. The larger and more accessible the envelope, the more opportunities are likely to arise to exploit the additional flexibility to make the development process more efficient and effective.

We discuss two case studies designed to explore acceptability envelopes. These case studies inject off-by-one errors into the popular Pine mail client [1] and the SurePlayer MPEG video decoder [2]. Each program contains a certain number of places where it is possible to insert such errors (in our case studies, loop termination conditions). Our results show that it is possible to insert off-by-one errors in *almost all* of these places without causing the program to behave unacceptably.

For SurePlayer, some of the injected off-by-one errors caused the program to behave unacceptably. Upon investigation we were able to identify a property that tended to separate unacceptable errors (which disabled the program) from acceptable errors (which the program could tolerate) — the unacceptable errors access metadata that allows SurePlayer to parse the input data stream while the acceptable errors tended to process the data in each image.

## 1.3 Exploiting the Acceptability Envelope

These results suggest two characteristics of the relationship between current systems and their acceptability envelopes. First, current systems may be overengineered in the sense that they have many fewer errors than they can acceptably tolerate and may therefore be much farther inside the acceptability envelope than necessary. One obvious way to exploit the acceptability envelope is to simply leave acceptable errors in place and avoid the risks and potential complications of attempting to repair such errors. Another is to adopt, when appropriate, development methodologies that, in return for other advantages, produce systems with more errors.

Second, acceptability envelopes may contain unforgiving regions — areas of the program that must be close to perfect for the system to operate acceptably. To successfully exploit the additional flexibility that the acceptability envelope may provide, it is important to be able to distinguish unforgiving regions from more forgiving regions that can tolerate more errors. Instead of an unfocused development effort that directed resources equally to all parts of the program, developers could then focus their efforts more heavily on the unforgiving regions and less heavily on forgiving regions. This prioritized development effort may yield a better end result — a more acceptable system (characterized by a minimal number of errors in the unforgiving regions and more errors in the forgiving regions) obtained with reduced development effort and resources.

## 1.4 Increasing the Acceptability Envelope

Basic safety condition violations (such as null pointer or array bounds check violations) or assertion failures provide clear evidence that the system is not executing perfectly. If the development goal is to produce a perfect system, *fail-stop behavior* is a rational response to such a violation or failure — instead of continuing with the normal execution path, the program either throws an exception (which the program must handle explicitly if it is to continue) or simply halts and awaits external intervention.

But our results show that, *as long as we apply techniques that disable safety checks and enable the program to execute through violations and failures without disturbing the normal execution*, the acceptability envelopes of our two systems include many variants that violate basic safety checks. Disabling these checks substantially increased the ability of our two programs to deliver acceptable results in the face of programming errors. This increased resilience translated directly into a significant increase in the sizes of their acceptability envelopes.

Our results therefore highlight one important drawback of the aspiration to perfection in software systems — namely, this aspiration misleads systems and language builders to inappriopriately and systematically scatter self-sabotaging checks pervasively throughout the system. Our results show that these checks can unnecessarily and artificially increase the brittleness of the system and deny the system the inherent resilience that it would have otherwise enjoyed. We suspect that a significant part of the frustration users occasionally experience when they try to use today's existing brittle software systems can ultimately be traced to the presence of inappropriate checks in the system.

The techniques that we applied to eliminate these checks (a variant of failure-oblivious computing [23] and source code transformations to discard exceptions and programmer-provided checks) are relatively simple both in concept and implementation. They do, however, conflict with the prevailing philosophy that it is unsafe to let a computer system continue to execute in the face of evidence that it has experienced an error. A developer focused on eliminating as many errors as possible would never develop, apply, or even consider these techniques — they are designed to improve the system, but offer no prospect of ever making it perfect.

One intriguing aspect of our results is how clearly they demonstrate that acceptability envelopes can contain many variants that violate basic programmer expectations. One potential explanation for this phenomenon is that developers may try to reduce their cognitive burden by keeping the system within an *anticipated envelope* of executions that is 1) narrower than required to deliver acceptable behavior but 2) has properties that make the program easier to reason about than other acceptable programs outside the anticipated envelope. This explanation suggests that to realize the full potential of exploiting the acceptability envelope, it may be necessary to relax the extent to which we expect developers to understand the systems that they build. In particular, we may need to be willing to allow systems to continue to execute through unanticipated, poorly understood conditions in the expectation that they will nevertheless deliver acceptable behavior.

## 1.5 Maintaining the Acceptability Envelope

Software development has traditionally been seen as a branch of engineering. But for maintaining acceptability, the field of medicine might provide a more productive comparison point. A primary concern in both disciplines is to how deal with an existing large, complex, and often poorly understood system that may be operating suboptimally but is far from completely disabled. There is often no expectation that any theraputic activity will deliver a perfect system, and a large part of the focus is simply to preserve the existing desired behavior or function of the system in the face of changing goals or environments.

