Tolerating Inconsistency*

Robert Balzer†

USC Information Sciences Institute

Abstract

Software systems, especially large ones, are rarely consistent. Modifications are constantly slowly propagating through them. Yet no principled basis exists for managing their development during these periods of inconsistency.

Similarly, exceptions inevitably arise in the data managed by practical software applications. Previous techniques required constraints to be modified to allow particular exceptions. In practice, this often led to removal of the constraints themselves.

Instead of treating inconsistency informally (i.e. outside the system) or as hard constraints, formalisms are needed for spotting such violations, treating them as problems, organizing resources to resolve them, recognizing when this has occurred, and limiting access to the inconsistent data by agents not involved in the resolutions.

We present a simple technique for creating such a formalism that allows development environments and other software systems to tolerate and manage their inconsistencies. It softens constraints without introducing special cases by treating violations as temporary exceptions which will eventually be corrected. Until then the offending (i.e. inconsistent) data is automatically marked by guards to identify it to code segments that can help resolve the inconsistency — normally by notifying and involving human agents — and to screen such inconsistent data from other code segments that are sensitive to the violation.

When the violation is eventually resolved so that the data is no longer inconsistent, the guards are automatically removed to signal that no further resolution activity is required, and to restore the accessibility of this data to code segments using the guards to screen off such data while it is inconsistent.

An additional benefit of this technique is to allow all the updates required by some constraint to occur distributed in time in any arbitrary order rather than requiring them to be synchronized with each other and occur all at once. Thus, it is particularly useful in multi-client applications with shared responsibilities and authority, such as cooperative design — such as software development itself — where such synchronization is both difficult and counter-productive to the cooperation.

We've created two fully automatic implementations for softening arbitrary first-order constraints. The first converts the original constraint into a pair of constraints that require the presence of the guards while the original constraint is violated and their absence otherwise. These new constraints "repair" violations by respectively inserting or removing the guards. The second implementation replaces the original constraint by an inference rule which derives the guard if and only if the original constraint predicate was violated.

Our contribution lies in creating a formalism in which constraint violations can be treated as temporary, automatically marked, exceptions and resolved in a time distributed and cooperative manner rather than forcing resolution (or roll-back) at the point of violation as is the case with current transaction mechanisms.

1 Background

Recent progress in programming languages, database technology, and knowledge representation systems has progressed to the point where a wide range of semantic integrity constraints can be stated and enforced. This is part of the broader thrust within Computer Science toward explicit declarative structures with formal se-

*This work was supported by DARPA under contract NCC 2-520.

†The author is deeply indebted to his colleagues in ISI's Software Sciences Division for creating the rapid prototyping and evolution technology that made this work possible and the intellectual climate that stimulated it.

mantics. That is, declarations which are not merely informal comments, but have semantic import.

This is an important direction which needs to be encouraged and supported. The constraints that exist within an application, as well as the requirements imposed by particular algorithms, must be explicated and verified (either statically or dynamically). Doing so increases system reliability, provides a richer interface specification for component reuse, and facilitates later evolution.

But explicating these constraints and assumptions leads us to a conundrum; almost all such restrictions have exceptions. Only three possibilities exist:

1. Don't state the constraint or assumption.
2. Don't enforce the constraint or assumption statement (i.e. treat it as a comment).
3. Weaken the constraint or assumption to allow the exceptions.

The first two are unacceptable because they obviate all the advantages of explicating declarations with semantic import. Yet this is how most of our software development environments and application programs operate. Because there is no way of handling violations of their integrity conditions (e.g. type safety, called functions being defined, and invocations matching function signatures in development environments), they are not stated as constraints. Rather, these integrity conditions are implicit in tools (such as static analyzers, compilers, loaders, or configuration managers in development environments) which are operationally invoked at particular points and list the "problems" they find at that time. These "problem lists" are for human consumption only. They are only used by the system, if at all, to prevent further processing until they are resolved. They are not updated as the problems are fixed, but only when the analysis tools are run again.

