Adaptation Space: Surviving Non-Maskable Failures

Crispin Cowan, Lois Delcambre, Anne-Francoise Le Meur, Ling Liu,
David Maier, Dylan McNamee, Michael Miller, Calton Pu,
Perry Wagle, and Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
(crispin@cse.ogi.edu)
http://www.cse.ogi.edu/DISC/projects/heterodyne

Abstract

Some failures cannot be masked by redundancies, because an unanticipated situation occurred, because fault-tolerance measures were not adequate, or because there was a security breach (which is not amenable to replication). Applications that wish to continue to offer *some* service despite non-maskable failure must *adapt* to the loss of resources. When numerous combinations of non-maskable failure modes are considered, the set of possible adaptations becomes complex. This paper presents *adaptation spaces*, a formalism for navigating among combinations of adaptations.

An adaptation space describes a collection of possible adaptations of a software component or system, and provides a uniform way of viewing a group of alternative software adaptations. Adaptation spaces describe the different means for monitoring the conditions that different adaptations depend on, and the particular configurations through which an adaptive application navigate. Our goal is to use adaptation spaces to provide survivable services to applications despite non-maskable failures such as malicious attacks. We present the basic concepts concerning adaptation spaces, with examples. We then present a formal model for reasoning about and selecting alternative adaptations, allowing developers of survivable application to automate their system's adaptive behavior.

1 Introduction

Some failures cannot be masked by redundancies. Non-maskable failures can occur because more failures occurred than anticipated; a trivial example is the case where *no* redundant resources were provided. Non-maskable failures can also occur due to a design or implementation failure, including and especially failures that result in security breaches[13]. Design and implementation failures are not amenable to replication, because replicas faithfully reproduce the failure [23].

Applications that wish to continue to offer *some* service despite non-maskable failure must *adapt* to the loss of resources. Numerous projects have developed adaptive capabilities for various conditions [2, 4, 5, 10, 12, 14, 16, 20, 21, 24, 28, 29]. However, when one wishes to *compose* these adaptive techniques, one encounters a complex space of alternatives, especially when some of the adaptations affect the same resources, producing conflicting adaptive behavior.

This paper presents *adaptation spaces*, a framework for navigating among combinations of adaptations. It is theoretical work with immediate practical applicability. Adaptation spaces provide application designers with a framework in which to reason about various contingencies, including non-maskable failure scenarios. Using such a theoretical framework can enhance confidence that all contingencies have been considered, and an understanding of what will happen to the system in each case.

Section 2 describes the concepts of Adaptation Spaces, and how they can be used to make software adapt to loss of resources, and then thrive when resources are returned. Section 3 provides a formal definition of adaptation space. Section 4 describes our experimental uses of adaptation spaces, and Section 5 discusses the implications. Section 6 presents our conclusions.

2 Adaptation Spaces and Cases

An *adaptation space* is a collection of alternative configurations, called *adaptation cases*, for a software component. We use the term "configuration" loosely. The adaptation cases in an adaptation space might represent completely independent pieces of code that accomplish similar functions, variants on a single piece of code, different selections of configuration parameters for the same piece of code, or simply different behaviors a program exhibits under different conditions. Each adaptation case has a *use condition*: a predicate that must be true in order for that case to be usable. The use condition is expressed in terms of items in the context where the adaptation space runs. The context can include items internal to a component—such as the value of a variable—and external to the component—such as the state of another component. Each adaptation case also has a set of *provided properties*: conditions that will hold if that adaptation case is selected. Properties provided by examples in this paper specify enabled functionality, quality-of-service guarantees, security levels, resource use and accuracy of answers.

In addition to information about its cases, the implementation of an adaptation space must provide a mechanism to ensure that use conditions hold for the current case (and a means to switch to another case if they don't) and a policy for selecting among cases when more than one is enabled according to their use conditions.

An adaptation space may cover only a subset of the conceivable configurations of its components as adaptation cases. The adaptation space represents design decisions about which of the possibilities are of interest in a particular situation. Some possible cases may not confer any benefit, such as an alternative query plan that consumes more resources while providing a less accurate answer, and hence are not worth including. Other cases may be judged to occur so rarely as to not be worth the added complexity of including them, or it might be that the conditions under which a case holds are so transitory that the cost of switching to and from the case outweighs any benefits from being in the case. Finally, some cases provide properties that are so weak that the case is judged unacceptable by the designer.

2.1 Specialization Classes

Adaptation spaces are a generalization of *specialization classes* [11, 32], which in turn generalize on *predicate classes* [6]. We begin by describing an example using specialization classes, and then expand the concepts to include the other capabilities of adaptation spaces. While specialization classes could be employed in a non-object-oriented setting, they fit the object paradigm nicely, so we will explain them in those terms.

A *specialization plan* is collection of specialization classes all implementing the same type. By type here we mean an interface composed of operation signatures [3, 17]. Each specialization class is described by giving the a specialization context, which is a predicate over the program state that must hold for the specialization class to function correctly. Some example predicates are:

```
instance_variable_x == value
instance_variable_y == instance_variable_z
```

Specialization class predicates are conjunctions of these expressions. Each specialization class indicates which of its methods should be specialized relative to its specialization context, exploiting the predicate to make the specialized methods more efficient than in the general case. Often specialization using predicates can be automated using partial evaluation [9, 8, 7].

For example, consider the **`read()`** operating system call. The general case handles numerous conditions, such as various kinds of files (including sockets and NFS file systems), concurrent writers, etc. We can specialize the implementation of **`read()`** to exploit various predicates [22], such as:

`refcount == 1`: Exclusive access, only one process has the file open, so there cannot be any concurrent writers.

`fs_type == local`: The file resides on a local file system, and therefore is not an NFS-mounted file, socket, FIFO, etc.

`fs_type == NFS`: The file resides on an NFS-mounted file system, and therefore is not a local file, socket, FIFO, etc.

A specialization plan for the **`read()`** system call combines these predicates into a set of specialization classes that the specialization designer deems useful. We can draw a specialization plan as a hierarchy of specialization classes, with the general case at the bottom, as shown in Figure 1. Note that if the condition of a specialization class is true, the conditions of all classes below it in the implication hierarchy are also true.

