Building Resilient Systems Using Acceptability-Oriented Computing

Martin Rinard
MIT Laboratory for Computer Science
Cambridge, MA 021139

***Abstract*— With the aid of several examples, we outline an approach that developers can use to help ensure that the program executes in an acceptable way. Developers using this approach use a set of acceptability properties to define an acceptability envelope for the program. They then develop a set of acceptability monitoring and enforcement mechanisms that detect impending acceptability violations and respond by taking action to ensure that the program remains within its acceptability envelope. Potential benefits of this approach include the ability to build more adaptive software systems that resiliently recover from errors and the ability to more productively combine a variety of software components to build an acceptable system.**

I. Introduction

With the aid of several examples, we present a new perspective on building software systems that we call acceptability-oriented computing. A developer practicing acceptability-oriented computing first identifies simple properties that the program must satisfy to be acceptable. Together, these properties define an *acceptability envelope*, or a region of the potential executions that the computation must remain within to be acceptable. The developer then augments the program with components that detect impending acceptability violations, then react to those violations by taking actions to ensure that the program stays within its acceptability envelope.

There are several mechanisms that a developer can use to enforce the acceptability properties. Black box mechanisms monitor and potentially change the inputs and the outputs, but do not interfere with the execution of the program itself. Gray box mechanisms leave the code of the program unchanged but may read and write the data structures of the program or intercept procedure and system calls. White box mechanisms augment the program with code and new data that monitors and potentially affects its execution.

The goal of acceptability-oriented computing is to build more resilient computing systems that can tolerate, adapt to, and recover from errors. Instead of requiring developers to build perfect software, acceptability-oriented computing may help developers to build more reliable systems out of partially faulty components.

Because acceptability-oriented computing allows programs to continue executing even after an error, it may be one way to build long-lived systems that sustain damage, but continue to execute and satisfy the needs of their clients and users.

II. Example

We next present a simple example. The goal of this example is to illustrate how to apply the basic concepts of acceptability-oriented computing to a simple map program. This program maps names to numbers. It accepts as input a sequence of commands on the standard input stream and produces its output as a sequence of numbers on the standard output stream. 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 core procedures that implement each command in the program. 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 program 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`.

A. Acceptability Properties

To practice acceptability-oriented computing, the developer must first identify the acceptability properties that are important in the context in which the program will be used. Let us assume that we identify the following two acceptability properties:

- **Output Sanity:** The program must return either zero or a number between the minimum and maximum numbers `put` into the table. A program may trivially satisfy this acceptability property by always returning zero. It is expected, however, that the program will attempt to return the same number for each command that a completely correct implementation would return.

```
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);
}
```

Fig. 1.   Implementation of Map Core Procedures

- **Continued Execution:** The program must continue to execute so that as many of its mappings as possible remain accessible to its client. Presumably, the program will exist in a context in which potentially degraded but continuous execution is important, perhaps because it is used in a system that controls unstable physical phenomena and degraded execution is clearly preferable to stopping the program.

Both of these acceptability properties fall short of requiring perfect execution. Indeed, if the acceptability properties are too stringent, they may be so difficult to implement that the acceptability monitoring and enforcement mechanisms provide little additional benefit.

B. Output Monitoring and Rectification

Some inputs can cause the program in our example to produce outputs that are outside of the minimum and maximum numbers passed to the program. For example, the following input:

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

causes the program to produce 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`, puts the entry on the free list but does not remove the entry from the table.

The standard way to fix this problem is to debug the program and fix the error. But unless the developer produces perfect software, there will always be undetected errors that may make the program execute unacceptably. And changing complex systems to remove known errors is a notoriously risky activity — the developer may get the change wrong or the change may have a negative impact on other parts of the program that actually depend on the presence of the error for their own correct execution.

We instead discuss an approach that uses an external component to enforce our first acceptability property, namely, that any output must either 1) be between the minimum and maximum values inserted into the mapping, or 2) be 0. The code in Figure 2 enforces this property by creating two filters. The first filter (the `extract` procedure) processes the input to determine the next command. 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 3 presents the resulting process structure.

```
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]);
  }
}
```

Fig. 2. Black-Box Implementation of Min/Max Output Monitoring and Rectification

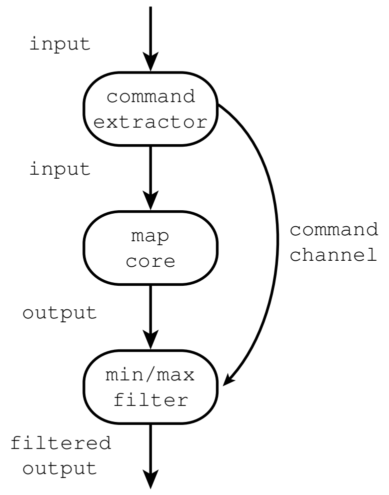

Fig. 3. Process Structure for Black-Box Implementation of Min/Max Output Monitoring and Rectification

When the output filter detects an out of bounds, it replaces the result with 0. Another acceptability enforcement component would simply replace the result with the minimum or maximum value. This approach might be preferable when any value in the table is acceptable, especially for a usage context in which the program maps names to resource numbers, and any of the resources are acceptable. An example might be a batch processing facility that uses the program to map symbolic machine names to the network addresses of the machines.