If we are to progress beyond this operational approach in which there are no explicit constraints but only a set of "algorithmic tests" which must be passed at explicit points before proceeding, we must formally state our constraints but somehow allow violations.

We are thus driven to weakening the restrictions. Two forms of weakening have been studied: the data to which the restriction applies; and the times when, and the clients for which, the restriction applies. The former will be dealt with here, while the latter will be addressed in the Related Work section after we've fully described our approach.

Current efforts to allow data exceptions have split into two approaches, one based on classes of exceptions and the other on individuals.

The class based approach [MBW80, ACO85, SFL81, Bor88] uses inheritance to provide subclass specializations which shadow and override those specified for the parent class (classes for multiple inheritance). Subclasses can be very broad or include only a single individual. A standard example is that "Birds fly", but "Penguins don't fly" even though they are birds, and "Tweety flys" even though he is Penguin. In this approach, the exceptions are attached to the subclasses they pertain to, and are thus distributed throughout the inheritance lattice rather than concentrated at a single spot in a comprehensive statement.

The individual based approach [BW85, Bor85] instead dynamically detects constraint violations, and when directed to do so by user written exception handlers, modifies the constraint so that the exception is not included in the constraint's domain and stores the facts identified by the handler as "exceptional" in special relations. Later access of such exceptional facts through the normal relations, rather than explicitly through these special relations, raises an access exception. The exceptional data is thus limited only to that code that is explicitly prepared (via handlers or direct access of the special relations) to deal with the inconsistencies contained in that data.

This approach was motivated by the desire to provide a structured method for gradually incorporating special case handling, and is designed to maintain the readability of an application as it deals with the inevitable exceptions that arise in real applications by segregating the complexities of processing those special cases in exception handlers. The exceptions addressed are *persistent*, i.e. the domain model is actually overly restrictive and the application must be extended (either by new code or by user intervention) to handle the special cases encountered. By explicitly identifying the exceptional data, it too can be segregated so that access is limited to code prepared to deal with the particular exceptions.

Our work is most similar to this approach, and will be described below in terms of it, but our motivation is quite different. Whereas it was intended for permanent exceptions that could be rationalized and for which new code had to be created, we are concerned with the problem of transient exceptions with relatively short lifetimes and their processing during this period of inconsistency.

In the individual approach, exceptions represent an inadequacy of the domain model and the application is extended to incorporate the handling of the new special cases. For us, exceptions represent an inadequacy in accurately capturing the *state* of the domain. That is, there is nothing wrong with our model of the domain, but rather with our knowledge of its current state.

There are two sources of such temporary exceptions: inaccurate data and unsynchronized updates. The first may arise from either clerical error or imprecise measurement (including sensor malfunction), while the second arises from partial update.

Current techniques flatly reject both types of inconsistency with the expectation that the user will simply correct or complete the update, thus sparing the system from having to deal with inconsistent data. While this is a reasonable expectation for simple constraints with a very narrow scope (such as type or range restrictions), it isn't for constraints with a broader scope involving non-trivial consistency sets (as with development environment constraints such as actuals matching formals, interface compatibility, or type safety). For such constraints — which are the main focus of this paper — users (whether human or program) may not know which piece is wrong or have the authority to change it if they do.

Rather than rejecting such inconsistencies, our approach marks the offending data and allows it to persist temporarily while an activity is mounted to resolve the inconsistency. By explicating the inconsistency, its resolution can be tracked and handled within the system. This removes some of an application's rigidity by relaxing the synchronization requirement inherent in constraints that forces the entire update to be done all at once. This is especially important in multi-client applications with shared responsibilities and authority, such as information systems or cooperative design (i.e., software development environments), where constraints deal with the integrity of the system as a whole and commonly span the boundaries of individual clients.