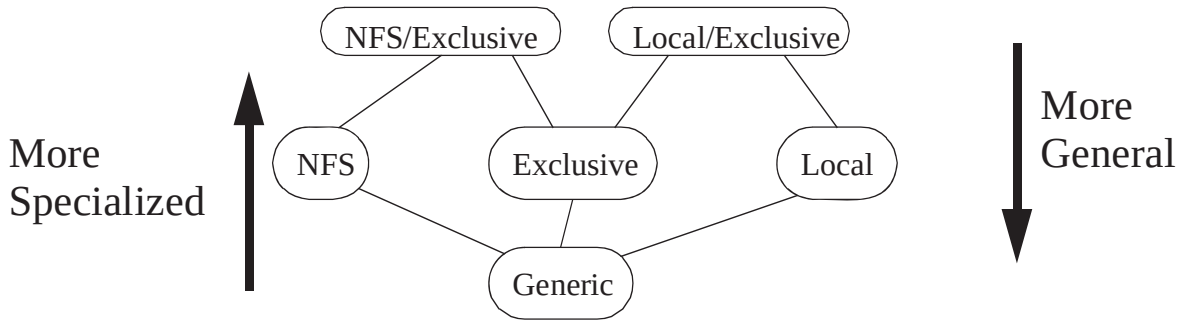

Figure 1 `read()` Specialization Plan

Each specialization class is responsible for *guarding* its context condition, and selecting an alternative class to switch to if the predicate condition becomes false. The switching between specialization classes is done independently, on an object-by-object basis, and a single conceptual object might be implemented by different specialization classes over its lifetime.

2.2 Adaptation Space

Adaptation spaces generalize specialization classes in several regards. These generalizations facilitate adaptations to provide continued operation in a partially degraded mode when non-maskable failures occur, enhancing the survivability of the application [25].

1. Code for adaptation cases can result from means other than code specialization. The cases in an adaptation space might represent settings on some configuration parameters on a routine, or they could represent completely separate implementations of a component.
2. The property of interest might be something other than execution speed. The provided property for a specialization class is assumed to be more efficient execution. Adaptation cases may provide properties concerning resource usage, quality of service, or robustness. An adaptation case may also provide a combination of properties of different types, representing trade-offs such as quality of service versus resource usage, or convenience versus security of communication.
3. Specialization plans implicitly select the most specialized case whose condition is true to maximize efficiency. To provide more than one kind of property, adaptation spaces allow more sophisticated selection policies that reflect preferences for provided properties, restrictions on movement between adaptation cases, and costs of moving to a particular adaptation case.
4. An adaptation space may use other mechanisms for monitoring the validity of the use conditions. In specialization classes, use conditions are predicate expressions using state variables from the system being specialized. An adaptation space may use other mechanisms to compute dynamic values like “average available bandwidth,” using tools such as SWiFT [15].
5. The implementations of adaptation spaces may use a wider variety of mechanisms for switching between adaptation cases. It might be some form of dynamic replugging of code used for specialization of a monolithic kernel [10], or it might be adjustment of configuration parameters, replanning a query, externalizing a program’s state then reinitializing a different program from that state, turning on a software feedback loop [5], and so forth. It might even be that no explicit change is needed when changing cases. The adaptation space may simply document the change in behavior depending on context of a single implementation, for purposes of analyzing its interaction with other parts of an application.

2.3 Sample Adaptation Spaces of Interest

Here we describe some existing technologies, both our own and others’, using adaptation spaces. We also consider new kinds of adaptations that might enhance application survivability.

1. **Quality of Service**: Much of the current work in distributed multimedia systems looks at ways to intelligently manage quality of service, as seen at the application interface [5, 12, 28, 34]. Adaptation cases can represent trade-offs between quality of service and resource consumption, such as video frame rate versus available bandwidth, or trade-offs between different quality dimensions, such as frame rate versus spatial resolution. The resource-quality trade-off is partic-

ularly interesting as a survivability adaptation, because it provides a way to reduce resource needs when resources are lost or need to be freed for a more important capability.

2. **Quality Objects:** The Quality Object (QuO) framework developed by BBN [19, 34] is a means of expressing adaptivity between objects, currently expressed via extensions to CORBA. A QuO quality contract is described by a client using a Contract Description Language (CDL), which specifies its adaptation preferences within individual negotiated regions. When available resources or client behavior exits a negotiated region, the system invokes a specified client callback to negotiate a new region. The adaptation space of a QuO system is the sum total of the quality contracts between all client/server pairs. Adaptation cases are the individual negotiated regions, and the use cases are the contract transition rules in the client callbacks.
3. **Hardening of Components:** A software component can be modified or wrapped [31, 1, 33] in order to check for out-of-range behavior, such as induced in an attack [13]. Such changes will generally be at the of expense efficiency, so they are candidates for adaptation spaces to manage these trade-offs. An example is StackGuard [14], which can insert varying degrees of self-checking for evidence of "stack smashing" attacks. More checking induces more overhead, so different efficiency-safety trade-offs can be cases in an adaptation space.
4. **Alternative Mechanism:** Mobile applications operate in a dynamically changing environment. An adaptation space can specify cases that provide the same or similar service using different underlying mechanisms. For example, a mobile application might need network connectivity to a remote resource, but the machine hosting the application might sometimes have a wired or wireless network connection, and have to rely on a modem link at other times. Adaptation cases can represent the various connection mechanisms: Ethernet, Wavelan, or PPP over a modem.
5. **Information Quality:** Applications that access information from remote, autonomous information sources (including all information on the World-Wide Web) will have to be prepared to make trade-offs not normally required in accessing a dedicated, local database. Information sources may come and go; servers can go off-line due to a security breach; the response time of a source can vary depending on demand at its server and congestion in the network leading to it; the information at a source can go out of date; and so forth. These faults are non-maskable for the application, which might need to back off from a complete or absolutely correct answer in order to get that answer in a timely manner, or to get an answer at all. Consider a SQL-style query with the basic form **`select`** `<data elements>` **`from`** `<sources>` **`where`** `<conditions>`. Adaptation cases for such a query can reduce the set of data elements required, change or eliminate sources, or relax the conditions.

3 Formal Definition of Adaptation Space

We formally define an adaptation space as a set of adaptation cases, partially ordered by the relation "more specialized than." Case *a* is more specialized than case *b* if the use condition in case *a* is logically implied by the use condition in case *b*, which we write $a \supseteq b$, and say that "*a specializes b*". Conversely, case *b* is more general than case *a* if the use condition of case *b* is a subset of the use condition in case *a*, which we write $b \subseteq a$, and say that "*b generalizes a*".

