Acceptability-Oriented Computing

Martin Rinard
MIT Laboratory for Computer Science
Cambridge, MA 021139

ABSTRACT

We discuss a new approach to the construction of software systems. Instead of attempting to build a system that is as free of errors as possible, the designer instead identifies key properties that the execution must satisfy to be acceptable to its users. Together, these properties define the *acceptability envelope* of the system: the region that it must stay within to remain acceptable. The developer then augments the system with a layered set of components, each of which enforces one of the acceptability properties. The potential advantages of this approach include more flexible, resilient systems that recover from errors and behave acceptably across a wide range of operating environments, an appropriately prioritized investment of engineering resources, and the ability to productively incorporate unreliable components into the final software system.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications;
D.2.3 [**Software Engineering**]: Coding Tools and Techniques;
D.2.3 [**Software Engineering**]: Software/Program Verification;
D.2.3 [**Software Engineering**]: Testing and Debugging;
D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Reliability, Security

Keywords

Acceptability Properties, Repair, Monitoring, Rectification

*This research was supported in part by DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, and NSF Grant CCR00-63513, NSF Grant CCR02-09075, an IBM Eclipse Award, and the MIT-Singapore Alliance.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

1. INTRODUCTION

Software developers face increasing demands to improve both the functionality and the reliability of the software that they produce. This combination is challenging because of the inherent tension between these two properties: increasing the functionality increases the complexity, which in turn increases the difficulty of ensuring that the software executes reliably without errors.

We believe that current software engineering practices may be approaching the limit of the combination of functionality and reliability that they can deliver. This is of some concern because of the compelling need for improvements in both of these two properties:

- **Functionality:** Functionality pressure has traditionally come from the dynamics of the desktop software market. In this context, increased functionality has been seen as a way to obtain a marketing advantage over competitors, to motivate customers to purchase upgrades, and to satisfy customer demands for more useful or usable computer software.

  In the future, several trends will increase the pressure for additional software functionality (and therefore additional software complexity). First, increasing interconnectivity places demands on the software to interoperate with a wider range of systems and data sources. Each new system and data source typically requires the development of new functionality to support the interoperation.

  Second, embedded systems are the next large software growth area. As the scope and sophistication of the phenomena that these systems monitor and control increases, so must the functionality and complexity of the software.

- **Reliability:** Because many embedded systems control physical phenomena, failures can have immediate and disastrous consequences that threaten human safety. Reliability has therefore always been a primary concern for embedded systems; its importance will only increase as embedded systems continue to expand the scope of the phenomena that they monitor and control.

  Reliability has also been a central concern for traditional corporate information management systems: down time translates directly into impaired operations, lost profits, and frustrated customers may look to competitors to satisfy their needs. The increased dependence on highly interconnected information systems

only magnifies the importance of reliability. As interconnectivity increases, more of the business becomes unable to function in the absence of a working system, and the system as a whole becomes increasingly vulnerable to cascading failures triggered by the failure of a single component. Both of these phenomena increase the consequences of failures, in some cases dramatically.

Finally, after many years in which features were the primary concern, consumers of desktop productivity software are tiring of dealing with software that fails. Reliability is therefore becoming an important issue for this kind of software, even though the consequences of failure are usually small in comparison with embedded systems or corporate information management systems.

The standard approach for achieving reliability has been simplicity, or to state it more pejoratively, lack of ambition: keep the program simple enough so that it can be made to work with standard techniques. As legitimate functionality demands increase, this approach will become increasingly infeasible. The more ambitious alternative, namely the deployment of increasingly less reliable systems, seems unattractive and potentially disastrous. And the fact that these issues play out against a backdrop of ruthless competition, ever-increasing time and market pressures, accelerated development schedules, and continuously changing requirements and computing environments only exacerbates the need for better solutions.

If we step back and examine the situation, it becomes clear that there is almost never just a single acceptable execution that the program must deliver. Instead, there is usually a range of acceptable executions, with some executions more acceptable than others. This fact suggests the concept of an *acceptability envelope:* a region of the execution that the program must stay within to remain acceptable. This concept, in turn, suggests the following development approach: instead of trying to develop a perfect program, instead develop a program that may contain errors, but that always stays within its acceptability envelope in spite of those errors. Several activities characterize this approach:

- **Acceptability Property Identification:** The designer identifies properties that the state and behavior must preserve for the program's execution to be acceptable. The result is a set of (potentially) partially redundant and prioritized *acceptability properties.* Unlike traditional specifications, which completely characterize the desired behavior, acceptability properties are *partial:* they identify minimum acceptability requirements for specific aspects of the execution.
- **Monitoring:** The developer produces components that monitor relevant aspects of the program's execution to detect impending acceptability violations.
- **Enforcement:** Once the component detects an impending acceptability violation, it takes actions designed to bring the program's execution back within its acceptability envelope.
- **Logging:** The component logs important events such as the detection of impending acceptability violations and the actions that the system takes to avert such violations.

1.1 Perspective Change

Acceptability-oriented computing promotes a perspective change on the part of the developers and users of software. In particular, the following notions are implicit but central to this approach:

- **The Goal of Perfection is Counterproductive:** The aspiration to eliminate as many programming errors as possible creates a development process that diffuses the focus of the project, wastes engineering resources, and produces brittle software that is helpless in the presence of the inevitable errors or faults. A more productive aspiration is to develop systems that contain errors and sometimes behave imperfectly, but remain within their acceptable operating envelope.
- **Flawed Software Has Enormous Value:** The most productive path to better software systems will involve the combination of partially faulty software with techniques that monitor the execution and, when necessary, respond to faults or errors by taking action to appropriately adjust the state or behavior.

If successful, this approach will enable the construction of computer systems that can sustain (potentially self-inflicted) damage, process unexpected or illegal inputs, and take incorrect actions, yet nevertheless continue to execute productively.

1.2 Architecture

Acceptability-oriented systems have an architecture that differs substantially from that of traditional systems. Specifically, acceptability-oriented systems contain multiple partial, redundant, and layered components that enforce acceptable system behavior and structure properties. Outer layers take priority over inner layers, with the layers becoming progressively simpler, more partial, and more likely to accurately capture the intended property as they move towards the periphery of the layered structure.

The core innermost layer will typically consist of the source code of the system, usually written in a standard programming language. Conceptually, the core is intended to completely specify both the behavior of the system and the structure required to completely implement this behavior. The possibility of errors within the core or unanticipated actions on the part of the environment in which the computation exists is the only reason that the core does not, by itself, comprise a completely acceptable system.

The outer layers identify increasingly minimal requirements for acceptable system behavior and structure. Conceptually, the information in these layers may be redundant with the information in the innermost layer. The partial nature of these outer layers means that many of the layers address orthogonal aspects of the system and are therefore largely independent. Others provide information that is partially or completely redundant with other layers. The outermost layers focus on only those most basic properties required for minimal system acceptability while layers closer to the core focus on more elaborate properties that, when satisfied, enhance the value of the program.

The outer layers of the program monitor the execution of the program to identify impending violations of the desired acceptability properties. Potential responses to such violations include repair actions that are designed to de-

liver an acceptable state for continued execution, exit actions that are designed to safely shut down the system and await external intervention, and logging actions that record the impending acceptability violations and the resulting acceptability enforcement actions.

The monitoring, repair, and safe exit algorithms will vary depending on the kinds of requirements they are designed to enforce. In many cases, it may be productive to express some of the layers using a variety of high-level specification languages. Tools will translate the resulting specifications into code that implements the monitoring, logging, repair, and safe exit actions. In other cases, it may be more productive to code the acceptability enforcement operations directly in standard programming languages.

1.3 Acceptability Enforcement Approaches

We identify two approaches that a system can use to respond to impending acceptability violations and enforce its acceptability envelope:

- **Resilient Computing Approaches:** A resilient computing approach responds to impending acceptability violations by taking actions that are designed to bring the system back into an acceptable state from which it can continue to execute indefinitely.

- **Safe Exit Approaches:** A safe exit approach responds to impending acceptability violations by executing a *safe exit strategy*: a sequence of actions designed to extricate the system safely from the problematic situation in which it finds itself. This sequence is designed to leave the system and any parts of the physical world with which it interacts in a *stable state*, or a state in which it is acceptable for the system to take no action at all until it is repaired by some external intervention.

We expect safe exit approaches to be used primarily when waiting for external intervention to repair the system is a reasonable response to failures or the perceived negative consequences of continuing to execute a potentially incompletely repaired system outweigh the benefits. Many existing systems already contain a very conservative, limited version of a safe-exit strategy: standard error checking mechanisms (such as assertions) can be viewed as enforcing very basic acceptability properties (the locally checkable conditions that appear in the assertions) with a simple safe exit strategy (halt the computation at the first sign of an acceptability violation). We expect that most such systems would benefit from a more comprehensive and purposeful application of acceptability-oriented computing, in particular, an increased awareness and understanding of explicit acceptability properties, a more organized approach to detecting impending acceptability violations, and a larger effort devoted to identifying more appropriate safe exit strategies.

We expect resilient computing approaches to be most appropriate in several situations: when the repair is expected to deliver a completely acceptable system, when the effect of the repaired damage will be flushed out of the system within an acceptable amount of time provided that it continues to execute, when external intervention is not practical, or when there is a strong incentive for continued execution over outright system failure, even if the continued execution may be somewhat compromised. In many embedded systems that control unstable physical phenomena, for example, compromised execution is far preferable to outright failure. Resilient computing may be especially useful for systems that are composed of several largely independent components that share a fate — repairing one component may enable the other components to continue with their normal execution, while allowing the component to fail may disable the remaining components. It may also be useful when the software is known to execute incorrectly in certain situations, but the developers are unwilling to risk the system disruption or damage that might result from changing the code, preferring instead to repair any damage after it occurs.

We note that it is possible to apply resilient computing and safe exit responses in the same system, with some unacceptabilities repaired via the resilient computing techniques and others triggering a safe exit strategy.