In particular, as is the normal case in software development, this allows each client (developer) involved in a multi-party update (system revision) to submit its individual portion (module) independently of the others, in arbitrary order. During this process the update will be partial, and hence, inconsistent, but once it is completed the inconsistency will have been resolved. In fact, as described later, the application can facilitate and control the cooperation and coordination of the multiple parties in response to the first update that initiates the inconsistency.

As with the individual approach, the application must also be manually extended to process, or be screened from, these exceptions. But for us, this is not a refinement process handling a new special case, but rather a robustness process determining how to deal with the uncertainty of the application's true state until consistency is restored. Normally, since only a very small portion of an application is dependent upon a particular constraint, and hence sensitive to its violation, the set of code to be extended or screened is also small. But it must be located and extended manually.

2 Approach

As described above, we are interested in relaxing particular constraints so that the updates needed to move to the next valid state can be entered asynchronously. Until the update is complete, the database is inconsistent. This inconsistency must be detected and localized to the data participating in the constraint violation so that the rest of the database can be processed normally.

This is accomplished by changing the constraint so that instead of prohibiting some condition Q, it introduces a new condition P and requires that Q and P occur jointly, i.e. either both hold or neither holds. The new condition is a **Pollution Marker** indicating the violation of the specific constraint. It is a n-ary relation whose parameters are the instantiated leading quantified variables of condition Q. By incorporating the instantiated values of Q's leading quantified variables, the Pollution Marker uniquely identifies the particular data that violates the original consistency criteria (i.e. condition Q).

Consider a toy example from an imaginary spreadsheet system which allows "results" to be entered by users as well as "inputs." Each formula in such a spread-sheet is a constraint which relates the value of its result to the values of its inputs. If the value of the result is changed while the inputs remain constant, then the constraint between the inputs and outputs is violated.

To temporarily tolerate such inconsistencies while the user decides which inputs, or combination of inputs, to adjust to resolve the inconsistency, the constraint is changed to require that either the original formula (say, "Formula1: $A1 + B1 = C1$") is satisfied, or a Pollution Marker (Inconsistent-Formula1) instantiated with the correct values of the inputs and results of the violated formula ($A1 + B1 = C1$) is satisfied (e.g., where the values for A1, B1, and C1 were 5, 7, and 13 respectively because $5 + 7 \neq 13$). This instantiated Pollution

Marker could be used to change the displayed colors of the input and result cells to indicate to the user that their current values are inconsistent with the formula relating them. Likewise, the removal of this Pollution Marker, when the consistency was resolved, could restore the displayed color of these cells to the original value.

As a real example drawn from an operational employee database application that keeps track of the percentage that employees work on various projects, consider the constraint that employees work the proper amount every day. Normally this is 100%, but for part timers it is some lesser amount. This constraint is defined in terms of two derived relations *Required%-Date* and *Total%-Date*. The first defines how much an employee should work on a particular date, while the second sums the employee's effort assignments across all projects on that date. The constraint states that there can't exist any date for any employee on which these two amounts differ.

Each violation of this constraint involves an employee, the date on which the employee worked the wrong amount, the amount the employee should have worked (Required-Amount), and the amount he or she did work on that date (Total-Amount). These four values are included in the Pollution Marker, *Work-Pollution*, defined for this constraint, and the constraint is changed so that this Pollution Marker occurs jointly with each violation (i.e. whenever an employee works the wrong amount on some date).

Because these Pollution Markers occur jointly with violations of the original consistency criteria, they can be used as guards (as in Dijkstra's Guarded Commands [Dij76]) to screen or conditionalize code sensitive to the inconsistency. By employing these guards, programs can be modified to avoid any inconsistencies completely or to tolerate them by adjusting their behavior. This control is fine grained because each type of Pollution Marker identifies violations of a specific constraint, and each instance of that Pollution Marker corresponds to a particular violation. Code that is insensitive to violations of a specific constraint, need not be guarded by its Pollution Marker, and will process the database, including the inconsistencies, normally.