The code in Figure 2 uses a black-box approach. It processes the input and output streams, but leaves the execution of the program unchanged. Much of the code in this approach is devoted to implementing the input and output interposition mechanisms. A white-box approach typically requires less code because the relevant parts of the computation are more easily accessible from inside the address space of the computation. The drawback is that the white-box code must execute in a more complex and presumably less well understood context. It is also vulnerable to interference from the original program.

Figure 4 presents a white-box implementation of the acceptability enforcement mechanism. We have augmented the `main` procedure to maintain the minimum and maximum values put into the mapping and to coerce out of bounds values to 0. The code implements the same functionality as the black-box code in Figure 2, but because it is implemented within the input and output processing code in the `main` procedure, it is substantially smaller. 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.

A primary consideration when implementing combined input and output filters is the mechanism that the filters use to communicate. The black-box implementation in Figure 2 uses a Unix pipe to carry the extracted command information to the output filter. The white-box implementation in Figure 4 uses variables. Other potential mechanisms include network connections and shared memory segments.

```
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);
  }
}
```

Fig. 4. White-Box Implementation of Min/Max Output Monitoring and Rectification

C. Data Structure Repair

Running out of memory is an uncommon situation, and many programs fail to check for it. Our example program is no exception. The `alloc` procedure in Figure 1 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 the client allocates too many entries from the table and the free list is empty, the program simply continues, using the invalid index. The result is that the program may crash, violating our second acceptability property of continued execution.

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.

It is possible to view both the presence of circularities in the data structure and the inappropriate response to an empty free list as violations of an internal acceptability property that requires the data structures to be consistent. With this perspective, the appropriate response is to augment the program with a component that detects and eliminates any inconsistencies. We identify two consistency properties: 1) 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), and 2) the free list is not empty.

Figure 5 presents a procedure, `repair`, that detects and repairs any violations of these two data structure consistency properties. 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 `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. We anticipate that the data structure repair algorithm will be invoked before each call to `put`, `get`, or `rem`. This invocation will ensure that these procedures can rely on the data structures to be consistent when they execute.

1) Specification-Driven Repair: We have presented a manual implementation of the data structure repair algorithm. We have also developed an approach that accepts a specification of data structure repair properties, then automatically produces a repair algorithm. The advantages of this approach is that it avoids coding difficulties (the need to always check for illegal references, out of bounds array accesses, and cyclic structures) that can complicate the manual development of code that can repair arbitrarily corrupted data structures.

2) Obscured Errors: There is an issue associated with techniques (such as data structure repair) that help the system recover from errors. Specifically, these techniques hide errors and reduce the need to address the errors. In the best case, this property helps the system to execute more successfully and acceptably. This property is especially useful when there is no prospect of external intervention, and potentially quite useful even when there is an organization responsible for the system's health. But it may also result in a situation in which the system is mostly broken, its functionality is degraded, but no repairs are performed. In general, this situation is not amenable to a technical solution. It instead requires discipline on the part of those responsible for the maintenance of the system. To help the maintainers understand what is going on, our data structure repair algorithm logs each repair action to `stderr`. The resulting log may make it easier to become aware of and investigate any errors.

3) Key Properties: 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

```
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();
}
```

Fig. 5. Data Structure Repair Implementation

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. The fact that the acceptability property is partial increases the likelihood that the developer will be able to successfully implement it.
- **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 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.

D. 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. This basic problem is the source of buffer overruns, a common and notorious security vulnerability. The enormous incentive to eliminate buffer overruns combined with their continued presence in many software systems bears witness to the difficulty of eradicating these errors using standard means.

An acceptability-oriented approach might instead interpose a filter on the input. This filter would monitor the input stream to detect and eliminate overly long inputs. Figure 6 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 1. 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.

Note that this mechanism keeps the program within its acceptability envelope by filtering out unacceptable inputs — in effect, protecting the program from a hostile environment.

In general, acceptability monitoring and enforcement techniques may involve any combination of the inputs, outputs, state, timing, and behavior of the computation. Some may relate the state and the outputs; others may relate the inputs and the timing of responses from the system. There is no limit to the

```
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();
  }
}
```

Fig. 6. Token Length Filter Implementation

III. Opportunistic and Systematic Applications

We anticipate that developers will use acceptability-oriented computing in two distinct ways. Systematic applications will integrate an acceptability-oriented perspective throughout the development process, from the requirements analysis phase through implementation and maintenance. Acceptability properties that do not involve aspects of the the internal implementation will be identified during requirements analysis. Acceptability properties that involve aspects of the internal implementation will be identified during the design phase. In both cases, the acceptability property monitoring and enforcement activities will be part of the mainstream development activities.

Opportunistic acceptability-oriented computing, on the other hand, enables developers to compensate for errors as they arise. Instead of embarking on the complex process of localizing and eliminating the error, the developer will instead simply develop mechanisms that detect the manifestation of the error and adjust the state or behavior to compensate for its presence.

Acceptability-oriented computing has a good incremental adoption path. Developers can deploy it gradually within limited parts of the system and to address a small set of acceptability properties, then incrementally grow the deployment to include more parts and properties of the system. There is no need to change all at once to a new development environment, methodology, or language.

IV. 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 augments the program with layers of partial and potentially redundant acceptability monitoring and enforcement components. This approach helps the system adapt to recover from the inevitable errors that occur in every large software system. It may also make it possible to build resilient systems that continue to execute productively even after they take an incorrect action or sustain damage. Finally, it may help 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 Tim Kay, 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.

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.