1.4 Acceptability Enforcement Mechanisms

It is possible to apply acceptability-oriented computing to all aspects of a system's structure and behavior. We have explored mechanisms that interpose acceptability filters on input and output streams, that dynamically traverse data structures to find and repair violations of key data structure consistency properties, and that enforce properties that relate the content of the data structures to the input and output values. Other likely candidates include enforcing process structure properties (such as the presence of processes that implement a crucial part of the system's functionality), system configuration properties (these constrain the values of various system configuration setting), and library usage properties (that involve the order in which clients may invoke library operations and the values that they may pass to the library).

We distinguish several distinct classes of mechanisms that can be used to apply acceptability-oriented computing:

- **Black-Box Mechanisms:** A black-box mechanism treats the core as a unit. It does not modify the core at all and affects it only through well-defined aspects of its interface such as its input and output streams.

- **Gray-Box Mechanisms:** A gray-box mechanism does not modify the code of the core, but may use more invasive techniques (such as procedure call interception and data structure modification) that affect conceptually internal aspects of the core.

- **White-Box Mechanisms:** A white-box mechanism augments the core directly with new acceptability enforcement code.

The basic trade-off is that less invasive techniques require less knowledge of the core system and are less likely to introduce new errors, but typically require more involved implementation mechanisms and may be limited in the amount of control they can exert over the core. More invasive techniques offer more control and typically use simpler implementation mechanisms, but require the developer to operate within a more complex implementation environment and offer greater opportunities for collateral damage in the form of inadvertently introduced errors.

1.5 Structure of the Paper

The remainder of the paper is structured as follows. In Section 2 we use a running example to present several different resilient computing techniques. We discuss issues associated with these techniques as they arise. In Section 3 we discuss several other techniques that we expect acceptability-oriented developers to use. In Section 4 we discuss several broader technical and social issues associated with the potential use of acceptability-oriented computing. In Section 5 we present several more aggressive extensions that may interfere more deeply with the core system in an attempt to generate more reliable execution. We discuss related work in Section 6, unrelated work in the area of reliability in Section 7, and conclude in Section 8.

2. EXAMPLE

We next present a simple example that illustrates how to apply the basic concepts of acceptability-oriented computing. The core component in our example implements a simple map from names to numbers. We assume that this component will be used to translate names to numerical identifiers, as in, for example, a name server that maps a bank of printer names to printer network addresses.

Like many Unix components, the core component in our example accepts its inputs as a sequence of commands on the standard input stream (`stdin`) and writes its outputs to the standard output stream (`stdout`). It accepts three commands: `put name num`, which creates a mapping from `name` to `num`; `get name`, which retrieves the `num` that `name` maps to; and `rem name`, which removes the mapping associated with `name`. In response to each command, it writes the appropriate `num` onto the standard output: for `put` commands it writes out the number from the new mapping, for `get` commands it writes out the retrieved number, and for `rem` commands it writes out the number from the removed mapping.

Figure 1 presents the code for the `main` procedure, which parses the input stream of commands, invokes the appropriate procedure (`put` for `put` commands, `get` for `get` commands, and `rem` for `rem` commands), then writes the return value of the procedure to the output stream. Figure 2 presents the code for these procedures. We have omitted the definitions of several constants (`LEN`, `N`, and `M`) and auxiliary procedures. The complete code for all of the examples in this paper is available at
`www.cag.lcs.mit.edu/~rinard/paper/oopsla03/code`.

The core maintains a hash table that stores the mappings. Each bin in the table contains a list of entries; each entry contains a mapping from one `name` to one `num`. The `bin` procedure uses a hash code for the `name` (computed by the `hash` function) to associate names to bins. The procedures `alloc` and `free` manage the pool of entries in the table, while the `find` procedure finds the entry in the table that holds the mapping for a given `name`.

```
int main(int argc, char *argv[]) {
  char cmd[LEN], name[LEN];
  int val;
  initialize();
  while (scanf("%s", cmd) != EOF) {
    val = 0;
    if (strcmp(cmd, "put") == 0) {
      if (scanf("%s %d", name, &val) == 2) {
        put(name, val);
      }
    } else if (strcmp(cmd, "get") == 0) {
      if (scanf("%s", name) == 1)  {
        val = get(name);
      }
    } else if (strcmp(cmd, "rem") == 0) {
      if (scanf("%s", name) == 1)  {
        val = rem(name);
      }
    }
    printf("%d\n", val);
    fflush(stdout);
  }
}
```

Figure 1: Implementation of `main` Procedure of Core

2.1 Acceptability Properties

We identify several acceptability properties that the core should satisfy:

- **Continued Execution:** Our first acceptability property is that the core continues to execute so that as many of its mappings as possible remain accessible to its client. The rationale for this property might be, for example, that the client (and potentially the rest of the system) will hang unless it receives a response for each request. Our acceptability enforcement techniques must therefore protect the core against problematic inputs and internal data structure inconsistencies that could cause it to fail.

- **Output Sanity:** Ideally, the core would always return the correct number for each `get` command. We may, however, be willing to relax this constraint to allow it to return 0 (indicating no mapping) even if a past `put` command established a mapping. We may also be willing to accept (or even require) the core to return any number within the minimum and maximum numbers to which mappings have been established. The rationale for these properties might be, for example, that the bank of printers has been allocated a contiguous range of addresses, and while it might be acceptable to deliver a job to any printer within the range, it might not be acceptable to route the job to some other arbitrary printer.

Note that both of these acceptability properties fall well short of requiring the core to execute correctly. They also depend, to some extent, on the needs of the client in the context of the larger system. In general, we expect the acceptability properties for the different components to vary depending on the context in which they are deployed. Finally, we note that the client can reasonably hope for more than just acceptability property satisfaction — instead of returning 0 for every `get` query, it is reasonable to expect the core to return the correct value much of the time.

2.2 Data Structure Repair

The `alloc` procedure in Figure 2 assumes that there is always a free entry to return to the caller; the `put` procedure assumes that it always gets a valid entry back from the `alloc` procedure. If these assumptions are violated (presumably because the client has attempted to put too many mappings into the table), there is a memory corruption error and the core may fail. The standard way to fix this problem is to

```
struct {
  int  _value;
  int  _next;
  char _name[LEN];
} entries[N];

#define next(e) entries[e]._next
#define name(e) entries[e]._name
#define value(e) entries[e]._value

#define NOENTRY 0x00ffffff
#define NOVALUE 0

int end(int e) {  return (e == NOENTRY); }

int table[M], freelist;

int alloc() {
  int e = freelist;
  freelist = value(e);
  return e;
}

void free(e) { value(e) = freelist; freelist = e; }

int hash(char name[]) {
  int i, h;
  for (i = 0, h = 0; name[i] != '\0'; i++) {
    h *= 997; h += name[i]; h = h % 4231;
  }
  return h;
}

int bin(char name[]) { return hash(name) % M; }
int find(char name[]) {
  int b = bin(name), e = table[b];
  while (!end(e) && strcmp(name, name(e)) != 0)
    e = next(e);
  return e;
}

int rem(char name[]) {
  int e = find(name);
  if (!end(e)) {
    int val = value(e), b = bin(name);
    table[b] = next(e);
    name(e)[0] = '\0'; free(e);
    return val;
  } else return NOVALUE;
}

int put(char name[], int val) {
  int e = alloc();
  value(e) = val; strcpy(name(e), name);
  int p = find(name);
  if (!end(p)) free(p);
  int b = bin(name);
  next(e) = table[b]; table[b] = e;
  return val;
}

int get(char name[]) {
  int e = find(name);
  if (end(e)) return NOVALUE;
  else return value(e);
}
```

Figure 2: Implementation of Map Core Procedures

modify the core software to recognize when it is out of entries and augment it with code to handle this case.

This approach, of course, requires the developer to operate directly within the core software. Two potential pitfalls include the need for the developer to understand the core software and collateral damage in the form of introduced errors. Moreover, the standard approach to running out of memory is to have the allocation routine return a value indicating that the allocation failed, leaving the application to deal with the problem. The application usually has no idea how to recover and exits (if it checks the return code at all). The alternative, distributing infrequently executed recovery code throughout the application, may be unattractive not only because of the increased complexity and damage to the structure of the code but also because of the difficulty of delivering recovery code that executes acceptably.

Modification also requires access to compilable source code, which may be problematic either because the core was delivered in executable form only, because the original source code has been lost, because legal considerations preclude modification, because recertification costs make modifications impractical, because another organization maintains the code and any local modifications will be wiped out in future releases, or because the core is coded in an archaic language for which a compiler is no longer available.

An alternate view is that the core is failing because its data structures do not satisfy the data structure consistency property that the `freelist` must point to a valid entry when the `alloc` procedure executes, and the way to eliminate the failure is to enhance the core with a component that detects, then repairs, any data structure consistency violations.

Further investigation reveals another consistency problem. The following input causes the core to infinite loop while processing the `get g` command:

```
put a 1
put c 2
put a 3
put e 4
get g
```

The cause of the infinite loop is a circularity in one of the lists of entries. This circularity violates the data structure consistency property that there is exactly one reference to each element of the `entries` array (these references are implemented as indices stored in the `freelist`, `table` array, and `next` and `value` fields of the entries). It may be worthwhile to ensure that this property holds as well.