Let S be the set of all individual predicates in the use conditions we are considering. A complete adaptation space can be computed by taking the power set $\mathbf{r}(S)$ of all of the individual predicates being considered, including a separate case for each value of a predicate of the form "**`foo ==`**

..." If some of the predicates are equalities for continuous values (i.e. **foo** is an integer or a float) then this set is infinite. Being a power set, it is a lattice [30].

The "bottom" of the lattice is the case where *none* of the predicates are known to hold, and "top" is the case where *all* of the predicates hold. An implementation that can function in the bottom case is trivial, but likely uninteresting: the null function. An implementation that functions in the top case may not exist. However, "top" can be abstractly modelled as an "oracle" that always returns the right answer immediately.

The meet operation on two cases *a* and *b* in the lattice is the intersection of their use conditions, which we write $a \cap b$. The join operation is similarly the union of their use conditions, which we write $a \cup b$. Conceptually, $c = a \cap b$ means that *c generalizes a and b*, because the use conditions for *c* are satisfied by $a \cap b$. Similarly, $d = a \cup b$ means that *d specializes a and b*.

3.1 An Example: Query Relaxation

Consider an application where we would like to contact all faculty who have taught classes in the past calendar year. Let Query 1, the SQL query to find all faculty names and e-mail addresses be:

```
SELECT    F.name, F.e-mail
FROM      Faculty F, Course-Schedule C
WHERE     F.ssn = C.f-num AND
          C.year = 1997;
```

The two tables of interest, **Faculty** and **Course-Schedule**, may be stored in different locations, and thus each may become unavailable. To make our query survive the loss of these tables, we consider two adaptations using *query relaxation* [18], which *generalize* the query to require fewer (or different) tables. If only the **Faculty** table is available (without the **Course-Schedule** table) then we get a superset of the *involved* faculty and can use that to dispatch our message. Query 2 generalizes Query 1, producing a less precise answer with some false positives:

```
SELECT    name, e-mail
FROM      Faculty;
```

If we lose access to the **Faculty** table, then we consider using a different table (**Phonebook**) which includes both faculty and staff. Query 3 generalizes Queries 1 and 2, giving us a larger superset of the involved faculty that includes staff:

```
SELECT    name, e-mail
FROM      Phonebook;
```

To describe these adaptations as an adaptation space, we first consider the use conditions for the various queries: when is it appropriate to use Query 1? Query 2? or Query 3? The use condition for these three queries hinges on the availability of tables. Query 1 requires tables **Faculty** and **Course-Schedule**; Query 2 requires **Faculty**; and query 3 requires the **Phonebook** table. The adaptation space resulting from the power set of these three bits of information is shown in Figure 2. The use condition for each case is shown inside each node in the lattice, where F represents the **Faculty** table, C represents the **Course-Schedule** and P represents the **Phonebook** table.

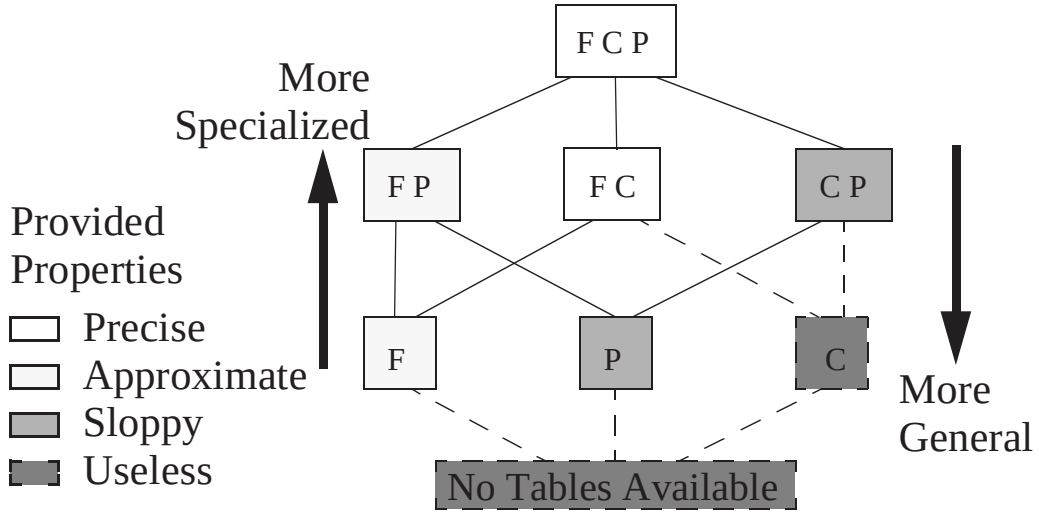

Figure 2 Adaptation Space for Query Relaxation Example

The adaptation cases have also been colored according to the properties provided. In this example, the provided properties describe the precision with which the query captures the faculty involved. All the cases capture all the faculty involved except C, which captures none of them. However, the "Approximate" cases also capture *uninvolved* faculty, and the "Sloppy" cases capture staff as well.

Note that while the adaptation space contains all 8 nodes from the power set of conditions, we do not implement 8 queries. In particular, if no relevant tables are available, then only the null query still operates, and that does not provide us with any useful results. We indicate this in Figure 2 by dashed lines around cases that are not implemented. We would also eliminate any nodes where the use conditions conflict, e.g. "`foo < 2`" and "`foo >3`". This begs the question of which cases *are* implemented, and when to switch between them. The relationship between the adaptation space and its implementation is specified by the *transition graph*, described in Section 3.2.

3.2 The Transition Graph

To map the adaptation space to a set of implementations, we first re-examine the adaptation space. Often multiple cases can be accommodated by a single implementation, needing only a parameter change, or even no change at all, as shown in Figure 3. Each query accommodates two adaptation cases. No changes at all are required to span the two cases, because the more specialized case includes a table that the query is not using. Each group of cases in a single implementation is marked with the use conditions enabling that implementation, which is the meet of all the cases in the group. The mapping in Figure 3 also reveals that case C is not used; if we *only* have the **`Course-Schedule`**, we cannot generate any faculty e-mail addresses, and thus this case is not useful.