The need to tolerate inconsistencies in the example constraint arose from the fact that only project leaders could modify the amount someone works on their project and that only the Business Office could modify the amount that someone is supposed to work. Using conventional serializable transactions, there was no way for any of these parties to move someone from one project to another or to change the amount they should work without violating the constraint.

The problem was that movement between consistent states required changes authorized by different users. Possible, but unappealing, solutions to this problem included eliminating the constraint, coordinating the changes through someone (such as the Database Administrator) who had the authority to make all the necessary changes, or reifying the problem by defining a "change request" which each party could modify to indicate the changes they wanted and which, when "complete" would be performed by the system on behalf of each party.

Instead, by applying our approach to tolerate temporary inconsistencies of the constraint, any project leader can add or delete, or increase or decrease the amount that an employee works on their project. That change violates the constraint and causes the *Work-Pollution* Pollution Marker to be inserted for that employee on the date of the change. A similar violation arises from a change made by the Business Office in the amount an employee ought to work. In either case, the Pollution Marker automatically disappears when the violation is resolved.

Toleration of these inconsistencies was achieved by altering the application's report generators so that they added a notation on the report when they used the amount an employee ought to work, the total they actually worked, or the amount they worked on a particular project on a date for which the employee's work data was inconsistent.

Our two implementations of these Pollution Markers are presented in the next section with illustrations of their use. Then we describe our techniques for resolving the inconsistencies and compare our approach with other work.

3 Implementation

We have created two implementations of Pollution Markers with identical semantics that maintain their joint occurrence with violations of the constraint.[1] The first is more operational and easier to understand. The second is more declarative and elegant. Both are illustrated on the real example described above.

[1] These implementations have been added to our growing library of Evolution Transformations which support the evolution of specifications [Bal85, Joh88]. These semantics changing transformations facilitate the creation of the next version of a specification, rather than the conversion of a specification into an implementation.

3.1 Paired Maintenance Constraints

The first implementation replaces the original constraint by a pair of constraints that maintain the joint occurrence relationship between the consistency violations and the Pollution Markers. The first of these, the Marker Insertion Constraint, requires that whenever the original constraint is violated the corresponding Pollution Marker (i.e. the one with the instantiated values of the constraint's leading quantified variables) be present; its repair program asserts that Pollution Marker and is executed whenever the constraint is violated. Thus, if the original constraint is violated, the new constraint will also be violated because the corresponding Pollution Marker is not already present. This causes the constraint's repair to be invoked (within the same atomic update) which asserts the necessary Pollution Marker. Hence, the effect of this new constraint is to maintain the joint appearance of the consistency violation and its Pollution Marker.

The second element of the pair, the Marker Retraction Constraint, is similar, except that it ensures that when the consistency violation disappears, so does the corresponding Pollution Marker. Because these two constraints are the only ones that assert and retract the Pollution Markers, the Markers will thus jointly appear and disappear with the consistency violations.

In this implementation, the original constraint[2]:

```
Require For-All Person, Date,
  Required-Amount, Total-Amount
    Where Required-Amount =
      Required%-Date<Person,Date>
    Where Total-Amount =
      Total%-Date<Person,Date>
That Required-Amount = Total-Amount
```

is replaced by the two following constraints. The Marker Insertion Constraint adds the Pollution Marker *Work-Pollution* when an inconsistency is detected, and the Marker Retraction Constraint removes it when the inconsistency is resolved:

```
Marker Insertion Constraint:
  Require For-All Person, Date,
    Required-Amount, Total-Amount
      Where Required-Amount =
        Required%-Date<Person,Date>
      Where Total-Amount =
        Total%-Date<Person,Date>
  That Required-Amount ≠ Total-Amount
        Implies Work-Pollution<Person,Date,
          Required-Amount,Total-Amount>
  Repair-By Assert Work-Pollution<Person,
    Date,Required-Amount, Total-Amount>
```

```
Marker Retraction Constraint:
  Prohibit Exists Person, Date,
    Required-Amount, Total-Amount
      Where Required-Amount =
        Required%-Date<Person,Date>
      Where Total-Amount =
        Total%-Date<Person,Date>
  That Required-Amount = Total-Amount
        And Work-Pollution<Person,Date,
          Required-Amount,Total-Amount>
  Repair-By Retract Work-Pollution<Person,
    Date,Required-Amount,Total-Amount>
```