Figure 3 presents a procedure, `repair`, that detects and repairs any data structure consistency violations in our example program from Figure 2. The `repair` procedure first invokes `repairValid` to replace all invalid references with `NOENTRY`. The procedure then invokes `repairTable`, which ensures that all entries in lists in the table have at most one incoming reference from either an element of the `table` array or the `next` field of some entry reachable from the `table` array. This property ensures that each element of `table` refers to a distinct `NOENTRY`-terminated list and that different lists contain disjoint sets of entries. Finally, `repair` invokes `repairFree` to collect all of the entries with reference count 0 (and that are therefore not in the table) into the free list. If the free list remains empty (because all entries are already in the table), the procedure `chooseFree` chooses an arbitrary entry and inserts it into the free list, ensuring that

```
int valid(int e) {  return (e >= 0) && (e < N); }

void repairValid() {
  int i;
  if (!valid(freelist)) freelist = NOENTRY;
  for (i = 0; i < M; i++)
    if (!valid(table[i])) table[i] = NOENTRY;
  for (i = 0; i < N; i++)
    if (!valid(next(i))) next(i) = NOENTRY;
}

static int refs[N];

void repairTable() {
  static int last = 0; int i, e, n, p;
  for (i = 0; i < N; i++) refs[i] = 0;
  for (i = 0; i < M; i++) {
    p = table[i];
    if (end(p)) continue;
    if (refs[p] == 1) {
      fprintf(stderr, "t[%d] null (%d)\n", i, p);
      table[i] = NOENTRY; continue;
    }
    refs[p] = 1; n = next(p);
    while (!end(n)) {
      if (refs[n] == 1) {
        fprintf(stderr, "n(%d) null (%d)\n", p, n);
        next(p) = NOENTRY; break;
      }
      refs[n] = 1; p = n; n = next(n);
    }
  }
}

void chooseFree() {
  static int last = 0; int i, n, p;
  fprintf(stderr, "freelist = %d\n", last);
  n = last; last = (last + 1) % N;
  name(n)[0] = '\0'; value(n) = NOENTRY; freelist = n;
  for (i = 0; i < M; i++) {
    p = table[i];
    if (end(p)) continue;
    if (p == freelist) {
      fprintf(stderr, "t[%d]=%d (%d)\n", i, next(p), p);
      table[i] = next(p); return;
    }
    n = next(p);
    while (!end(n)) {
      if (n == freelist) {
        fprintf(stderr, "n(%d)=%d (%d)\n", p, next(n), n);
        next(p) = next(n); return;
      }
      p = n; n = next(n);
    }
  }
}

void repairFree() {
  int i, f = NOENTRY;
  for (i = 0; i < N; i++)
    if (refs[i] == 0) {
      next(i) = value(i); value(i) = f; f = i;
    }
  if (end(f)) chooseFree();
  else freelist = f;
}

void repair() {
  repairValid(); repairTable(); repairFree();
}
```

Figure 3: Data Structure Repair Implementation

`freelist` refers to a valid entry. It then removes this entry from the table.

The repair algorithm maintains a reference count `ref[e]` for each entry `e` and uses this count to guide its repair actions. It also logs each repair action to `stderr`. The resulting log may make it easier to become aware of and investigate any errors in the core and to understand the behavior of the repaired system should it become desirable to do so.

The repair algorithm has several properties:

- **Heuristic Structure Preservation:** When possible, the algorithm attempts to preserve the structure it is given. In particular, it has no effect on a consistent data structure and attempts to preserve, when possible, the starting linking structure of inconsistent data structures. The repaired data structure is therefore heuristically close to the original inconsistent data structure.

- **Continued Execution:** When the free list is empty, the repair algorithm removes an arbitrary entry from the table and puts that entry into the free list. This action removes the entry's mapping; the overall effect is to eject existing mappings to make way for new mappings. In this case the repair algorithm converts failure into somewhat compromised but ongoing execution. Because the repair algorithm also eliminates any cycles in the table data structure, it may also eliminate infinite loops in the `find` procedure.

This example illustrates several issues one must consider when building components that detect impending acceptability violations and enforce acceptability properties:

- **Acceptability Property:** The developer must first determine the acceptability property that the component should enforce. In our example, the acceptability property captures aspects of the internal data structures that are directly related to the ability of the core to continue to execute. Note that the acceptability property in our example is partial in that it does not completely characterize data structure consistency — in particular, it does not attempt to enforce any relationship between the values in the entries and the structure of the lists in the `table` data structure. In a fully correct implementation, of course, the hash code of each active entry's name would determine the list in which it is a member.

- **Monitoring:** The monitoring component in our example simply accesses the data structures directly in the address space of the core to find acceptability violations. In general, we expect the specific monitoring mechanism to depend on the acceptability property and on the facilities of the underlying system. We anticipate the use of a variety of mechanisms that allow the monitor to access the address space of the core processes (examples include the Unix `mmap` and `ptrace` interfaces), to trace the actions that the program takes, or to monitor its inputs, outputs, procedure calls, and system calls.

- **Enforcement:** A white-box application of data structure repair would insert calls to the `repair` procedure at critical places in the core program, for example just

before the `put`, `get`, and `rem` procedures in our example. It is also possible to wait for the core to fail, catch the resulting exception, apply data structure repair in the exception handler, then restart the application from an appropriate place.

A gray-box implementation might use the Unix `ptrace` interface (see Section 2.5) or `mmap` to get access to the address space(s) of the core process(es). All of these mechanisms update the data structures directly in the address space of the core, heuristically attempting to preserve the information present in the original inconsistent data structure.

In general, we expect that enforcement strategies will attempt to perturb the state and behavior as little as possible. We anticipate the use of a variety of mechanisms that update the internal state of the core, cancel impending core actions or generate incorrectly omitted actions, or change the inputs or outputs of the core, components within the core, or the underlying system.

- **Logging Mechanism:** Our example simply prints out to `stderr` a trace of the instructions that it executes to eliminate inconsistencies. In general, we anticipate that the logging mechanism will vary depending on the needs of the application and that some developers may find it desirable to provide more organized logging support. Note also that logging is useful primarily for helping to provide insight into the behavior of the system. The log may therefore be superfluous in situations where it is undesirable to obtain this insight or it is impractical to investigate the behavior of the system.

We note one other aspect that this example illustrates. The problem in our example arose because the core (like many other software systems) handled a resource limitation poorly. Data structure repair enables continued execution with compromised functionality. But because no system can support unlimited resource allocation, even the best possible implementation must compromise at some point on the functionality that it offers to its clients if it is to continue executing.

Finally, there is another way to prevent `put` from failing because the free list is empty — change the implementation so that it checks if the free list is empty and, if so, skips the call to `put`. This mechanism is an instance of conditional code excision as discussed below in Section 5.4. Although this technique does not protect the core against the full range of potential data structure corruption problems, it does avoid a specific execution that is known to cause corruption.

2.3 Input Monitoring and Rectification

Our example program uses fixed-size character arrays to hold the `name` in each entry, but does not check for input names that are too large to fit in these arrays. It may therefore fail when presented with input names that exceed the array size. A standard way to fix this problem is to modify the core so that it either checks for and rejects excessively long names or uses dynamic memory allocation to correctly handle names of arbitrary length.

An acceptability-oriented approach might instead interpose a filter on the input. This filter would monitor the input stream to detect and eliminate inputs that would cause array overflow. Figure 4 presents the code for just such a filter. The filter truncates any token (where a token is a contiguous sequence of non-space characters) too long to fit in the `name` arrays from Figure 2. Using the Unix pipe mechanism to pass the input through this filter prior to its presentation to the core ensures that no token long enough to overflow these arrays makes it through to the core.

```
int main(int argc, char *argv[]) {
  int count = 0, c = getchar();
  while (c != EOF) {
    if (isspace(c)) count = -1;
    if (count < LEN-1) { putchar(c); count++; }
    else fprintf(stderr, "character %c discarded\n",
                 (char) c);
    if (c == '\n') fflush(stdout);
    c = getchar();
  }
}
```

Figure 4: Token Length Filter Implementation

There are several issues that arise when building these kinds of input filters:

- **Acceptability Property:** The developer must first determine the acceptability property that the filter should enforce. In our example, the property is simple: no token longer than `LEN` characters. For systems with real-time constraints, one twist is that the property may involve timing characteristics such as the frequency at which the inputs arrive.

  Note that the acceptability property in our example is partial in that it does not completely characterize the legal inputs. It instead captures a simple property that every input must satisfy to be acceptable. This simplicity enables the filter to avoid the complexity of performing a complete legality check on the input. The result is a simpler filter that requires less development effort and is less likely to have acceptability problems of its own.

- **Interposition:** The input filter must use some mechanism that allows it to observe and potentially process the input before passing it along to the core. In our example we use the standard Unix pipe mechanism to pass the input through the filter before it reaches the core. This black-box approach requires no modifications to the core at all.

  If the core uses procedure calls to obtain its input, a gray-box interposition mechanism may redirect the calls to invoke the filter's procedures, which then obtain and process the input before returning it back to the core. Ways to implement this redirection include providing a new implementation of the invoked procedure, using binary rewriting techniques [16] to change the invoked procedure, using operating systems mechanisms to redirect an invoked system call [2], or using aspect-oriented programming mechanisms to intercept method calls [15]. If the core uses memory-mapped I/O, it may be possible to use memory protection mechanisms to have the attempted I/O operations generate a memory protection fault. The provided fault handler would then implement the interposition. Other mechanisms may be appropriate for other input strategies.

- **Monitoring:** The monitor processes the input to detect inputs that do not satisfy the acceptability property. In our example, the monitor keeps a running count of the number of contiguous non-space characters. When it encounters a character that would cause this count to exceed `LEN`, it has detected an impending violation of the acceptability property. We expect input monitoring implementations to vary depending on the property that they are designed to check. Note that the monitor may maintain state to help it check the acceptability of the input. In our example, the `count` variable is an example of this kind of state.

- **Enforcement:** In addition to enforcing the acceptability property, the filter will typically attempt to keep the rectified input (heuristically) close to the original input. In our example, the filter truncates overly long tokens but otherwise leaves the input stream unchanged. Alternative filters might split overly long tokens by inserting spaces into the middle of long non-space character sequences or remove overly long tokens altogether. We rejected these alternatives because splitting or removing tokens appears to be more likely to confuse the core's input stream processing than truncation.

  In general, we expect that enforcement strategies will tend to be application dependent and heuristic. Error-correcting parsers insert keywords to make an input program parse [1]; spell-checkers use a dictionary of known words to correct misspellings. For systems whose acceptability properties involve real-time constraints, the enforcement mechanism may delay inputs that arrive too early and/or spontaneously generate replacements for inputs that arrive too late.

- **Logging:** In this example the filter simply logs the discarded characters. A more elaborate logging implementation might provide more information about the truncated tokens and the position in the input where the truncation occurred.