Seen from this perspective, it is clear that bug-finding tools can be counterproductive. In the same way that finding out about a medical condition may lead to a course of treatment that winds up producing a worse result than simply leaving the condition untreated (treatment errors account for tens of thousands of deaths every year [6]), so too may the discovery of errors lead to preventative maintenance that leaves the software system worse off than it would have been had the errors simply been left in place [18]. Successfully maintaining a software system in the face of known errors may therefore require both a way to distinguish unacceptable errors from acceptable errors and the discipline to avoid the temptation to attempt to fix errors that are better off left in place. A common problem with medicine is the tendency of specialists to lose the big picture as they focus on a problem within their area of speciality. So too may specialists in different parts of the software system lose the big picture as they maintain their part of the system. In both cases a larger, more holistic perspective may produce a better overall result.

## 1.6 Obtainable Perfection

One may wonder if it is ever possible or worthwhile to aspire to perfection. In some parts of the system, paradoxically in arguably the most complex parts of the system, perfection is not just obtainable but also desirable. The key difficulty in developing software systems is almost always scale, not inherent complexity. Most errors occur because of the difficulty of performing a huge number of straightforward tasks perfectly, because of misunderstandings between different developers operating on interacting parts of the system, or because of changes elsewhere in the system that make previously correct behavior inappropriate.

It is possible, and even desirable, to obtain perfection in small, complex, well-understood components of the system. Examples of such components include data structure implementations, encapsulated algorithms, and many standard libraries. These kinds of components typically have clear, precise specifications, stable interfaces, and are small enough for a single talented developer to build. Moreover, it is often easy to use such components as building blocks in multiple systems, which may justify the development effort required to make them perfect.

## 2. PINE

Pine is a widely used mail user agent (MUA) that is distributed with the Linux operating system. Pine allows users to read mail, fetch mail from an IMAP server, compose and forward mail messages, and perform other email-related tasks.

## 2.1 Methodology

In this case study, we worked with Pine version 4.44. We used the source files in the `pine` directory of the Pine source distribution package. This directory constitutes the core of the Pine system. It is composed of 30 C source files which together contain over 150,000 lines of code.

We started by identifying the `for` loops in Pine which contain an integer counter `q` and a termination condition of the form `q<expr`, `q>expr`, `q<=expr`, or `q>=expr`, where `expr` is an arbitrary integer expression. Our error injection mechanism transforms conditions of the form `q<expr` into `q<=expr`; conditions of the form `q>expr` into `q>=expr`; conditions of the form `q<=expr` into `q<expr`, and conditions of the form `q>=expr` into `q>expr`. We divide the possible transformations into two main categories: transformations that increase the scope of the loop, namely transformations that modify loops of the form `q<expr` and `q>expr`; and transformations that decrease the scope of the loop, namely transformations that modify loops of the form `q<=expr` and `q>=expr`.

We identified 330 such `for` loops, among which 226 increase, and 104 decrease the scope of the loop after error injection. Figure 1 presents the number of loops that we identified in each category.

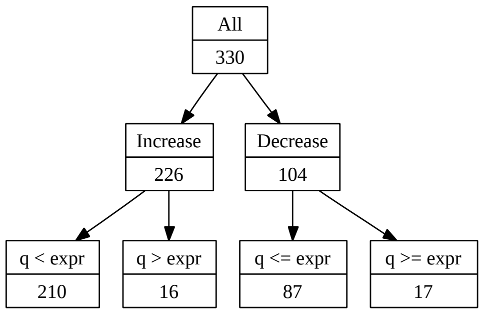

**Figure 1: For loops in Pine**

We constructed a variant of Pine that contained all these 330 off-by-one errors and compiled this variant with two different compilers: the standard gcc compiler to obtain a `standard` version, and a compiler that generates failure oblivious code to obtain a `failure-oblivious` version [23].

Our failure oblivious compiler performs array bounds checks. It tracks the allocated memory block to which each pointer should refer. If the program creates a pointer that points beyond the bounds of its memory block, then uses the pointer to attempt a read or write access, the generated code instead redirects the access to the first element of the allocated block. This mechanism enables programs to continue to execute through memory errors without memory corruption. Moreover, it tends to ensure that out of bounds reads access properly initialized data that satisfy the key consistency constraints of the data structure. We would have preferred a version that discarded out of bounds writes, but implementation limitations inherited from the base compiler made this technique impractical.

In addition, we also disabled all assertion checks, by simply commenting out the body of any procedures which terminate the program, such as the `panic` and `fatal` procedures.

To evaluate the acceptability of the Pine variants that we created, we designed an *acceptability test*, which tests the behavior of Pine on a set of standard mail management tasks:

1. **Main Menu**: We start Pine and test whether the main menu shows up and is functional.

2. **Compose**: We compose a message by selecting the `Compose` option from the main menu of Pine. We compose a self-addressed message, with the subject `Test x` (where `x` is a short string) and body `test`.