From the marked-up adaptation space in Figure 3, we can compute the *transition graph* in Figure 4. The nodes in the transition graph are labelled with the software used to implement this space, and have been annotated with the use conditions required for each implementation. The edges represent transitions from one implementation to another, and have been labeled with the *changes* in use conditions that mandate transitioning to another node, as well as the cost of making the transition. In this case, the cost of transitioning to and from Query 3 is higher because Query 3 uses a completely different table, necessitating re-optimizing the SQL query. The cost values are arbi-

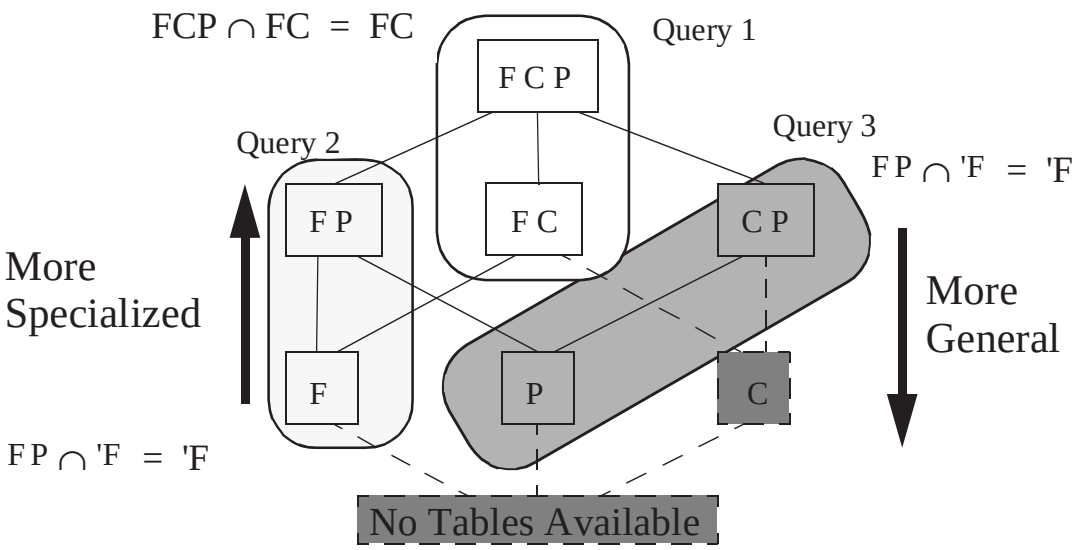

Figure 3 Implementations and the Adaptation Space

1. Combine cases in the adaptation space that can be satisfied by a single implementation. Mark each combination with the ∩ of each case, i.e. the intersection of use conditions.
2. Place the implemented nodes on the transition graph.
3. For each edge in the adaptation space connecting separate implementations, add a *directed* edge to the transition graph, labelling the edge with the changes in use conditions between the two cases in the adaptation space. Transitions "upward" in the adaptation space represent *thriving*: they are optional (representing an opportunistic improvement in service), and are labelled only with the use conditions of the destination case. Transitions "downward" are *mandatory* (representing a survivability adaptation to the loss of some resource) and are labelled "destination & !source". For instance, in Figure 3 the edge from FP to FCP is upward, and is labelled "C", while the edge from FP to P is downward, and is labelled "P & !F".
4. Combine and simplify the edges connecting implementations in the transition graph. For instance, step 3 added two edges from Query 1 to Query 2 to the transition graph, both of which are labelled "F&!C", thus these two edges from Query 1 to Query 2 can be combined.

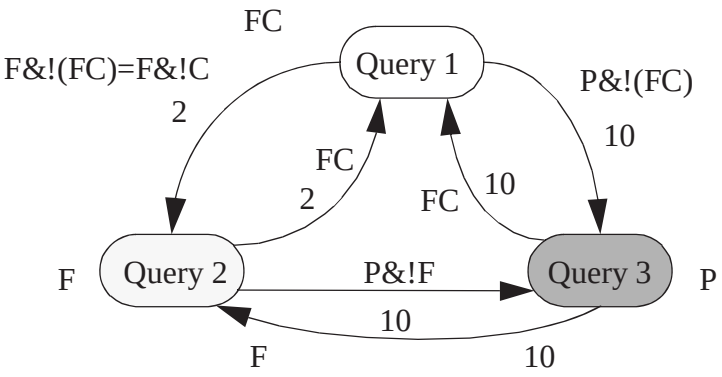

Figure 4 Transition Graph: When to Change Implementations, and What it Will Cost

5. Label the transition edges with the relative expected cost of each transition. The cost results from computations necessary to transition from one implementation to another, such as refreshing caches, initializing state, etc.

From this exercise, we have identified a minimal set of implementations that provide a selection of "acceptable" properties. We have a complete understanding of all the combinations of failures that might occur, we know which combinations of failures will still allow us to produce results (of varying quality) and we know which combinations will result in total failure of the application.

3.3 Composing Adaptation Spaces

Here we describe the composition of adaptation spaces, to account for the fact that a component may need to be adapted in more than one way, accommodating more than one kind of failure. One *can* just consider all possible failure modes in a single adaptation space, but that requires one to contemplate *all* kinds of failure, and to understand their interactions. Instead, we provide for the *composition* of adaptation spaces. Composing adaptation spaces proceeds via the following steps:

1. Unify name spaces of the two adaptation spaces
2. Compose the spaces
3. Simplify

Unifying the name spaces is necessary if the adaptation space designers used different granularity to identify the terms in the use conditions. For instance, one designer might have said "# servers >= 3", while another said "servers a and b are available".

Once the name spaces are unified, we inspect the use conditions of the two adaptation spaces for overlap. If there is no overlap at all, i.e. none of the names from one space appear in any of the use conditions in the other space, then the two adaptations are said to be *orthogonal*. The adaptations do not affect each other, and thus can be trivially composed.

If there is overlap, then we compose the adaptation spaces by computing the direct product of the two lattices representing the respective complete adaptation spaces, say S and T. Since S and T are both lattices, the direct product is a lattice [30]. Once the spaces are composed, they can be pruned of impossible and uninteresting cases, grouped into implementations, and transformed into transition graphs as described in Section 3.1 and Section 3.2.

4 Experimental Implementation of Adaptation Spaces

This section illustrates our use of adaptation spaces in practice. We describe two efforts to build applications that are adaptive in more than one dimension. Section 4.1 describes adaptations to enhance the survivability of a distributed information system, and Section 4.2 describes adaptations to manage audio quality in the presence of variable bandwidth availability.