Note that in our example we applied input filtering at a well-defined interface consisting of input and output streams. It is possible, of course, to apply input filtering at arbitrary granularities throughout the software. Many implementations of standard libraries, for example, have extensive error checking that is designed to catch incorrect usage patterns (although the usual response is to signal an error, not to rectify the input and continue the execution).

2.4 Output Monitoring and Rectification

The core in our example, like many software systems, may produce unacceptable results under certain inputs. For example, the following input:

```
put x 10
put y 11
rem y
put x 12
rem x
get x
```

causes the core to generate the output `10 11 11 12 12 2` instead of `10 11 11 12 12 0` (a 0 returned from a `get` command indicates that there is no mapping associated with the name). The problem is that the `free` procedure, when invoked by the `put` procedure to remove the entry that implements the map from `x` to `10`, does not actually remove the entry. It instead puts the entry on the free list (which uses the `value` field to link together the free entries), leaving it also linked into the table of active entries.

One way to fix this problem is to change the implementation of the core. But (especially for larger, more complex components with sophisticated internal state and behavior) developing the understanding required to safely change this implementation may take an enormous amount of time and effort. And changing the core always raises the possibility of introducing new errors.

Figure 5 presents a component that enforces the acceptability property that any output must either 1) be between the minimum and maximum values inserted into the mapping, or 2) be 0. It enforces this property by creating two filters. The first filter (the `extract` procedure) processes the input to determine whether or not the next command is a `get`, `put`, or `rem`. The other filter (the `rectify` procedure) processes the output to record the minimum and maximum values put into the table. It uses these values to ensure that all outputs for `get` and `rem` commands are either between the minimum and maximum or 0; we call this activity *rectification*. The first filter uses the `channel` stream to pass the command information to the second filter. The `main` procedure sets up the channels, creates the two filters, and starts the core. Figure 6 graphically presents the resulting process structure.

When confronted with an out of bounds result from the core, the output filter replaces the result with 0. An alternative approach would be to simply replace the result with the minimum or maximum value. This approach might be preferable when any value in the table is acceptable. Consider our motivating usage context of a client that uses the core to map a bank of printer names to network addresses. Assuming the minimum and maximum values remain valid, the filter would ensure the delivery of the print job to some printer. If all of the printers are acceptable to the client, the result would be a system that could deliver acceptable results even in the face of unreliable behavior on the part of the core.

The code in Figure 5 uses a black-box approach: the implementation of the core remains unchanged. As may often be the case with such solutions, the management code required to implement the communication between the filters and the interpositions on the input and output streams dominates the code required to enforce the acceptability property. A white-box approach may require less code because the state and communication may be more easily accessible. The drawback, of course, is that the white-box code must execute in a more complex and presumably less well understood context.

Figure 7 presents a white-box reimplementation of the core `main` procedure from Figure 1. We have augmented this procedure to maintain the minimum and maximum values put into the mapping and to coerce out of bounds values to 0. The code is substantially smaller than the black-box code in Figure 5, but is embedded within the input and output processing code in the `main` procedure. As should be the case, the filter code is added around the edges of the core — it does not appear within the procedures (`put`, `get`, `rem`) that implement the primary functionality of the core.

```
void extract(int fd) {
  char cmd[LEN], name[LEN];
  int val;
  while (scanf("%s", cmd) != EOF) {
    if (strcmp(cmd, "put") == 0) {
      if (scanf("%s %d", name, &val) == 2) {
        printf("%s %s %d\n", cmd, name, val);
        fflush(stdout);
        write(fd, "p", 1);
        fsync(fd);
      }
    } else if (scanf("%s", name) == 1)  {
      printf("%s %s\n", cmd, name);
      fflush(stdout);
      write(fd, cmd, 1);
      fsync(fd);
    }
  }
}

void rectify(int fd) {
  int val;
  char c;
  static int min = MAXINT;
  static int max = 0;
  while (scanf("%d", &val) != EOF) {
    read(fd, &c, 1);
    if (c == 'p') {
      if (val < min) min = val;
      if (max < val) max = val;
    } else {
      if (val < min) val = 0;
      if (val > max) val = 0;
    }
    printf("%d\n", val);
    fflush(stdout);
  }
}

int main(int argc, int *argv[]) {
  int input[2], output[2], channel[2];

  pipe(input);
  pipe(channel);
  pipe(output);

  if (fork() == 0) {
    dup2(input[1], 1);
    extract(channel[1]);
  } else if (fork() == 0) {
    dup2(input[0], 0);
    dup2(output[1], 1);
    execv(argv[1], argv+1);
    fprintf(stderr, "io: execv(%s) failed\n", argv[1]);
  } else {
    dup2(output[0], 0);
    rectify(channel[0]);
  }
}
```

Figure 5: Black-Box Implementation of Min/Max Output Monitoring and Rectification

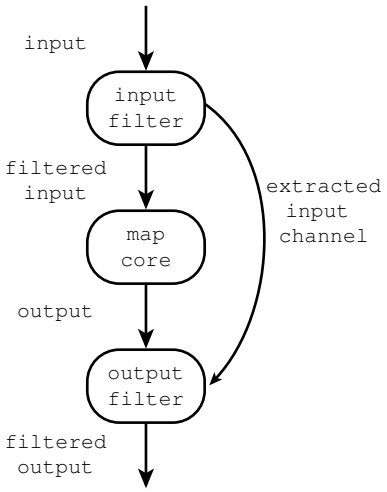

Figure 6: Process Structure for Black-Box Implementation of Min/Max Output Monitoring and Rectification

```
int main(int argc, char *argv[]) {
  char cmd[LEN], name[LEN];
  unsigned val;
  static int min = MAXINT;
  static int max = 0;
  initialize();
  while (scanf("%s", cmd) != EOF) {
    val = 0;
    if (strcmp(cmd, "put") == 0) {
      if (scanf("%s %u", name, &val) == 2) {
        put(name, val);
        /* record min and max */
        if (val < min) min = val;
        if (max < val) max = val;
      }
    } else if (strcmp(cmd, "get") == 0) {
      if (scanf("%s", name) == 1)  {
        val = get(name);
      }
    } else if (strcmp(cmd, "rem") == 0) {
      if (scanf("%s", name) == 1)  {
        val = rem(name);
      }
    }
    /* enforce acceptability property */
    if (val < min) val = 0;
    if (val > max) val = 0;
    printf("%u\n", val);
    fflush(stdout);
  }
}
```

Figure 7: White-Box Implementation of Min/Max Output Monitoring and Rectification

```
int scan(char name[], int val) {
  int i;
  for (i = 0; i < N; i++)
    if (strcmp(name, name(i)) == 0)
      return(value(i));
  return val;
}
```

Figure 8: Code to Scan `entries` Array

The primary new consideration when implementing combined input and output filters is the mechanism that the filters use to communicate. The black-box implementation in Figure 5 uses a Unix pipe to carry the extracted command information to the output filter. The white-box implementation in Figure 7 uses variables. Other potential mechanisms include network connections and shared memory segments. Other issues (such as interposition mechanisms, monitoring, and acceptability property enforcement) are similar to those discussed in Section 2.3.

2.5 Structure and Behavior

Further testing may reveal another anomaly — the following input:

```
put a 5
put c 6
rem a
get c
```

produces the output `5 6 5 0` instead of the output `5 6 5 6`. One hypothesis may be that the core is dropping mappings because it has exceeded its mapping capacity, but this hardly seems likely with only three mappings. It turns out that there is a different problem. When the `rem` procedure in Figure 2 removes a mapping, it removes (in addition to the removed entry) all entries in the list before the removed entry.

As with the previous deletion example from Section 2.4, it is possible to recode parts of the core so that the `rem` procedure does not delete the preceding list entries. The same disadvantages (the need to understand the core code, the potential to introduce new errors) apply.

An alternative is to apply an acceptability property that relates the data structures that the core builds to the outputs that it generates. Specifically, the `entries` array contains the mappings that the `get` procedure makes available to the client. An appropriate acceptability property is therefore that `get` should never return 0 if there is a mapping present in the `entries` array for the name passed as an argument to the `put` command. One can therefore view the `table` data structure as (like a cache) providing quick access to a set of mappings that may provide an acceptable answer. But if this data structure fails to provide an answer, the implementation should examine the `entries` array directly to find the mapping.

Figure 8 presents code that scans the `entries` array to find the mapping for a given name. A white-box implementation of the acceptability property discussed above would simply invoke the the `scan` procedure whenever the `get` procedure (from Figure 2) returns 0. With this change, the implementation correctly returns `6` as the result of the `get c` command for the input discussed above.

```
void extract(int fd) {
  char cmd[LEN];
  char name[LEN];
  int val;
  while (scanf("%s", cmd) != EOF) {
    if (strcmp(cmd, "put") == 0) {
      if (scanf("%s %d", name, &val) == 2) {
        printf("%s %s %d\n", cmd, name, val);
        fflush(stdout);
        write(fd, name, strlen(name));
        write(fd, "\n", 1);
        fsync(fd);
      }
    } else if (scanf("%s", name) == 1)  {
      printf("%s %s\n", cmd, name);
      fflush(stdout);
      write(fd, name, strlen(name));
      write(fd, "\n", 1);
      fsync(fd);
    }
  }
}

int main(int argc, int *argv[]) {
  int input[2], output[2], channel[2], pid;

  pipe(input);
  pipe(channel);
  pipe(output);

  if (fork() == 0) {
    dup2(input[1], 1);
    extract(channel[1]);
  } else if ((pid = fork()) == 0) {
    dup2(input[0], 0);
    dup2(output[1], 1);
    execv(argv[1], argv+1);
    fprintf(stderr, "st: execv(%s) failed\n", argv[1]);
  } else {
    dup2(output[0], 0);
    rectify(pid, channel[0]);
  }
}
```

Figure 9: Implementation of `name` Extractor for `entries` Filter

It is also possible to develop a gray-box implementation. One approach interposes an output filter and scans for 0 outputs. It then uses the Unix `ptrace` package to gain access to the address space of the core process and examine the contents of the `entries` array to determine if it should replace 0 with some other value. As in the example in Section 2.4, the output filter needs some information from the input stream. It is possible to use the same solution: an input filter that extracts the information and uses a separate channel to pass it along to the output filter. Figure 9 presents the code for this input filter. The `extract` procedure implements this filter — it extracts the names from the input stream and passes them to the output filter. The `main` procedure creates the input filter, the core, and the output filter and sets up the connections between them.