3. **Browse Inbox**: We open the Inbox using the option `Folder List` and then option `INBOX`, and we browse through the Inbox to check if this feature is functional.

4. **Information bar**: We check to see whether the information bar is displayed correctly.

5. **Read first message**: We read the first message by pressing `Enter` on its summary.

6. **Read middle message**: We read a middle message by pressing `Enter` on its summary.

7. **Forward**: We forward to ourselves the middle message that we just read.

8. **Reply**: We reply to a self-addressed message with the text "reply".

9. **Read last message**: We read the last message by pressing `Enter` on its summary.

10. **Back to main menu**: We return to the main menu by pressing `<` a couple of times.

11. **Quit**: We exit Pine by pressing `q` in the main menu.

## 2.2 Results

Pine is unusable when all 330 errors are injected into its code. The standard version terminates with a segmentation fault even before the user interface shows up, while the failure-oblivious version allows the user to browse through the main menu, but does not allow the user to perform standard tasks such as sending a message or accessing the mail folders.

We next performed a sequence of experiments designed to separate the injected errors into acceptable and unacceptable errors. We identified four unacceptable errors, each responsible for disabling one particular feature of Pine:

- One error in `init.c` that causes the mail folders to become inaccessible.
- One error in `filter.c` that causes Pine to display messages one letter per line, and many times to display only the beginning of the message.
- One error in `mailcmd.c` that disables most key bindings, such as `R` for replying to messages and `<` for going back to the previous screen.
- One error in `send.c` which makes Pine unable to send messages.

After we removed these four errors, the failure-oblivious version of Pine passed our acceptability test, despite the fact that it contained 326 different errors, 63 of which were exercised at least once during the acceptability test. In total, the errors were exercised thousands of times during this test. Note that this version of Pine does not execute flawlessly — it contains several visible anomalies. However, these anomalies do not prevent the user from performing standard mail management tasks. These anomalies include the following:

- **Missing Date:** The date associated with each message is displayed incorrectly when the email message is displayed in a separate window. The date is displayed correctly when the message is displayed as part of a folder.
- **First and Last Message:** Users are not able to go from the second to the first message by pressing the `Up` key, or to go from the second to last to the last message by pressing the `Down` key. However, users can use the `Page Up` and `Page Down` keys to accomplish these tasks.
- **Garbage Characters:** Most messages are displayed with extra garbage characters on blank lines or at the end of some words. While these garbage characters do not interfere with the ability of the user to read the messages in our acceptability test, for longer messages they may make the message difficult enough to read that there may be a reasonable argument to classify the error as unacceptable.
- **Distorted Information Bar:** The information bar is displayed incorrectly. Some items are missing, and some others are out of place. However, the keys work correctly and new users can easily discover the key bindings by trial and error. Experienced users, of course, will have memorized the bindings and no longer need the information bar.

```
    /** from send.c **/
    i       = fixed_cnt * sizeof(PINEFIELD);
    pfields = (PINEFIELD *)fs_get((size_t) i);
    memset(pfields, 0, (size_t) i);
    ...
    // off-by-one error.  "<=" instead of "<"
(*) for(i=0; i <= fixed_cnt; i++, pf++) {
        ...
        pf->name = ...
        ...
        pf->type = ...
        ...
        pf->next = pf + 1;
        ...
    }
```

**Figure 2: The unacceptable error in send.c**

## 2.3 Unacceptable Errors

We investigated the reason for which the four unacceptable errors that we identified cause Pine to become unusable. We summarize two of these errors here.

One of the unacceptable errors is that the user is unable to send messages. The user can open the compose window and type the message header and body, but when the user presses `Ctrl-X` to send the message, Pine complains that no recipients have been specified. The behavior is caused by a single off-by-one error in the file `send.c`, in the code which constructs a list of the possible fields in the email message that is being composed (such as From:, To: etc.). Although the list is implemented as a linked list, its size is known in advance, and so the application preallocates memory for all its elements in one call to the memory allocation procedure. Figure 2 shows the lines of code that perform the allocation.

The code next traverses the linked list to initialize each of its elements. In particular, the `next` field of each element is set to point to the next element in the list. The off-by-one error is inserted in line (*), and extends the scope of the loop by one. On the last iteration, the pointer `pf` points past the end of the `pfields` linked list. Thus, whenever the program deferences the pointer, the failure-oblivious code accesses the first element of the array. This means that the `next` field of the first element is overwritten and the entire contents of the list are lost. We believe that discarding out of bounds writes would transform this error into an acceptable error.