4.1 Adaptive Information Survivability

We have developed a simulation of a DoD distributed mission planning system using a client server architecture. This system consists of three servers and any number of clients. Each server provides a particular type of information: targets, available resources (aircraft), and planned missions,

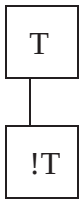

Figure 5 Target Server Failure Adaptation Space

respectively. Clients provide user interfaces to create new missions. The “normal” (optimal) case has all clients fully connected with all of the servers and all of the other clients. This case will end up being the “top” of our adaptation lattice, as it has the strongest possible set of assumptions: all servers and clients are available. We build this adaptation space by considering two simple adaptation spaces, and then composing them. One adaptation space considers the possible failure of the servers, and the other considers possible partitioning of the network of clients.

4.1.1 Server Failure Adaptation Space

Access to a server can be lost due to network or server failure. Clients can adapt by sharing data instead of depending on the server to provide the data. The non-accessed server will not be updated with any new data from newly created missions, but the clients can continue to operate, enhancing the system’s survivability. Figure 5 shows the (trivial) adaptation space for a single client adapting to the failure of the target (T) server. The component of the client that provides a list of targets to the user adapts to either get the data from the target server, or from the other clients .

4.1.2 Network Partition Adaptation Space

Connectivity between clients can be lost due to a network partition. A possible adaptation is to partition the targets and available resources a priori among the clients. When a network partition occurs, each client restricts its activities to the union of the targets and resources assigned to the cli-