Figure 10 presents the code for the output filter. The `addr` procedure uses the `readelf` command to extract the symbol table information for the core process and find the address of the `entries` array within this process. It uses the `getLine` procedure to parse the symbol table information. The `rectify` procedure implements the output filter. It

```
int getLine(FILE *f, char buf[], int len) {
  int i = 0, c;
  while (((c = fgetc(f)) != EOF) && (i < len-1)) {
    buf[i++] = c;
    if (c == '\n') break;
  }
  buf[i] = '\0';
  return(i);
}

#define SIZE 256

int getAddr(int pid, char sym[]) {
  char cmd[SIZE], buf[SIZE];
  int addr = 0;
  sprintf(cmd, "readelf -s /proc/%d/exe", pid);
  FILE *f = popen(cmd, "r");
  while (getLine(f, buf, SIZE) != 0) {
    if (strstr(buf, sym) != NULL) {
      int i = 0, j = 0;
      while ((buf[i] != ':') && (buf[i] != '\0')) i++;
      i++;
      sscanf(buf + i, "%x", &addr);
    }
  }
  return addr;
}

int getEntries(int pid, int offset) {
  int i;
  for (i = 0; i < sizeof(entries) / sizeof(int); i++) {
      ((int *) entries)[i] =
        ptrace(PTRACE_PEEKDATA, pid, offset+i*sizeof(int));
      if (((((int *) entries)[i]) == -1) && (errno != 0))
        return -1;
  }
  return i;
}
void rectify(int pid, int fd) {
  int val;
  int offset = getAddr(pid, "entries");
  char name[LEN];
  int i;
  int stat;

  while (scanf("%d", &val) != EOF) {
    for (i = 0; i < LEN; i++) {
      read(fd, &name[i], 1);
      if (name[i] == '\n') break;
    }
    name[i] = '\0';
    if (val == 0)
      if (ptrace(PTRACE_ATTACH, pid, 1, 0) != -1) {
        if ((waitpid(pid, &stat, 0) != -1) &&
            (getEntries(pid, offset) != -1))
          val = scan(name, val);
        ptrace(PTRACE_DETACH, pid, 0, 0);
      }
    printf("%d\n", val);
    fflush(stdout);
  }
}
```

Figure 10: Implementation of `entries` Filter

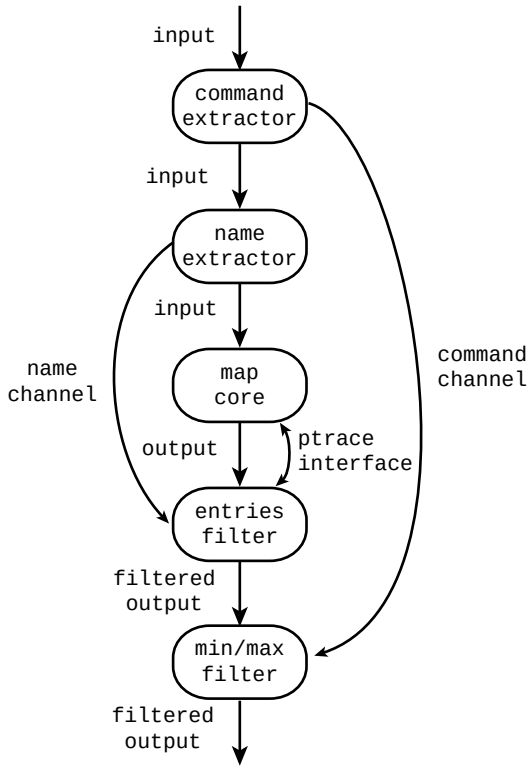

Figure 11: Process Structure for `entries` Filter

examines the output stream and, when it finds a 0 output, uses the `ptrace` package to attach to the core process. It then invokes the `load` procedure to copy the contents of the `entries` array from the core into its own local version and invokes the `scan` procedure to find an appropriate entry (if one exists) and return the value from this entry.

Figure 11 graphically presents the process structure that the `entries` filter generates when combined with the `min`/`max` filter from Section 2.4. There are two input filters: the first extracts the command information from the input stream to pass it to the `min`/`max` output filter; the next extracts the name information to pass it to the `entries` filter. The entries filter uses the `ptrace` interface to access the data structures in the core.

This example illustrates how to use the structures that the core builds as a foundation for stating and enforcing acceptability properties. Instead of specifying output properties as a function of the inputs, it is instead possible use properties that capture the relationship between the outputs and the structures that the core builds as it processes the inputs. Leveraging the core data structures may substantially reduce the complexity and implementation effort required to enforce some acceptability properties — without these data structures, the enforcement component may be forced to reimplement much of the data structure functionality to store the input information that it needs to enforce its acceptability property.

It may also be useful to enforce acceptability properties involving the inputs and the resulting core data structures. Many of these properties can be viewed as capturing acceptability properties of the core state as a function of the core's interaction history with the environment.

There is a conceptual connection between some transaction processing failure recovery algorithms and resilient computing mechanisms that enforce acceptability relationships between the input and the state. Transaction processing systems typically maintain a log of all operations performed against the database. Conceptually, the database is simply a data structure (like the `table` in our example) that the transaction processing system uses to accelerate access to the data (whose primary representation is in the log). When the database fails, the recovery system may rebuild the database by replaying part or all of the log. One can therefore view the recovery algorithm as enforcing a correctness relation between the input history in the log and the structure of the database.

3. MORE TECHNIQUES

In Section 2 we presented a range of acceptability-oriented computing techniques in the context of an example program. We next discuss other potential techniques and applications.

3.1 Data Structure Consistency

The data structure repair implementation discussed in Section 2.2 consists of a set of hand-coded procedures that dynamically detect and repair violations of key data structure consistency properties. While this code satisfies its repair goals, hand-coded data structure repair code can be difficult to develop (because the developer cannot assume that the data structures satisfy their normal invariants and because the final code can be quite complex). And it can be difficult to examine the code to determine which properties it does and does not enforce.

We have therefore developed a new specification-based approach to enforcing data structure consistency properties [10]. The goal is to reduce development effort and place the field on a firmer foundation by making the enforced consistency properties explicit. Our system accepts as input a specification of the desired data structure consistency properties. During the execution of the program, it traverses the data structures in the heap to find specific objects that violate the consistency properties. When it finds a violation, it applies repair actions that coerce the data structures back into a consistent state. It is also possible to apply the inconsistency detection and repair process to a persistent data structure (such as an application data file) just before a program attempts to access the data structure.

One complication is that correct programs may temporarily violate the consistency constraints during data structure updates. We address this complication by enabling the programmer to identify the points in the program where he or she expects the data structures to be consistent. The tool enforces data structure consistency at only those points.

A potential concern is that repair followed by continued execution may lead to the silent generation of unexpected results. Our approach generalizes to support the tagging of all values computed using repaired data. These tags would support user interfaces that suitably distinguish results produced from repaired data, alerting the user to potentially questionable results.

We have applied our system to several applications — an air-traffic control system, a multi-player game, a simplified version of the Linux ext2 file system, and Microsoft Word documents. We found that the specifications had acceptable development overhead and that the resulting repair algorithms enabled the systems to effectively recover from otherwise fatal errors. We found the air-traffic control system to be a particularly interesting application. It tracks hundreds of aircraft and implements features that provide different kinds of functionality (flow visualization, route planning, conflict detection, trajectory projection, and scheduling). Without data structure repair, an error in any one of the aircraft or any one of the features can cause the entire system to fail, denying the controller access to any of its functionality. Data structure repair enables the system to continue to execute and provide normal functionality for almost all of the aircraft and features.

3.2 Process Structure Consistency

Many distributed systems are structured as collections of communicating processes. Like data structures, these process structures often come with consistency properties. These properties typically become violated because processes (or the machines that host them) fail, hang, or become inaccessible. It is possible to augment these programs with additional components that monitor the process structure for inconsistencies caused by the presence of failed, inaccessible, or duplicated processes. Like most other monitoring components, these components could be manually coded or automated generated from a specification of relevant consistency properties.

Resilient computing techniques might repair inconsistent process structures by regenerating failed or inaccessible processes and shutting down duplicate processes (after rerouting connections to a chosen primary process). In our map example from Section 2, one might isolate the core behind an input and output filter. If the core fails, the filters would cooperate to restart it. An extension might record `put` commands, then replay these commands to reinitialize the new core. Safe exit techniques might shut down the remaining processes after taking appropriate actions to move to a stable state.

3.3 Configuration Consistency

Many programs and operating systems must be configured to operate correctly. System administrators are typically responsible for (more or less manually) configuring systems and maintaining the configurations to ensure that they support the needs of the user and are consistent with the surrounding computing infrastructure. A configuration can become inconsistent in a variety of ways: updates to the computing infrastructure in which the system is embedded, botched, incompatible, or undesirable software installations, or attacks such as software viruses. Different applications may also require different configurations. It is possible to augment these systems with components that monitor the configuration to detect inconsistent or suboptimal configurations. Resilient computing approaches might reconfigure the system automatically; safe exit approaches might notify a system administrator and block any actions affected by the inconsistent or suboptimal part of the configuration.

3.4 Library Usage Consistency

Many libraries come with implicit consistency requirements on the sequence in which the operations must be invoked. Standard examples include requiring files to be opened before reading or writing and ordering constraints (such as two phase locking) on lock and unlock operations.

It is possible to monitor the execution of the program to detect violations of these ordering requirements. When a violation is detected, resilient computing techniques could automatically insert or remove library calls to move a program from inconsistent to consistent execution sequences. For example, if a client passes an invalid file descriptor to a `read` or `write` operation, the operation can simply open a default file and perform the operation on that file.