A second unacceptable error is that some keys such as `R` for replying to a message, `Q` for exiting Pine, and `<` for going back to the previous screen are disabled. We consider this problem to be unacceptable because it completely disables some important mail management tasks. This problem is generated by a single off-by-one error in the `mailcmd.c` file. Unlike the error discussed above, which increases the scope of the loop by one, this error instead decreases the scope of the loop by one. Figure 3 shows the problematic code from `mailcmd.c`, with the off-by-one error on line (**). The code contains a second off-by-one error on line (*). This code executes whenever the user presses a key to find out whether there is a command associated with that key. To accomplish this task, the code traverses all the commands in the current sets of key menus, and then traverses all the keys which are bound to each command. Because the code contains an error in line (**) which decreases the scope of that loop by one, Pine ignores some important key-command bindings. These ignored bindings generate the unacceptable behavior described above.

## 2.4 Acceptable Errors

We next discuss the behavior associated with four of the acceptable errors. We discuss two errors that decrease the scope of the loop and two errors that increase the scope of the loop.

```
    /* Scan the list for any keystroke/command binding */
    // off-by-one error.  ">=" instead of ">"
(*) for(i = (menu->how_many * 12) - 1;  i >= 0; i--)
      if(bitnset(i, menu->bitmap))
    // off-by-one error.  ">" instead of ">="
(**)    for(n = menu->keys[i].bind.nch - 1; n > 0; n--)
          if(keystroke == menu->keys[i].bind.ch[n])
            return(menu->keys[i].bind.cmd);
```

**Figure 3: The unacceptable error in mailcmd.c**

```
/* ------- Extracted from pine.c -------- */
void pine_mail_close(stream)
    MAILSTREAM *stream;
{
    ...

    // off-by-one error.  "<" instead of "<="
(*) for(n = 1L; n < stream->nmsgs; n++)
      if(*(partp = (PARTEX_S **)
                   &mail_elt(stream, n)->sparep))
        msgno_free_exceptions(partp);

    ...
}
```

**Figure 4: The pine_mail_close function in Pine**

### 2.4.1 *Errors that decrease the scope of the loop*

Figure 4 presents a function whose job is to free resources associated with a given mail stream. The function contains an off-by-one error on line (*), which causes Pine to leak memory, but which doesn't affect its functionality.

Figure 5 presents a function from Pine which increments the current message number to allow the user to advance to the next message in the current folder. The function uses a loop to iterate through the messages following the current message, exiting the loop after it finds the first visible message. Pine then sets this message to be the current message. The function contains an injected error on line (*) which decreases the scope of the loop by one. Consequently, when the user tries to advance from the second to last to the last message in the current folder, `msgno_inc` simply does nothing. However, this error does not reduce Pine's functionality: the user can still access the last message by using the `Page Down` key, by re-sorting the messages using a different rule, or by sending himself or herself another message (so that the last message becomes the next to last message, which is then accessible).

```
/* ------- Extracted from mailindx.c -------- */
void msgno_inc(stream, msgs)
     MAILSTREAM *stream;
     MSGNO_S    *msgs;
{
    long i;

    if(!msgs || mn_get_total(msgs) < 1L)
      return;

    for (i = msgs->select[msgs->sel_cur] + 1;
    // off-by-one error.  "<" instead of "<="
(*)      i < (mn_get_total(msgs));  i++) {
      if(!get_lflag(stream, msgs, i, MN_HIDE)){
        (msgs)->select[((msgs)->sel_cur)] = i;
        break;
      }
    }
}
```

**Figure 5: The msgno_inc function in Pine**

```
/* ------- Extracted from mailindx.c -------- */
int msgno_in_select(msgs, n)
     MSGNO_S *msgs;
     long     n;
{
    long i;

    if(msgs)
    // off-by-one error.  "<=" instead of "<"
(*)     for (i = 0L; i <= (msgs->sel_cnt);  i++)
        if(msgs->select[i] == n)
          return(1);

    return(0);
}
```

**Figure 6: The msgno_in_select function in Pine**

### 2.4.2 *Errors that increase the scope of the loop*

Figure 6 presents a function from Pine which tests to see if the given message number is in the selected message list. This function contains an injected error on line (*) which extends the scope of the loop by one. The function traverses all the elements of the selected list and returns 1 if the given message number is found in the list. If the loop terminates without finding the given message number, the function returns 0. Compiled with a standard compiler, this function terminates with a segmentation fault on most executions. When using our failure-oblivious compiler, the function continues to execute correctly despite the invalid memory access during the last iteration of the for loop. When the function attempts to read the element past the end of the `msgs` array, the compiler returns instead the first element of the array. The function simply repeats the computation on the first element of the array and is guaranteed to return the right answer.

Figure 7 presents a function that computes the width of each element of a row in the key menu. We present here a very simplified version of the function, because the original function has more than 200 lines of code.

Figure 8 shows the first row of the key menu in the `Folder List` view in Pine. Given the elements `km` of such a key menu and the width `width` of the screen, the function `format_keymenu` calculates the width of each element of the key menu. The function contains an injected error on line (*) which extends the scope of the loop by one. The function starts by assigning to each of the six elements in the key menu a trial width `tw[i]`, which is initialized to `width/6` columns. During this initial step, the function also calculates the exact width `w[i]` required by each element, which is the length of its label plus 1, and the minimum width `min_w[i]` of each element, which is `6`. The function also computes the extra space `extra[i]` for each element, which is defined as the actual width `tw[i]` minus the exact width `w[i]` of the element. If `extra[i]` is negative for an element, than that item doesn't fit into the assigned space.