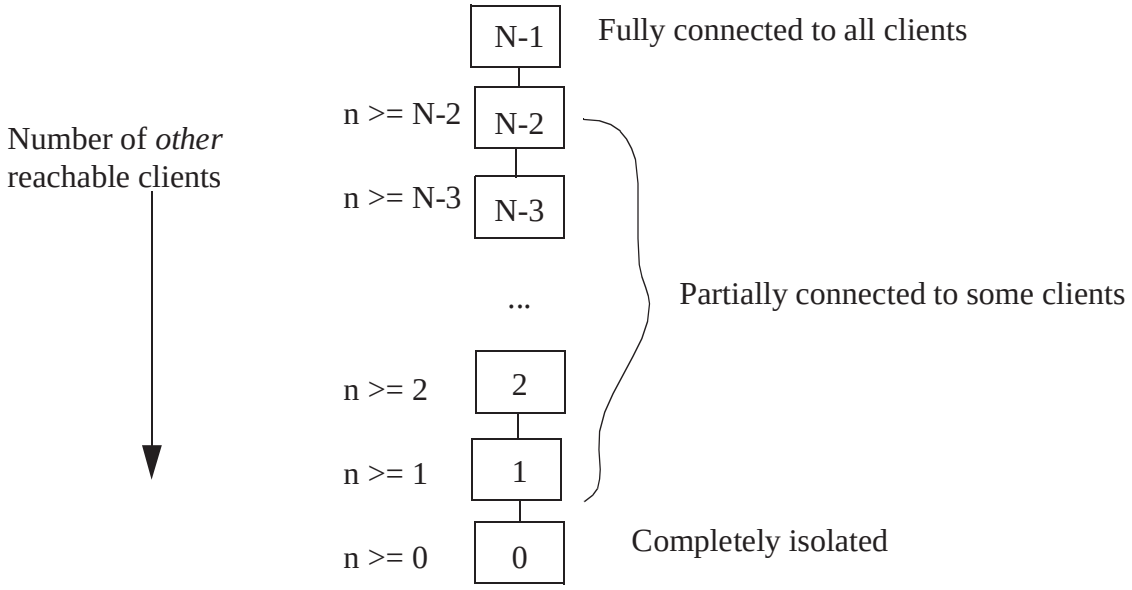

Figure 6 Network Partition Adaptation Space for N Clients

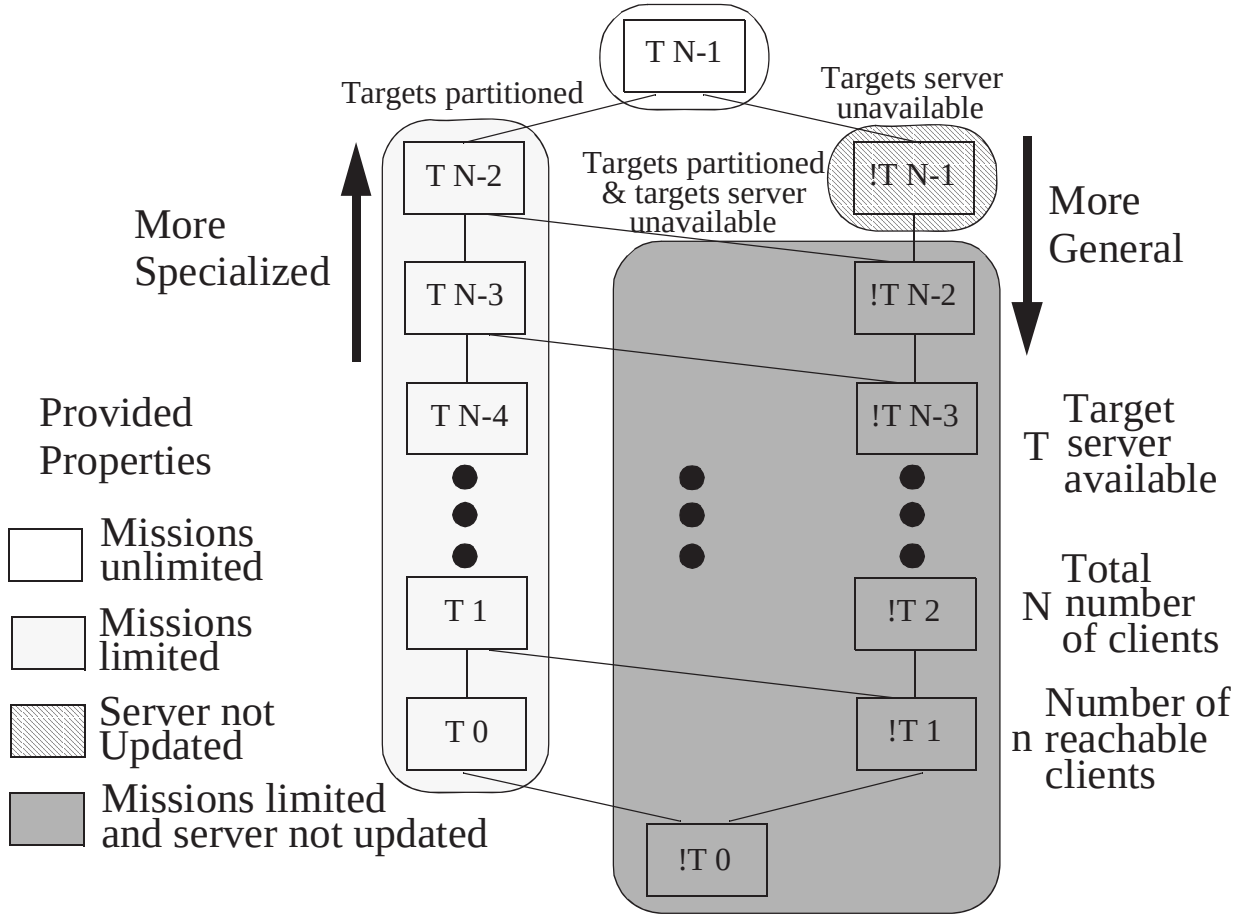

Figure 7 Composition of Target Server Unavailable and Partitioning Spaces

ents that are still accessible in the reachable network partition. Clients in the other network partition(s) are presumed to continue operating on their respective partitions of targets and resources.

Clients in partitions other than the server poll other clients to find the scope of their network partition, and compute the set of targets and resources that they can still use. Figure 6 shows the adaptation space for a single client, in terms of the number of *other* clients that it can still reach.

4.1.3 Composing Server Failure and Network Partition Adaptation Spaces

Consider composing the adaptation spaces shown in Figure 5 and Figure 6 with respect to a client's target handling component. These adaptations are not orthogonal, because they both affect the set of targets presented to the user. Thus we compose the two spaces by computing the direct product, as shown in Figure 7. This adaptation space consists of two copies of the space in Figure 6, one with and one without the target server, denoted T and !T, respectively. Sets of cases have been grouped to indicate the different implementations of the client target handler that will cover those cases, as described in Section 3.2. Figure 8 shows the client target handler's transition graph.

4.2 Adaptive Audio Quality

Streaming media presentations using shared resources such as the Internet are greatly enhanced by adapting to available resources. We are developing quality of service specifications [28] that help

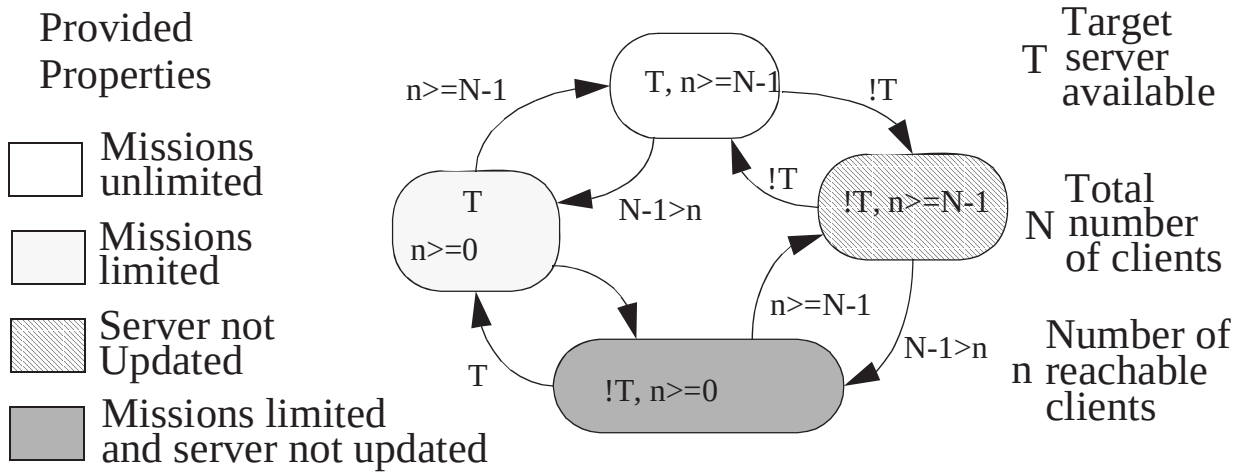

Figure 8 Transition Graph for Composed Spaces

the system navigate the adaptation space provided by sets of media filters to maximize perceived user quality. Our model [26, 27] defines quality in terms of deviation from a perfect presentation. The quality of service specification allows the user or application to relatively weight various provided properties through the adaptation space.

Figure 9 shows three simple adaptation spaces, representing three different dimensions of audio quality: sampling frequency, bit resolution of the sample, and whether it is stereo or mono. The use condition for each case is that there be sufficient data bandwidth to support that case. However, these cases must be combined to discover the bandwidth requirement, because the needed bandwidth is a function of all three parameters: needed bandwidth = frequency * bits * # of channels. The use conditions in the three separate adaptation spaces are only relative, i.e. the bandwidth required for 44 KHz is twice as great as for 22 KHz, but we don't know the absolute bandwidth needed without knowing the frequency, resolution, and # of channels.

Unifying the name space of the three spaces is trivial, because they all have a single use condition: sufficient bandwidth, in bits per second. We compute the product of the three lattices, and then label each case with the absolute bandwidth requirement in bits per second, as shown in Figure 10.

Figure 10 shows a different kind of adaptivity than in Section 4.1. This lattice has several cases with identical use conditions, i.e. "22KHz/16-bit/stereo", "44KHz/8-bit/stereo" and "44KHz/16-bit/mono" all have a use condition of at least 705 Kbits per second of available data bandwidth. Yet we do *not* seek to group them together in a single implementation, because they provide *different* properties.