4. ISSUES

Acceptability-oriented computing as presented in this paper is largely untested and it is unclear to what extent it will succeed. It has the potential to significantly change many of the engineering and social trade-offs associated with software development, in both obvious and quite subtle ways. The overall success of the technique will depend, in large part, on the interaction between these changes.

4.1 Continued Execution After Repair

Resilient computing systems are strongly biased towards continued execution even after the confirmed detection of an incorrect or unexpected execution in part of the system. In many situations it may be far from clear when this continued execution is more desirable than the more traditional approach of terminating the computation, then relying on human intervention to restore the system to an acceptable operating state, or on the slightly more powerful concept of monitoring for unacceptable states or behavior, then executing a safe exit strategy.

Resilient computing should therefore be most easily accepted in situations where continued execution, even if compromised, has clear advantages over outright failure. Examples of such situations include autonomous systems that must execute largely without human intervention, systems with long recovery times and stringent availability expectations, and when (such as in image processing software) the basic acceptability of a proffered result is immediately obvious upon inspection. Resilient computing may be useful when the software is known to execute incorrectly in certain situations, but the developers are unwilling to risk the system disruption that might result from fixing the incorrect code, preferring instead to repair any damage after it occurs. It may also be useful for systems with multiple largely independent components — repairing an otherwise fatal flaw in one component may enable the system as a whole to continue its execution, with the remaining components continuing to provide their normal functionality.

Resilient computing may be less acceptable when (as is the case for many numeric calculations) the acceptability of the result is difficult to determine by simple inspection, especially when a repaired error may cause the produced result to satisfy stated consistency properties but still deviate from a correct or acceptable result. Ways to attack this problem include the use of credible computation (computations that produce a checkable proof of the correctness or acceptability of their results) [20, 21] or explicit tagging of results that depend on repaired state or computation.

4.2 Easy Adoption Path

Technologies that require wholesale changes to current practice typically have a difficult adoption path, even when it is clear that they offer substantial advantages. Technologies that require few changes typically have a much easier time. This dichotomy is especially acute for technologies that involve a large, stable installed base. The installed base of C and C++ programmers, for example, has in the past comprised a substantial (and in many cases unsurmountable) obstacle to the adoption of new programming languages, even when these languages were clearly superior.

Acceptability-oriented computing has a very easy adoption path. It can be incrementally added to preexisting core software written in standard legacy languages such as Fortran, C, or C++. There are no serious obstacles that significantly complicate its use in multilingual software systems. Only those developers actively producing the acceptability monitoring and enforcement components or specifications need even be aware of its presence in the system; other developers can continue to obliviously use the same practices they have always used. In fact, resilient computing techniques may even reinforce current software development practices — by increasing the reliability of software produced using these practices, they may postpone or eliminate the need to move to new practices. Even if organizations wind up never using its failure recovery features in deployed systems, the inconsistency detection features may facilitate the detection and localization of errors during the development phase. And the exercise of identifying the key acceptability properties may help the organization to understand the properties that the system should preserve and to focus their development effort. Acceptability-oriented computing therefore has the profile of technology that can proliferate very rapidly throughout the software development ecosystem once its advantages become apparent.

4.3 Poorly Understood Software Components

With current development practices, the use of potentially unreliable or poorly understood components introduces a substantial amount of uncertainty about the behavior of the system. In particular, it can be difficult or impossible to reason about what circumstances might cause unacceptable behavior to occur, what form the unacceptable behavior might take, and the ultimate consequences of the unacceptable behavior. This situation is unfortunate, because these kinds of components may, in many circumstances, provide substantial functionality and development advantages. For example, machine learning, neural networks, and software evolution (all of which produce software that may be difficult or impossible to understand) can efficiently deliver functionality that is difficult to obtain at any cost using standard development techniques. And many incompletely developed or debugged software packages also implement (albeit partially) valuable functionality.

Acceptability-oriented computing may be able to eliminate much of the uncertainty associated with the use of such potentially unreliable or poorly understood components. The acceptability enforcement mechanisms bound the range of potential system behaviors, making it possible to reason concretely and precisely about the potential impact of these components. Acceptability-oriented techniques therefore promise to enable a much more aggressive approach to software reuse and to the incorporation of poorly-understood software into the system. The end result may be a dramatic reduction in the difficulty of building acceptable software systems and corresponding increase in the amount of functionality that we can incorporate into an acceptable system.

4.4 Appropriate Engineering Investment

In almost every software development project, some parts of the system are, in practice, more important than others. An efficient development process would clearly devote more care to the engineering of these parts of the system. But traditional requirements analysis processes fail to prioritize the requirements, leaving the developers without any guidance as to how they should most efficiently invest their engineering resources to deliver the most acceptable system within their means.

Because acceptability-oriented computing provides such a prioritization, it may trigger the development of new software engineering processes that direct different amounts of engineering resources to different tasks based on the perceived importance of each task to the overall acceptability of the final system.

4.5 Impact on Software Quality

If acceptability-oriented computing delivers on its promise to make systems execute acceptably in spite of errors, it will also reduce the incentive to produce reliable software in the core. One potential outcome would be a substantial reduction in the amount of engineering required to produce an acceptable piece of software. Instead of sinking a large amount of engineering resources into finding and eliminating programming errors in the core software system, the organization would simply accept a larger number of core software errors, then rely on the outer layers to compensate for their presence.

The success of this particular scenario depends on the assumption that the production of close to perfect software is more expensive than the production of software that is, in the presence of a sufficiently sophisticated acceptability enforcement mechanisms, just acceptably flawed. But note that even in a resilient system, the core software must reach a certain level of reliability before the system as a whole can function acceptably. And it is also clear that, in a variety of settings that range from manufacturing [8] to personal relationships [17, 7], the mere presence of mechanisms that are designed to detect and compensate for human error has the effect of reducing the effectiveness of the participants in the setting and, in the end, the overall quality of the system as a whole. A potential explanation is the *bystander effect* — that the participants start to rely psychologically on the error detection and compensation mechanisms, which reduces their motivation to reduce errors in their own work. In fact, the most successful manufacturing process in the world today (lean production) can be viewed as designed to magnify the negative impact of each error on the process and to therefore increase the vulnerability of the system as a whole to these errors [23]. The rationale is that this approach makes each error immediately obvious and serious and therefore immediately addressed. The goal is also to increase the motivation of the human participants to reduce their individual errors to the lowest possible level and to apply their efforts to eliminating errors within the system as a whole.

The knowledge that they are developing software for an acceptability-oriented system with error detection and compensation mechanisms may therefore reduce the motivation and ability of the developers to produce quality software. They may even be incapable of delivering software that satisfies even the reduced requirements for successful inclusion in a resilient system. It may turn out to be the case that, in the end, the best way to build reliable software is to place each developer in a position where 1) each error that he or she makes will have serious consequences, and 2) he or she is solely responsible for ensuring the absence of errors in the parts of the system that he or she develops. The error monitoring and compensation mechanisms at the heart of acceptability-oriented computing are obviously counterproductive in such a scenario.

A related problem is that developers may subconsciously adjust their activities and work habits so that, across a very broad range of development processes, the same amount of effort is required to deliver a minimally acceptable software system regardless of the development methodology or system structure. In this case acceptability-oriented computing would not produce a better system, and its more sophisticated structure could be seen only as a disadvantage.

4.6 Automatic Property Generation

Acceptability-oriented computing requires the presence of identification of acceptability properties. We expect that, compared to the specification for the core software system, these specifications will be quite small and relatively easy for developers to produce. However, it may be desirable to produce these specifications automatically to reduce the development burden.

One approach is to statically analyze the program to extract likely acceptability properties. This analysis would obviously need to be unsound (if not, the core program would never violate the properties). Of course, the trade-off is that an unsound analysis has the freedom to generate more ambitious and potentially more useful properties.

Another approach is to monitor the execution of the code to learn key input, output, and data structure properties [9, 11]. The system would then enforce these properties for all executions. Potential issues include ensuring that the system observes enough behaviors so that it does not inflexibly enforce a overly narrow set of properties and eliminating undesirable properties generated by unacceptable executions.

4.7 Monitoring and Repair Overhead

Acceptability-oriented computing may introduce additional overhead in the form of the monitoring and repair software. The potentially most serious form of this overhead occurs when the system suffers from a recurrent error that is repeatedly repaired, and therefore masked, by the error compensation software. In this scenario, the system might spend most of its time executing the error recovery software, with the human users oblivious to the true source of the compromised performance and therefore unable or unmotivated to solve the problem. In the worst case, the problem could become progressively more serious until the recovery mechanisms were finally unable to compensate. In the absence of such monitoring and repair, of course, the error would become immediately obvious and problematic to its users, who would then repair the system and eliminate the error.

Logging mechanisms are designed to attack this problem — they produce information about repaired errors in part to alert the human users or administrators to the potential problem. It is unclear how effective such warnings would be in practice, given the discipline required to take action in response to warnings of errors that do not immediately interfere with the operation and use of the system.

An example of this kind of problem arises in intermittently faulty hard disks, which may require multiple reads of the same disk block to successfully retrieve the data. The system may spend much of its time repeatedly attempting to read the same disk block, but this fact is hidden because the retries are transparent to the client of the low-level disk hardware. Despite the fact that such intermittent errors are typically logged and often indicate an impending total disk failure, many users do not regularly monitor the log to preemptively replace faulty disks before they fail.

4.8 Acceptability Requirements Identification

One of the prerequisites for applying resilient computing is identifying key acceptability properties, which typically take the form of structural consistency properties or basic behavioral requirements. Many software engineers believe that explicitly identifying and documenting these kinds of properties improves the software development process, even if they are never explicitly used once they have been identified and documented. Potential benefits include obtaining a better understanding of the system under development and facilitating the communication of this understanding within the development team. What is now understood as a key advantage of acceptability-oriented computing (a more reliable system through automated compensation for errors and faults) may prove, over time, to be most important as an incentive that convinces organizations to explicitly document key system acceptability requirements.