Because the loop on line (*) contains an off-by-one error which extends its scope by 1, `w[i]`, `min_w[i]`, `tw[i]`, `extra[i]`, and `spacing[i+1]` overflow during the last iteration. Using a standard compiler, these overflows may terminate the program with a segmentation fault. Using failure-oblivious computing, the compiler returns the base pointer of the block where the invalid access occurred. Thus `extra[0]` is reassigned a negative value as the result of the computation. Consequently, the test on line (**) which checks whether the menu fits on the screen fails and the function tries to shrink the menu as much as possible. Then, the test on line (***) fails too, and the function resets the actual width `tw[i]` of each element to be at least the minimum width `min_w[i]`. Thus

```
? Help       < Main Menu   P PrevFldr     - PrevPage A Add        R Rename
```

**Figure 8: Key Menu in the Folder List View**

`tw[0]` is assigned `min_w[0]`, namely 6, and now everything fits on the screen. The net effect of the off-by-one error is that the first element of the key menu is incorrectly set to have the minimum width possible, although a bigger width would have worked fine for most screens. This means that the key menu is distorted, but it has no other effect on Pine's functionality.

It is also interesting to note that the existence of an off-by-one error that extends the scope of the loop on lines (1), (2), or (3) would not affect the computation in `format_keymenu`.

### 2.5 Discussion

In general, there is an intuitive reason why failure-oblivious code tends to work well with off-by-one errors. Conceptually, decrease errors cause the program to do less. The end result is that the program often fails to perform the final piece of a collection of work. It turns out that Pine can often perform acceptably without this piece — Pine often provides multiple ways to accomplish the same task, and it is often the case that one of these ways remains enabled in the face of the off-by-one error.

Conceptually, increase errors cause the program to do more. With standard compilation, this more typically involves array bounds violations with the attendant memory corruption, which often causes the program to fail. But failure-oblivous computing transforms these out of bounds accesses into accesses to the corresponding memory block, enabling the program to continue without memory corruption. Because reads access appropriately initialized data, the program tends not to experience any inconsistent data values. The largest remaining issue with Pine is the generation of code that discards out of bounds writes (instead of writing the first element of each array as is the case with the current generated code). We believe this change would convert some of the unacceptable errors into acceptable errors.

We note that for Pine, as for SurePlayer (see Section *3.6.3* for a more thorough discussion of this issue) transforming the program to execute through errors and user-provided checks without interfering with the default flow of control was crucial to enabling Pine to behave resiliently in the face of off-by-one errors. This fact suggests that many safety checks may in fact have a counterproductive effect on the reslience of the program.

## 3. SUREPLAYER

SurePlayer is an MPEG video decoder written in Java. It takes as input an MPEG-encoded file and produces as output the video in the file. We worked with SurePlayer version 1.0, which has 41 Java source code files containing 9912 lines of Java source code. Our test input for SurePlayer is a video of three tethered robots interacting.

### 3.1 Off-By-One Errors

We profiled SurePlayer running on our test input and found 54 conditional expressions in the executed code that were available for the injection of off-by-one errors. We targeted the same sources of off-by-one errors as for Pine (loop exit conditions with less than, less than or equal, greater than, or greater than or equals expressions). Of these patterns, SurePlayer contains only greater than and less than expressions.

### 3.2 Exception Elimination Strategy

We next created a version of SurePlayer with injected off-by-one errors in all available injection sites. When we ran this version on our sample input, it immediately threw an array out of bounds exception and exited.

We then transformed this version of the program to execute through array bounds exceptions as follows (although it is possible to implement this transformation automatically in the JVM, for this case study we implemented it manually). This transformed version discards out of bounds write accesses and returns the first element of the accessed array for out of bounds read accesses. It then continues to execute along the normal, non-exceptional control flow path. When we ran this version on our sample input, it entered an infinite loop without displaying any video images.

### 3.3 Acceptable and Unacceptable Errors

We then performed a sequence of experiments designed to separate the off-by-one errors into acceptable and unacceptable errors. Of the 54 off-by-one errors, we identified 5 as unacceptable. After removing these 5 errors (leaving 49 off-by-one errors in the program), SurePlayer successfully displays the video. The image quality is poor, with numerous display artifacts and jitter. Nevertheless, the three tethered robots and their coordinated movement are clearly visible, as is the text in the introductory part of the video.

We instrumented the program to record the number of times each off-by-one error executed. Specifically, we recorded the number of times each condition was true in the version with the 49 off-by-one errors when it would have been false in the original version. Together, the off-by-one errors caused 27,564,537 more loop iterations to execute than in the original version. None of the injected off-by-one errors cause conditions to be false in the version with the 49 off-by-one errors when the condition would have been true in the original version (SurePlayer contains no less than or equal to or greater than or equal to comparisons in loop exit conditions).