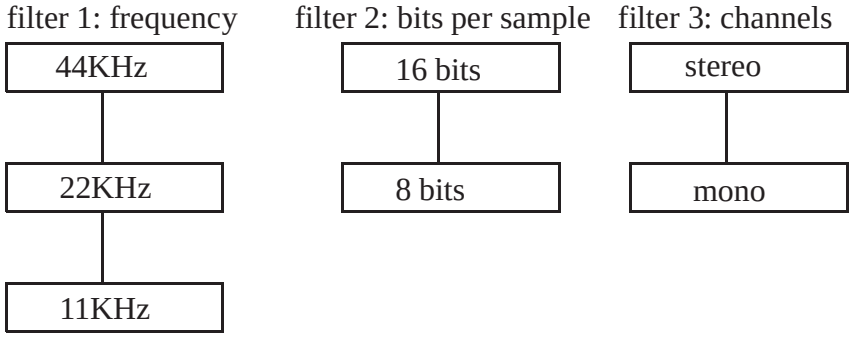

Figure 9 Independent Audio Quality Adaptation Spaces

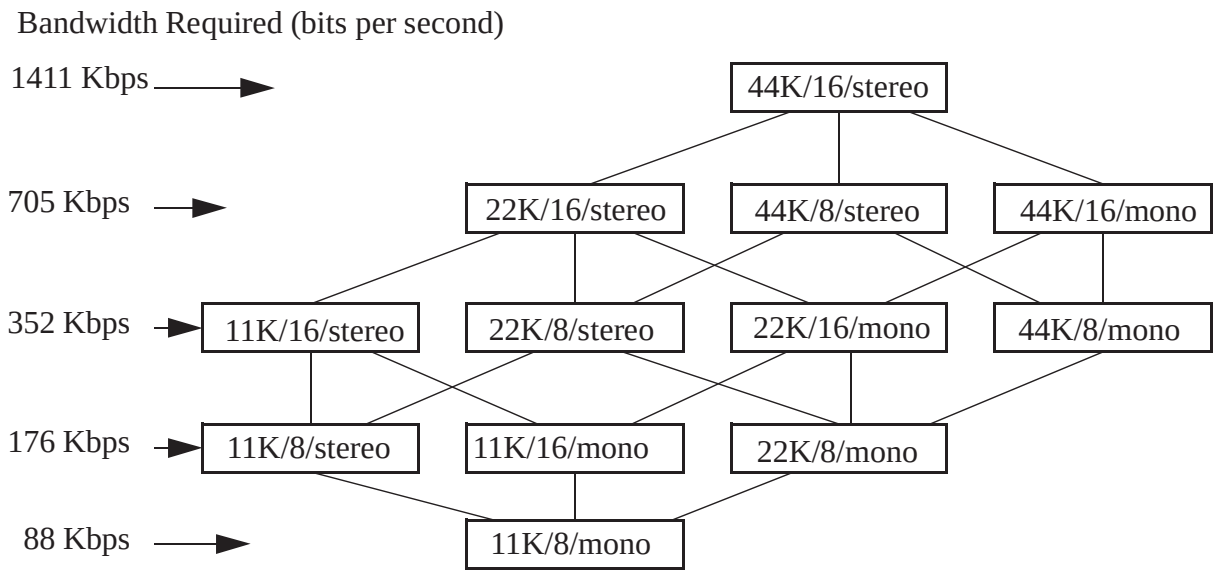

Figure 10 Adaptation space for audio quality adaptation.

This example shows the need for sophisticated adaptation case navigation: *user preference*, encoded as a quality of service specification, selects which case to use under each bandwidth condition.

5 Discussion

Adaptation spaces provide a systematic way to consider possible adaptations for a software component. The initial lattice of the adaptation space is generated exhaustively from the use conditions of interest. The complexity of this lattice illustrates the number of issues to be considered to make an application survive non-maskable failures, illustrating the need for a systematic approach.

Within the lattice, we are guaranteed substitutability of all cases less-specialized-than the current case based on the implication of a use condition of a more-specialized case to a less-specialized case. Similarly, we are guaranteed that all cases more-specialized than the current case will provide more desirable properties. Thus adaptation spaces define the essence of surviving, by adapting to a less-specialized case, and thriving, by adapting (back) to a more-specialized case.

The choice of implementations to be built for an adaptation space is made by the designer of the survivable application. In the query example involving faculty in Section 3.1, the judgement that Query 2 and Query 3 are acceptable, albeit degraded, queries is clearly application-dependent. It is through the transition graph that the designer specifies the appropriate action for each adaptation case in the initial lattice. Either the case is NOT mapped to an implementation (indicating that the case does not permit an acceptable behavior) or the case is mapped to the specific implementation.

The generation of the transitions in the transition graph is done algorithmically based on the relevant use conditions. The transition graph is ideally suited to drive the adaptive behavior of the program because the conditions of interest as well as the proper transitions are concisely and precisely represented. The exhaustive, initial lattice of the adaptation space allows the application designer to consider the various alternatives in a systematic way whereas the transition graph condenses the information needed at run-time.

When considering multiple adaptations for a software system, there are several ways to compose adaptation spaces. In the simplest case, orthogonal adaptation spaces can be used independently; the transitions graphs can each guide the adaptation of the relevant components. In more complex situations, the adaptation spaces are formally composed to arrive at appropriate adaptations for multiple considerations. In future work, we will consider other types of composition of interest including layered adaptation spaces, and functionally composed adaptation spaces.

6 Conclusions

Adaptation spaces capture the essence of adaptive behavior through orderly assembly of the adaptation cases, with the associated use conditions and provided properties. Adaptation spaces enable systematic analysis of the combinations of failures to be considered. We have used adaptation spaces to analyze and direct the behavior of several of our systems in diverse areas, including database fault-tolerance, application fault-tolerance, and adaptive quality of service. Use of this framework should both ease the difficulty of constructing applications that survive non-maskable faults, and enhance the quality of these survivable applications.

References

[1] AUSCERT. `overflow_wrapper.c` – Wrap Programs to Prevent Command Line Argument Buffer Overrun Vulnerabilities. `ftp://ftp.auscert.org.au/pub/auscert/tools/overflow_wrapper`, May 1997.

[2] Veronica Baiceanu, Crispin Cowan, Dylan McNamee, Calton Pu, and Jonathan Walpole. Multimedia Applications Require Adaptive CPU Scheduling. In *Workshop on Resource Allocation Problems in Multimedia Systems*, Washington, DC, December 1996.

[3] Andrew P. Black. Object Identity. In *Proc. International Workshop on Object-Oriented Operating Systems*, Asheville, NC, December 1993.

[4] Jeremy Casas, Ravi Konuru, Steve W. Otto, Robert M. Prouty, and Jonathan Walpole. Adaptive Load Migration Systems for PVM. In *Proceedings of Supercomputing '94*, pages 390–399, Washington, D.C., November 1994.