[2] We have taken considerable liberty with the syntax to make it more readable. We have suppressed the value argument of the *Required%-Date* and *Total%-Date* relations by treating them as functions which return this value. Reserved words appear in boldface.

3.2 Derivation Rule

The second implementation of Pollution Markers defines them through logical derivations. These definitions state that instantiations of the Pollution Marker exist for those, and only those, violations of the original constraint. The "if, and only if" nature of these definitions allows them to define both the presence and absence of the Pollution Markers corresponding to the violation and resolution of the original constraint.

Performance Note: In our system [Coh89] derived relations can be cached and indexed just like normal stored relations. Thus, these Pollution Markers can be accessed just as efficiently as stored data even though they are derived.

In the derivation rule implementation, the original constraint:

```
Require For-All Person, Date,
  Required-Amount, Total-Amount
    Where Required-Amount =
      Required%-Date<Person,Date>
    Where Total-Amount =
      Total%-Date<Person,Date>
That Required-Amount = Total-Amount
```

becomes:

```
For-All Person, Date, Required-Amount,
  Total-Amount
    Where Required-Amount =
      Required%-Date<Person,Date>
    Where Total-Amount =
      Total%-Date<Person,Date>
When-And-Only-When
  Required-Amount ≠ Total-Amount
    Conclude Work-Pollution<Person,Date,
      Required-Amount,Total-Amount>
```

4 Resolving Inconsistencies

We've described how to modify constraints to explicitly identify their violations with Pollution Markers acting as guards that can be used to modify the application's behavior so that the inconsistency can be tolerated by screening code sensitive to the inconsistency or adjusting its behavior. This relaxes the synchronization problem among multiple users cooperating on some compound update and allows them to operate asynchronously.

However, these inconsistencies were introduced as waystations toward reaching the next valid state representing the completed compound update. As such they are expected to be temporary.

That is, there is an implicit informal goal of minimizing both the number and duration of inconsistencies. Towards this goal, we have identified three techniques for resolving these inconsistencies, the first two of which have been implemented.

The first is the use of specialized automated methods for resolving particular inconsistencies. One example of such a specialized method that occurred in the employee database application we described before is that if an inconsistency arises because an employee works too much, and one of the projects involved is a *Phantom* project (i.e. an artificial project used as a placeholder in planning data), then reduce the effort on that phantom project to the level necessary to resolve the inconsistency. These methods are encoded as repairs to constraints whose condition acts as the method's trigger.

While this automatic repair technique is great for particular situations, it is usually not applicable because there are many possible ways of resolving the inconsistency (in fact, automated repairs handle less than 5% of the inconsistencies of employees working the wrong amount). To resolve this ambiguity, additional user input must be solicited from the same or additional users.

We elicit the participation of these users, via our second technique, by informing them of the inconsistency. The notification occurs by electronic mail and is part of a general application support package that disseminates information about database changes to interested users. This package is driven by user supplied rules identifying which changes they are interested in (via general predicate) and the format they would like for the notification. This same mechanism is also used to inform users of the resolution of any inconsistencies in which they are interested (and hence previously notified when the inconsistency arose).