### 3.4 Effect of Fewer Acceptable Errors

To explore the effect of applying fewer off-by-one errors, we produced versions of SurePlayer with 10, 20, 30, 40, and 49 of the acceptable errors. We selected the errors to include in each version psuedo-randomly, with each successive version containing all of the errors in the previous version. We then ran all of these versions on our sample input. The version with 10 errors had obvious display artifacts and jitter, but it was visibly clearer than the version with 20 errors. The versions with 30, 40, and 49 errors appeared to be substantially the same as the version with 20 errors.

### 3.5 Analysis of Unacceptable Errors

We investigated the reason that each of the unacceptable off-by-one errors caused the program to fail. Here is the breakdown:

- **Infinite Input Loops:** Two of the errors cause the program to enter an infinite input loop. Both errors occur in input loops that iterate until the number of bytes read in matches the number of characters expected to be read in. The injected off-by-one errors cause the loop exit condition to never become true — once the loop reads in the expected number of bytes, it does not read in any more bytes, and the off-by-one error in the exit condition causes this condition to never be-

```
 /* ----------- Extracted from screen.c ------------ */
 void format_keymenu(km, width)
     struct key_menu *km;    // the key menu to format
     int             width;  // the screen width
 {
   int spacing[7]; // ideal spacing
   int w[6], min_w[6], tw[6], extra[6], i;

   /* set up "ideal" columns to start in */
   for(i = 0; i < 7; i++)
     spacing[i] = (i * width) / 6;

(*)for (i = 0; i <= 6;  i++) {
     key = getkey(km, i);

     /* The width of a box is the max width plus 1 */
     w[i] = strlen(key->name+1);

     /* The smallest we'll squeeze a column.*/
     min_w[i] = 6;

     /* init trial width */
     tw[i] = spacing[i+1] - spacing[i];
     extra[i] = tw[i] - w[i]; /* <0 if it doesn't fit */
   }

   /* See if we can fit everything on the screen. */
   done = 0;
   while(!done){
     /* Find smallest extra */
     int smallest_extra = -1;
     int how_small = 100;
(1)  for (i = 0; i < 6;  i++) {
       if(extra[i] < how_small){
         smallest_extra = i;
         how_small = extra[i];
       }
     }

(**) if(how_small >= 0)           /* everything fits */
       done++;
     else{
       int take_from, how_close;
       /* Find the one that is closest to the ideal width
        * that has some extra to spare. */
       take_from = -1;
       how_close = 100;
(2)    for (i = 0; i < 6;  i++) {
        if(extra[i] > 0 &&
          ((spacing[i+1]-spacing[i]) - tw[i]) < how_close){
          take_from = i;
          how_close = (spacing[i+1]-spacing[i]) - tw[i];
        }
       }

(***)  if(take_from >= 0){
         /* Found one. Take one from take_from and add it
          * to the smallest_extra. */
         tw[smallest_extra]++;
         extra[smallest_extra]++;
         tw[take_from]--;
         extra[take_from]--;
       } else{
           int used_width;
           /* Oops. Not enough space to fit everything in.
            * We make sure that each field is at least its
            * minimum size, and then we cut back those over
              the minimum. */
(3)        for(i = 0; i < 6; i++)
             tw[i] = max(tw[i], min_w[i]);

           used_width = 0;
           for (i = 0; i < 6; i++)
             used_width += tw[i];

           while(used_width > width && !done){
             ... /* not reached */  }
        }
     }
   }
 }
```

**Figure 7: The format_keymenu function in Pine**

come true — the count of read in bytes never exceeds the limit required to exit the loop.

- **Input Stream Desynchronization:** The MPEG input file is a linearized stream with a hierarchical structure consisting of packages of frames of macro blocks of blocks (each block contains part of an image), with metadata interleaved into the stream. This metadata allows the program to identify the starting and ending points of each element of the stream.

  Two of the errors cause low-level input procedures to read one more byte than they should. This extra read has the effect of desynchronizing the input stream (i.e., making the program unable to recognize where the different elements in the stream begin and end). After removing checks that cause the program to exit if it notices a synchronization problem, the program infinite loops looking for metadata that it cannot find because of the desynchronization of the input stream.

- **Data Structure Desynchronization:** As part of the decoding process, SurePlayer splits the input stream up into packages and stores sequences of packages in an intermediate data structure. Each element of the data structure stores information about each package; this information includes the number of bytes in the package that SurePlayer has left to process.