[5] Shanwei Cen, Calton Pu, Richard Staehli, Crispin Cowan, and Jonathan Walpole. A Distributed Real-Time MPEG Video Audio Player. In *Proceedings of the 1995 International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'95)*, pages 151–162, New Hampshire, April 1995.

[6] Craig Chambers. Predicate Classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'93)*, Kaiserstautern, Germany, July 1993.

[7] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[8] Charles Consel, Luke Hornoff, Jacque Noye, Francois Noël, and Eugen-Nicolae Volanschi. A Uniform Approach for Compile-Time and Run-Time Specialization. In *International Workshop on Partial Evaluation*, Dagstuhl Castle, Germany, February 1996. Springer-Verlag LNCS.

[9] Charles Consel and Francois Noël. A General Approach to Run-time Specialization and its Application to C. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburgh Beach, FL, January 1996.

[10] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast Concurrent Dynamic Linking for an Adaptive Operating System. In *International Conference on Configurable Distributed Systems (ICCDS'96)*, Annapolis, MD, May 1996.

[11] Crispin Cowan, Andrew Black, Charles Krasic, Calton Pu, Jonathan Walpole, Charles Consel, and Eugen-Nicolae Volanschi. Specialization Classes: An Object Framework for Specialization. In *Proceedings of the Fifth International Workshop on Object-Orientation in Operating Systems (IWOOOS '96)*, Seattle, WA, October 27-28 1996.

[12] Crispin Cowan, Shanwei Cen, Jonathan Walpole, and Calton Pu. Adaptive Methods for Distributed Video Presentation. *ACM Computing Surveys*, 27(4):580–583, December 1995. Symposium on Multimedia.

[13] Crispin Cowan, Calton Pu, and Heather Hinton. Death, Taxes, and Imperfect Software: Surviving the Inevitable. Submitted for review, April 1998.

[14] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Conference*, San Antonio, TX, January 1998.

[15] Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole. SWIFT: A Feedback Control and Dynamic Reconfiguration Toolkit. In *Proceedings of the Second USENIX NT Symposium*, Seattle, WA, August 1988. `http://www.cse.ogi.edu/DSRG/swift/`.

[16] Ajei Gopal, Nayeem Islam, Beng-Hong Lim, and Bodhi Mukherjee. Structuring Operating Systems using Adaptive Objects for Improving Performance. In *Proceedings of the Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS '95)*, pages 130–133, Lund, Sweden, August 1995.

[17] Wilf LaLonde and John Pugh. Subclassing not = subtyping not = is-a. *Journal of Object-Oriented Programming*, 3(5), January 1991.

[18] Ling Liu and Calton Pu. A Metadata Based Approach to Improving Query Responsiveness. In *Proceedings of the IEEE International Conference on Metadata*, June 1987.

[19] J.P. Loyall, R.E. Schantz, J.A. Zinky, and D.E. Bakken. Specifying and Measuring Quality of Service in Distributed Object Systems. In *Proceedings of the First Interational Symposium on Object-Oriented Real-Time distributed Computing (ISORC'98)*, Kyoto, Japan, 20-22 April 1988.

[20] Henry Massalin and Calton Pu. Fine-Grain Adaptive Scheduling Using Feedback. *Computing Systems*, 3(1):139–173, Winter 1990.

[21] Bodhisattwa Mukherjee and Karsten Schwan. Improving Performance by use of Adaptive Object: Experimentation with a Configurable Multiprocessor Thread Package. In *Second IEEE International Symposium on High-Performance Distributed Computing (HPDC-2)*, Spokane, WA, July 1993. Also available as GIT-CC-93/17, `ftp://ftp.cc.gatech.edu/pub/coc/tech_reports/1993/GIT-CC-93-`

17.ps.Z.

[22] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[23] Calton Pu, Andrew Black, Crispin Cowan, and Jonathan Walpole. A specialization toolkit to increase the diversity of operating systems. In *Proceedings of the 1996 ICMAS Workshop on Immunity-Based Systems*, Nara, Japan, December 1996.

[24] Calton Pu and Jonathan Walpole. A case for adaptive OS kernels. In *Proceedings of the 1994 OOPSLA Workshop on Flexibility in Systems Software*, Portland, Oregon, October 1994.

[25] Howie Shrobe. ARPATech '96 Information Survivability Briefing. `http://www.darpa.mil/ito/ARPATech96_Briefs/survivability/survive_brief.html`, May 1996.

[26] R. Staehli and J. Walpole. Using script-based QOS specifications for resource scheduling. In *Proceedings of the Fourth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 93–95, Lancaster, UK, November 1993.

[27] Richard Staehli, Jonathan Walpole, and David Maier. Device and Data Independence for Multimedia Presentations. *ACM Computing Surveys*, 27(4):640–642, December 1995. Symposium on Multimedia.

[28] Richard Staehli, Jonathan Walpole, and David Maier. Quality of Service Specifications for Multimedia Presentations. *Multimedia Systems*, 3(5/6):251–263, November 1995.

[29] H. Thimm and W. Klas. Delta-sets for Optimized Reactive Adaptive Playout Management in Distributed Multimedia Database Systems. In *12th IEEE International Conference on Data Engineering*, New Orleans, LA, February 1996.

[30] Jean-Paul Tremblay. *Discrete Mathematical Structures with Applications to Computer Science*, chapter Lattices and Boolean Algebra, pages 378–397. McGraw-Hill, 1975.

[31] Wietse Venema. TCP WRAPPER: Network Monitoring, Access Control, and Booby Traps. In *Proceedings of the Third Usenix UNIX Security Symposium*, pages 85–92, Baltimore, MD, September 1992. `ftp://ftp.win.tue.nl/pub/security/tcp_wrapper.ps.Z`.

[32] Eugen N. Volanschi, Charles Consel, Gilles Muller, and Crispin Cowan. Declarative Specialization of Object-Oriented Programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*, Atlanta, GA, October 1997.

[33] Joe Zbiciak. `wrapper.c` Generic Wrapper to Prevent Exploitation of `suid/sgid` Programs. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, May 19 1997. `http://cegt201.bradley.edu/ im14u2c/wrapper/`.

[34] J.A. Zinky, D.E. Bakken, and R.E. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, April 1997.