4.9 Error Tracking

It is often desirable to be able to understand why a system behaves the way it does. In general, we expect that resilient computing techniques may complicate this analysis — the additional layers may obscure or even completely mask the original sources of errors. One way to attack this problem is to ensure that the monitoring mechanisms adequately log all of the impending acceptability violations that they detect. The log entries should help the developer trace interesting events in the system and reconstruct potential reasons for its behavior. The logs may even make a system augmented with acceptability-oriented mechanisms easier to understand than the original, unaugmented system.

4.10 Close to Perfect Software

The recent proliferation and success of bug-finding software development tools and safe programming languages such as Java raises the possibility that developers may be able, in the near future, to dramatically increase the quality of the software that they produce. It is possible that this increase may be large enough to render the additional benefit of the resilient approach small enough to be not worth the additional cost and complexity.

4.11 More Creative Software

Acceptability-oriented programming may free developers from the tyranny of perfection, enabling the creative development and deployment of code with only a hazy, intuitive idea of what the code will do in some or even most situations. Instead of engaging in detailed reasoning about what will happen, programmers may simply adopt a more empirical approach in which they quickly throw together some code that they feel is close to what they might want, try it in the system to see how it works out, then incrementally modify it until it exhibits approximately the desired behavior most of the time, relying on the acceptability enforcement mechanisms to avoid the introduction of unacceptable behavior.

The potential advantages of this approach include faster exploration of the implementation space and a reduced need for developers to understand the software system. It may also reduce the cognitive capabilities required to function effectively in the system, enabling less competent developers to make meaningful contributions.

5. EXTENSIONS

As described so far, resilient computing can be seen as a way to inject redundancy into a system to increase acceptability. Its focus on influencing the execution via data structure updates and input and output filters often leaves the core software untouched. But resilient computing techniques may make some dramatically different approaches to developing and manipulating the core practical.

5.1 Failure-Oblivious Computing

Consider a safe language with null pointer checks and array bounds checks. If the program fails one of these checks, the run-time system throws an exception. Because programmers usually can't be bothered or don't know how to handle these exceptions in a more intelligent fashion, the program usually terminates.

Instead of terminating, the system can instead replace the execution of the offending statement with a simple default action that enables the system to continue to execute without throwing an exception. So, for example, if the program attempts to load a value from a null reference, the system might return a default, previously observed, or random value as the result of the load and continue to execute without throwing an exception. The system could also simply discard values stored to illegal addresses. One can view this approach as applying resilient computing at a low level with simple automatically generated recovery actions and no specification required from the developer.

One justification for this approach is that the program may produce many results, only some of which may be affected by any given error. But all of the results need the flow of control to pass through the computation that generates them. One artifact of the standard sequential computing paradigm is that the flow of control is artificially multiplexed between independent computations, with a failure in any one of the computations causing the flow of control to be unavailable to the others. A similar artificial resource constraint occurs in token ring networks and may be one of the reasons that these networks have proved to be less popular than other kinds of networks.

An alternate approach would apply techniques from lazy programming languages to view the computation as a (lazily generated) dependence graph containing computations that lead to the results. If the dependence graph for a given result contains an error, the program simply does not produce that result. Independent computations, of course, would still produce their results. Given this insight, it is possible to translate the basic philosophy of the approach back to sequential languages. This approach would use the sequential flow of control only to order interfering accesses (two accesses interfere if they access the same location and one of the accesses is a write). The failure of one computation would not prevent independent computations from successfully producing

their results. It is, of course, possible to generalize this idea to allow independent computations to produce results even if one of the computations does not terminate. One generalization would reason statically about the complete set of effects performed by potentially nonterminating loops to execute independent computations following the loop even in the absence of loop termination. An alternative generalization would simply forcibly exit loops that fail to terminate either within some predetermined number of iterations or after substantially exceeding previously observed numbers of executions.

5.2 Code and Input Variation

A traditional approach to surviving faults is to periodically checkpoint, then react to failure by rolling back to a previously checkpointed state, then restart. The disadvantage, of course, is that the system will simply fail again if presented with the same input.

One way to avoid this problem is to perturb the input in some way to elicit different behavior from the system. The specific perturbation will depend on the context, but examples could include varying the timing and order of inputs from independent sources, removing parts of the input, or transforming the values in the inputs in some way. Of course, this input transformation may cause the program to generate different results — but then, this is the whole point: to change the behavior of the program to move it away from failure.

Another alternative is to use fault injection to perturb the execution of the program for some time after failure. Potential targets of fault injection include modifying the data structures and changing the direction of conditional branches. The data structure modification could be guided by consistency property specifications if they are available. This approach would reduce the chance of the recovery mechanism causing new failures.

5.3 Code Omission

Much of the code in the core typically falls into one of two categories: common case code that implements the basic functionality and executes on almost all inputs, and uncommon case code that is there to correctly handle very rare combinations of events. It is well known that uncommon case code complicates the structure of the program and can make the program substantially larger and more difficult to understand.

Resilient computing opens up the possibility of simply omitting the uncommon case code. In most cases the omission would have no effect because the omitted code would not have executed. If the code would have executed, the acceptability enforcement mechanisms may produce a result that (while potentially different from the correct result that the omitted code would theoretically have produced) the user can live with.

The potential benefits of this approach include smaller, simpler code that is easier and cheaper to develop and modify. These advantages may enable the developers to produce common-case code with fewer errors, in which case the system as a whole may be more reliable than a system which contains code for the uncommon cases. The mere elimination of the uncommon case code may also, by itself, increase the reliability of the system. Because complex, infrequently executed code is notoriously difficult to develop without errors, its elimination (even with no replacement) may increase the overall reliability of the system.

As discussed below in Section 6.5 the use of garbage collection can be seen as an instance of code omission (an application that uses garbage collection omits all of the code used to support explicit memory management). The benefits of that garbage collection delivers (elimination of references and memory leaks; increased reliability) provide an indication of the potential advantages that code omission may offer when applied to other aspects of the computation.

5.4 Code Excision

In some cases the mechanism that detects acceptability violations may be able to identify the code that caused the violation. If a given piece of code is responsible for many violations, it may be beneficial for the acceptability enforcement mechanism to excise the code from the system so that it will not continue to cause problems. The excised code may be replaced by alternate code or simply removed from the system with no replacement of the functionality that it was intended to provide. The potential benefits may include reduced repair overhead and fewer faults propagated to otherwise acceptable parts of the system.

A generalization of code excision identifies properties whose satisfaction causes the code to fail, then uses `if` statements to skip the code in question when these properties are satisfied. We call this generalization *conditional code excision*; the conditional skipping of `put` invocations when the free list is empty as discussed in Section 2.2 is an example of this technique.

5.5 Code as an Inconsistency Generator

A correct data structure update can often be viewed as creating intermediate inconsistent data structures, then repairing the inconsistency. Given an automated consistency restoration mechanism, it may be possible to simply omit the explicit consistency restoration code, relying on the automated consistency restoration algorithm to restore the consistency properties and complete the update. Once again, the potential advantages include smaller and simpler code.

This approach may turn out to be especially useful for data structure initialization. The standard approach is to functionally decompose the initialization to match the decomposition of the data structure. Unfortunately, there are often complex interdependences between these different initialization components, which complicates the staging of the initialization. In the worst case, the need to eliminate cyclic initialization dependences can force counter-intuitive code refactorings, with the initialization code remaining brittle in the face of incremental data structure changes.

An alternative approach is to declaratively specify the consistency properties for initialized data structures, eliminate the explicit data structure initialization code, then invoke the data structure consistency enforcer after an allocation of the initial item in the data structure. The consistency enforcer will automatically generate a legal initialization sequence, eliminating the need for the developer to produce code that correctly orders the initialization steps.

5.6 Core Elision

All of the techniques discussed so far eliminate part of the system but leave other parts intact. The extreme logical endpoint is to simply eliminate all of core code so that the

system consists of nothing more than a set of acceptability specifications written in a variety of languages. We expect this approach to work best for relatively simple systems, and in fact some work in model-based computing and domain-specific languages can be seen as an instance of this core elision technique.

6. RELATED WORK

We next discuss several existing techniques that can be seen as instances of acceptability-oriented computing. We focus on resilient computing techniques. Most deployed systems contain embryonic safe exit techniques in the form of assertions or input safety checks, although as mentioned above in Section 1.3, we expect that these systems would benefit from an increased awareness and understanding of explicit acceptability properties, a more organized approach to detecting impending acceptability violations, a larger effort devoted to identifying more appropriate safe exit strategies, and, when appropriate, the application of resilient computing techniques. We note that monitoring for impending acceptability violations is a key component of acceptability-oriented computing. Monitoring is a well-established field of computer science; monitoring techniques have been developing in a wide range of fields and deployed to accomplish a variety of goals.

6.1 Hand-Coded Data Structure Repair

Data structure repair has been a key part of two of the most reliable software systems ever built: the IBM MVS operating system [18] and the software for the Lucent 5ESS switch [13]. Both of these systems contain a set of manually coded procedures that periodically inspect their data structures to find and repair inconsistencies. The reported results indicate an order of magnitude increase in the reliability of the system [12].

These successful, widely used systems illustrate the utility of performing data structure inconsistency detection and repair. We view the use of declarative specifications for data structure repair (see Section 3.1) as providing a significant advance over current practice, which relies on the manual development of the detection and repair code. The declarative approach enables the developer to focus on the important data structure consistency constraints rather than on the operational details of developing algorithms that detect and correct violations of these constraints. The expected result is a substantial reduction in the amount of effort required to develop reliable inconsistency detection and repair software.

6.2 Persistent Data Structures

Persistent structures are an obvious target for many inconsistency elimination algorithms. An inconsistency in a small part of the system can prevent the system from accessing any of the data at all. And the standard default error recovery technique, rebooting, does not eliminate the problem since the inconsistencies persist across reboots. File systems, for example, have many characteristics that motivate the development of automatic repair programs (they are persistent, store important data, and acquire disabling inconsistencies in practice). Developers have responded with utilities such as Unix fsck and the Norton Utilities that attempt to fix inconsistent file systems. Databases also have many characteristics that justify inconsistency detection and repair, and database researchers have investigated a variety of integrity management techniques [6, 5, 22]. These techniques update the relations in the database to enforce database consistency constraints.