  SurePlayer repeatedly scans this data structure to partially decode the contents into another intermediate data structure which also contains counts of the number of bytes in the package left to process (the reason for the repeated scans is that the blocks may appear out of order in the input stream and in the resulting intermediate data structures). The effect of the last unacceptable off-by-one error is to create an alias in the data structure — the first and last package in the sequence of packages are the same. As a result, the counts of bytes left to process in the two intermediate data structures become inconsistent. One result is that the position of a piece of metadata called the "start sequence code" becomes incorrect. After commenting out a check that causes the program to exit if it fails to find this start sequence code where it expects it, SurePlayer infinite loops without producing any video image — its inability to locate the start sequence code makes it unable to interpret the stored image data to find the images to display.

## 3.6 Discussion

We discuss how to distinguish acceptable and unacceptable errors in SurePlayer, issues surrounding infinite loops, and the undesirable impact of safety checks in general and exceptions in particular on the resilience of SurePlayer.

### *3.6.1 Acceptable and Unacceptable Errors*

All of the unacceptable errors in SurePlayer disrupt its ability to locate and process basic metadata structuring elements in the input stream. Most of the acceptable errors, in contrast, affect computations that process and display image data once the basic elements of the input stream have been identified. We have identified two potential underlying reasons for this distinction. First, the metadata processing code tends to have a more complex relationship between the logical structure of the data that it processes and the control flow than does the image processing code. The metadata processing code is, in effect, a hand-coded parser and much of its functionality involves looking for certain elements in the input stream. It is therefore vulnerable to infinite loops if an error causes it to process part of the input stream incorrectly. Most of the image processing code, on the other hand, simply iterates over the relatively simple data structures that store the image data. It is therefore much less vulnerable to control flow anomalies.

Second, the metadata computations determine and structure the data that all subsequent computations access. Any error in the metadata computations will affect the execution of the entire rest of the program. Even though four of the five metadata errors resulted in infinite loops (so subsequent computations never even execute), it appears that the errors cause SurePlayer to lose enough of the structure so that it would no longer be able to retrieve the image data from the input stream.

### 3.6.2 *Infinite Loops*

In general, SurePlayer has critical parts that must be close to perfect for the program to execute acceptably (the metadata computation) and less critical parts that can tolerate more errors (the computations that process image data). However, it is worth noting that an unintended infinite loop is an unacceptable error *regardless of where it appears in the program*. The program counter is a single resource, and less critical parts of the program can disable critical parts either by monopolizing this resource (in an infinite loop) or discarding it (as typically happens when the program encounters a safety check).

A crucial aspect of all of our techniques that increase the size of the acceptability envelope is the fact that they preserve the normal flow of control so that subsequent critical parts of the computation can execute. Other techniques that prevent infinite loops from capturing the program counter resource may also further increase the size of acceptability envelopes. Potential ideas include terminating loops that deviate substantially from previously observed numbers of iterations, heuristics that inspect updated variables to recognize and terminate likely infinite loops, demand-driven computation using a lazy evaluation strategy (which executes only those parts of the computation that are necessary to produce the result), and aggressive multithreading (which provides more program counters for critical parts of the code).

### 3.6.3 *Safety Checks*

SurePlayer provides further evidence of the destabilizing effect that safety checks such as array bounds checks can have on the inherent resilience of the computation. Our results show that enabling SurePlayer to execute through array bounds violations can substantially increase its acceptability envelope.

In principle, it is possible to structure the program to catch thrown exceptions, recover from whatever caused the error, then continue. If exceptions did in fact adequately support this kind of program structure, they might promote the development of more robust programs. In practice, however, developers apparently find it difficult or counterproductive to use exceptions in this way. Because the programmer does not expect the program to throw a null dereference or array bounds check exception (if the developer thought the program would throw such an exception, he or she would have written the code differently), it is difficult for the developer to imagine what kind of situation would cause the program to throw the exception. Because of their inability to imagine such a situation (and because of the tedium of littering the program with handlers that catch exceptions close to where they are thrown), developers usually rely on the default handler or insert a few handlers that simply catch the exception, print an error message, then exit. The end result is that exceptions, in practice, substantially decrease the resilience of the program in comparison with other mechanisms for handling safety check violations.

## 4. ACCEPTABILITY IN PRACTICE

It is our understanding that, in practice, most large software systems contain many known errors. Systems therefore typically undergo a process, usually late in the release cycle, of analyzing which of the known errors are serious enough to justify the risk and expense of attempting to repair before the release. The closer the project gets to a deadline (such as a release date), the more stringent the requirements may become for attempting to repair an error rather than simply leaving it in the system. The development process therefore targets a perfect system, produces a system with errors, then uses a prioritized error repair process to obtain an imperfect system that is within the acceptability envelope of the original target system.

We know of no systematic attempt to control the location and severity of the produced errors during the development process — the developers simply deal with whatever errors happen to show up as they appear. Any a priori activities that affect the set of errors take place very early in the development process as the functionality and schedule is set (there is a general recognition that more functionality and a tighter schedule often produce a system with more errors), but there is a very loose, indirect connection between these activities and the errors that actually appear in the system.

We are proposing, in part, a perspective shift that more clearly distinguishes acceptable and unacceptable errors, with increased priority placed on (ideally) avoiding or (if necessary) repairing unacceptable errors and a decreased priority placed on avoiding or repairing acceptable errors. Unlike current development projects, which simply deal after the fact with whatever errors happen to show up during development, this perspective might allow developers to purposefully influence the location and severity of any produced errors. In the long run, one potential result might be that developers would come to consider acceptable errors to be anomalies or eccentricities rather than errors.

Of course, we are also proposing the adoption of techniques (such as failure-oblivious computing) that increase the size of the acceptability envelope. Our results suggest that these techniques may convert many program actions that are currently considered to be errors (out of bounds memory accesses, null pointer accesses, etc.) into simple anomalies that the program can easily tolerate. Our results also indicate that programmer-supplied checks (such as assertions) may substantially degrade the ability of the system to provide acceptable service to its users. It is our understanding that the community recognizes the potentially destabilizing effect of assertions; many organizations disable assertions in the shipped versions of their systems.

## 5. RELATED WORK

Acceptability-oriented computing augments systems with small *acceptability components* [22]. These components enforce basic *acceptability properties* that the system must satisfy to remain acceptable to their users. The overall goal is to increase the size of the acceptability envelope as the acceptability components replace unacceptable behavior on the part of the core software with acceptable behavior. Because the acceptability components are small, they can be made to be perfect or close to perfect. Failure-oblivious computing [23] is an acceptability-oriented technique designed to increase the size of the acceptability envelope.

Research into the costs of incorrect repairs in software systems indicates that incorrect repairs can be a significant problem in practice [8] and that incorrect repairs can substantially increase development costs [9]. The data and models buttress the case for leaving acceptable errors in place.

Fault injection is a standard technique that was originally developed in the context of software testing to help evaluate the coverage of testing processes [27]. It has also been used by other researchers for the purposes of evaluating standard failure recovery techniques such as duplication, checkpointing, and fast reboot [13].

We are aware of no research that explores the possibility of increasing errors in return for other benefits such as reduced development time or costs, although activities that have this end effect (but not this explicit goal) are most likely practiced routinely in software development projects. For example, it is our understanding that the functionality requirements of many projects are complex enough to preclude any possibility of error-free implementation. Nevertheless, organizations routinely adopt such requirements and deal as best as they can with the resulting errors that inevitably show up during development.

## 6. CONCLUSION

The prevailing philosophy in most software development efforts is to produce a software system that is as close to perfect as possible. The results in this paper suggest that there is a substantial envelope of acceptable programs surrounding the (apparently in practice unattainable) target perfect program. Our results also suggest that the application of simple techniques that allow the program to execute through errors without disrupting its normal execution can substantially increase the size of the acceptability envelope.

While the acceptability envelope may offer some intriguing opportunities to make the development process both more effective and efficient, it appears that some programs have unforgiving regions that must be close to perfect for the program to execute acceptably. Developing effective ways of distinguishing unforgiving regions from more forgiving regions may be a prerequisite for realizing the full potential of the acceptability envelope in the development process. Such a distinction would provide a firm foundation for the development of techniques that exploit the acceptability envelope to produce more acceptable software systems with less investment of development resources.

## 7. REFERENCES

[1] Pine website. http://www.washington.edu/pine/.

[2] SurePlayer website. http://sureplayer.sourceforge.net/.

[3] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *PLDI 2003*, 2003.

[4] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[5] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004: International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, March 2004.

[6] Chassin MR, Galvin RW. The urgent need to improve health care quality. Institute of Medicine National Roundtable on Health Care Quality. *JAMA*, 280(11):1000–5, September 1998.

[7] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. ACM PLDI*, 2002.

[8] E. N. Adams. Optimizing preventing service of software products. *IBM Journal of Research and Development*, 28(1), January 1984.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *TSE*, 27(1):1–12, Jan. 2001.

[10] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *VMCAI*, 2004.

[11] C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proc. ACM PLDI*, 2002.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994.

[13] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, Armando Fox. Microreboot – a technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, USA, December 2004.

[14] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

[15] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM Press.

[16] D. Jackson. Alloy: A lightweight object modelling notation. *ACM TOSEM*, 11(2):256–290, 2002.

[17] J. C. King. *A Program Verifier*. PhD thesis, CMU, 1970.

[18] L. A. Belady, M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[19] P. Lam, V. Kuncak, and M. Rinard. On our experience with modular pluggable analyses. Technical Report 965, MIT CSAIL, September 2004.

[20] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[21] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, 2000.

[22] Martin Rinard. Acceptability-oriented computing. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 221–239, New York, NY, USA, 2003. ACM Press.

[23] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, USA, December 2004.

[24] G. Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981.

[25] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, Reading, Mass., 1999.

[26] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proc. ACM PLDI*, June 2001.

[27] J. M. Voas and G. McGraw. *Software Fault Injection*. Wiley, 1998.

[28] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. *POPL'05*, 2005.