6.3 Partial Reboots

Researchers have developed a technique that exploits the structure in component-based systems to avoid complete system reboots [4]. When the system starts to exhibit anomalous behavior, they apply techniques that attempt to reboot a minimal set of components required to restore the system to normal operation. The goal is to minimize recovery time by leaving as much of the system as possible intact.

6.4 Application-Specific Techniques

The utility of the acceptability-oriented approach can be seen in several cases in which developers have opportunistically applied such techniques to improve their systems. We are aware, for example, of avionics software with two flight control systems: one is a venerable, fairly conservative, and well-tested version, while the other is a newly developed, more sophisticated and aggressive version. The acceptability property is that the aggressive version must keep the aircraft within the same flight envelope as the conservative version. This strategy enables the developers to apply more sophisticated flight control algorithms without compromising the safety of the aircraft. We are also aware of a graphics rendering package that simply discards problematic triangles instead of including complex special-case code that attempts to render the triangle into the scene [14]. Embedded systems developers have used layered approaches that apply safe exit strategies to increase the probability that the system behaves acceptably even when confronted with unanticipated situations [19]. Given the utility of acceptability-oriented techniques, we expect that they have been applied in an application-specific manner in many other systems.

6.5 Garbage Collection

One reasonable acceptability property is that all memory in a system is either reachable via the application's data structures or present in the free list and available for allocation. Garbage collection can be seen as a resilient computing technique that enforces this property. From this perspective, developers who use standard garbage-collected languages such as Java can be seen as adopting a deliberate code omission strategy as discussed in Section 5.3. Specifically, the developers omit all of the code that is designed to support, enable, and perform explicit memory management. This perspective highlights some of the potential advantages of code omission. Explicit memory management code is notoriously difficult to get correct and is a serious source of problems in many deployed software systems. Removing this code and replacing it with a single uniform implementation that enforces a well-defined acceptability property can produce a substantially more reliable system.

7. UNRELATED WORK

Reliability has been a central issue in computer science ever since the very first computers were built. Early research in the area tended to focus on ways to enable the system to continue to operate after sustaining various kinds of physical damage. The standard approach is to apply some form of redundancy to enable the system to recognize or even reconstruct damaged data. This redundancy is sometimes

connected to mechanisms that allow the system to recompute any results lost or incorrectly computed because of the damage. We identify this research as unrelated research because its primary goal is to simply preserve the correct execution of the program in the face of damage or errors, not to actively monitor and change the execution to ensure that it satisfies acceptability properties.

7.1 Physical Redundancy

Physical redundancy provides the system with multiple copies of its basic components, enabling the system to continue to operate even if some of the components become inoperable. This basic idea can be applied at all levels of the system design, from the basic logic gates in the computer system to larger components such as processors and memories. Since broken hardware components may not be able to recognize that they are no longer functioning correctly, there is often some way to compare results from all components and choose the result that is perceived to be most likely to be correct. The standard mechanism is to use majority voting, which obviously works best with at least three versions of each potentially faulty component.

7.2 Information Redundancy

The basic idea behind information redundancy is to replicate information to enable the system to reconstruct missing or damaged data. The standard approach is to use parity or more sophisticated versions of error correcting code. This approach can be applied in space (redundancy in the bits in memory) or in time (redundancy in bits transmitted). The primary downside is the extra physical resources (memory or bandwidth) required to apply the technique. If the technique supports detection but not correction, another potential downside is decreased reliability. The standard response to the detection of an uncorrectable error is to terminate the computation. Because many of these errors may have a minimal effect on the overall system, it may be better for the system to simply continue to execute through the error.

7.3 Computation Redundancy

The basic idea behind computation redundancy is to replicate computation to enable the system to recognize and discard any incorrect results. One flavor basically boils down to physical redundancy in that the same computation is replicated on multiple hardware platforms. If one of the platforms sustains damage and produces an incorrect result, a comparison with the other results will reveal the discrepancy. The standard response is to discard the incorrect result and, in some circumstances, initiate a repair action.

Another way to replicate computation is to produce multiple different implementations of the same specification. The idea is that if one implementation is incorrect and produces an incorrect output, the other implementations will most likely not suffer from the same error and will produce correct output. As in hardware computation replication, a comparison of the results enables the system to recognize and discard the incorrect result. Note that the success of this approach relies on developers producing independent errors. In practice, even independent developments have an annoying tendency to produce implementations that often fail on the same inputs [3]. Another potential problem is that the specification may be incorrect.

7.4 Checkpoint and Reboot

A standard approach to corrupted or damaged state is to rebuild the state from scratch by simply rebooting the system. This approach works extremely well provided the corrupted state is discarded during the reboot process, it is acceptable to lose of the discarded state, and it is acceptable to disable the system during the period of time when it is rebooting.

This approach can be augmented with periodic checkpointing, enabling the reboot to lose less information by starting from recent state. One can view actions that save state to disk as periodically checkpointing parts of the state to ensure that it persists across failures. One potential problem with checkpointing is that the checkpoint may silently contain corrupted or inconsistent state.

At first glance, rebooting may seem to be pointless — after all, won't the system just fail again once it is rebooted and asked to retry the task that elicited the error? It turns out that, in practice, many software errors occur only under unusual combinations of circumstances, and that rebooting the system and retrying the task often changes the state and various aspects of the inputs (such as their timing) enough to avoid the specific combination of circumstances that caused the error.

7.5 Transactions

Transactions are a standard way to avoid corrupting data structures in the presence of errors in updates [12]. The transaction processing system implements a semantics in which the entire set of updates performed during the transaction become visible to the rest of the system atomically. If the transaction fails for any reason none of its updates become visible. It is possible to dynamically check for consistency at the end of each transaction just before it commits and abort the transaction if it leaves any of the updated data in an inconsistent state.

7.6 Comparison

All of the mechanisms discussed in this section either attempt to protect the computation and its state against physical damage, or to protect the system against errors that corrupt its data structures by eliminating the effect of any computation that leaves the state inconsistent or attempts to perform an illegal action. A fundamental difference with acceptability-oriented computing in general and resilient computing in particular is that these last two approaches accept the need to incorporate at least some of the effects of erroneous computations into the system and its state. Reasons why it might make sense to do this include the need to make forward progress in the computation and eliminating recovery time.

8. CONCLUSION

Software engineering has been dominated by the aspiration to produce software that is as close to perfect as possible, with little or no provision for automated error recovery. We discuss an alternate approach that explicitly rejects the aspiration of attempting to produce perfect software as counterproductive. Software built using this alternate approach instead consists of layers of partial and potentially redundant acceptability specifications, with the layers becoming progressively simpler, more partial, and more likely to accurately capture the intended acceptability property as

they move towards the periphery of the layered structure. This approach may make it possible to build resilient systems that continue to execute productively even after they take an incorrect action or sustain damage. It may also enable organizations to prioritize their development processes to focus their efforts on the most important aspects of the system, reducing the amount of engineering resources required to build the system and enabling a broader range of individuals to contribute productively to its development.

Acknowledgements

I would like to thank Daniel Jackson, Butler Lampson, and the members of my research group, in particular Brian Demsky and Karen Zee, for many interesting and useful discussions on the subject of this paper. Also, some of the ideas in this paper were honed in collaboration with Daniel Jackson when we worked together to prepare a grant proposal; in particular, the discussion of the trade-off between functionality and reliability emerged as part of that process. Butler Lampson brought the software configuration problem to my attention in a discussion we had some time ago; I would like to thank Tim Kay for the idea that parity checks decrease reliability; Tim also developed the graphics package that discards problematic triangles. Jim Larus and Daniel Jackson brought the example of dual aircraft flight control software to my attention. Patrick Lam helped me figure out how to use the Unix `ptrace` interface from Section 2.5.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[2] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. UFO: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, 16(3):207–233, August 1998.

[3] Susan Brilliant, John Knight, and Nancy Leveson. Analysis of faults in an n-version software experiment. *IEEE Transactions on Software Engineering*, SE-16(2), February 1990.

[4] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 110–115, Schloss Elmau, Germany, May 2001.

[5] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems*, 19(3), September 1994.

[6] Stefano Ceri and Jennifer Widom. Deriving production rules for constraint maintenance. In *Proceedings of 1990 VLDB Conference*, pages 566–577, Brisbane, Queensland, Australia, August 1990.

[7] J. Darley and B. Latane. Bystander intervention in emergencies: Diffusion of responsibility. *Journal of Personality and Social Psychology*, pages 377–383, August 1968.

[8] W. Edwards Deming. *Out of the Crisis*. MIT Press, 2000.

[9] B. Demsky and M. Rinard. Role-based exploration of object-oriented programs. In *Proceedings of the 2002 International Conference on Software Engineering*, Orlando, Florida, May 2002.

[10] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Ori ented Programming Systems, Languages, and Applications (OOPSLA '03)*, Anaheim, California, November 2003.

[11] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *International Conference on Software Engineering*, pages 449–458, 2000.

[12] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] G. Haugk, F.M. Lax, R.D. Royer, and J.R. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, July-August 1985.

[14] T. Kay and J. Kajiya. Ray tracing complex scenes. *Computer Graphics (Proceedings of SIGGRAPH '86)*, 20(4):269–78, August 1986.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Longtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, Jyvaskyla, Finland, June 1997.

[16] J. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, California, June 1995.

[17] B. Latane and J. Darley. Group inhibition of bystander intervention in emergencies. *Journal of Personality and Social Psychology*, pages 215–221, October 1968.

[18] Samiha Mourad and Dorothy Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, September 1987.

[19] P. Plauger. Chocolate. *Embedded Systems Programming*, 7(3):81–84, March 1994.

[20] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[21] Martin Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1999.

[22] Susan D. Urban and Louis M.L. Delcambre. Constraint analysis: A design process for specifying operations on objects. *IEEE Transactions on Knowledge and Data Engineering*, 2(4), December 1990.

[23] James Womack, Daniel Jones, and Daniel Roos. *The Machine that Changed the World: the Story of Lean Production*. Harper Collins, 1991.