To use this facility, the application developer has to identify which users need to be informed about the introduction and deletion of Pollution Markers identifying the inconsistencies. For our example, this was the other project leaders and the Personnel Manager.[3]

Our final technique for resolving inconsistencies, which is not yet implemented, is to install a tracking system which augments the initial inconsistency notifications with followup messages for those inconsistencies that have not been resolved within some time period (which may depend upon the type of inconsistency and the particular instance).[4]

5 Related Work

Our approach is most similar to the individual based exceptions described in the Background section. Like them, exceptions are dealt with independent of the type system and inheritance on a condition by condition basis. The constraints are rewritten to allow the exceptions. However, whereas Borgida's constraints are rewritten for each individual exception as it is dynamically encountered, our constraints are rewritten only once, and that rewriting permits the entire class of inconsistencies arising from the asynchronous entry of partial updates. Moreover, while Borgida's exceptions are persistent, ours are temporary. Therefore, part of our approach deals with resolving the inconsistencies.

Both approaches dynamically detect exceptions and identify the specific exceptional data in special relations. Borgida's approach requires the user to explicitly identify the exceptional data, while ours automatically marks the inconsistency. However, because the exceptional data has been explicitly identified, he is able to detect all uses of such data in code not prepared for these exceptions. We, on the other hand, don't know specifically which data is incorrect — just that the combined set of data is inconsistent — and therefore can't automatically detect uses that are not prepared for the inconsistency. Finally, both approaches rely on user augmentation of the code to specify the special processing, or screening, of the exceptions.

In addition to these efforts addressing data exceptions, there is also an interesting relationship with

[3] These notification rules can be modified and/or delegated to others as organizational responsibilities change.

[4] Since the inconsistencies are explicitly represented and since time triggers are supported by our application support package, implementation of such a tracking facility should be quite straight forward.

some recent work on transactions by Pu, Kaiser, and Hutchinson [PKH88] addressing the times and set of users over which a constraint applies. They, too, are interested in group transactions — particularly in the context of cooperative software design — and have proposed "splitting" transactions into pieces with differing visibility scopes so that users in some group could see the changes made by the splits within the transaction, while those outside the group wouldn't see any changes until the entire transaction was committed.

Split-Transactions can be viewed as another approach to exception handling. Rather than dealing with an exception or an inconsistency after a completed transaction, Pu, Kaiser, and Hutchinson subdivide the transaction so that the multi-party coordinations can occur within it. As in all transactions, constraints are only applied when committed. Thus, during the transaction the effect is to remove the constraints.

The power of the Split-Transaction approach is the scoping (potentially multi-level) that it provides to allow groups to cooperatively prepare a compound update for a broader community. However, within the transaction, since the constraints are not in effect, the inconsistency is not explicit, and hence cannot be tracked, managed, or supported except by informal mechanisms outside the system.

But there is no reason why Split Transactions couldn't be combined with our own techniques for tolerating inconsistency to provide the advantages of both. While the inconsistency existed, it would be guarded by Pollution Markers, but it would only be visible to the group (dynamically formed) needed to resolve that inconsistency. After it was resolved, the Pollution Markers would be removed, and the new consistent state would be visible outside the group.

Similarly, there is no reason why our approach can't be used in combination with either the class based or individual based approaches to exceptions cited earlier since they address permanent exceptions while we focus on transient ones.

6 Conclusions

Starting with the recognition that constraints force all the updates needed to transit from one valid state to the next to occur simultaneously (i.e. within the same transaction), and that this restriction is particularly onerous in multi-party applications with shared responsibility and authority, we developed an approach that relaxes this strict synchronization requirement so that each party participating in a compound update can operate asynchronously.

Rather than eliminating the constraint or limiting its scope, we modify it so that violations cause Pollution Markers to automatically appear which can be used by the application to organize activity to resolve those violations and as guards to screen sensitive code from the inconsistent data or to conditionalize the code so that the inconsistency is tolerated. When the inconsistency is resolved, the Pollution Markers disappear and the application processes the data normally.

We developed and implemented two semantically equivalent techniques for automatically maintaining the correspondence between the consistency violations and the Pollution Markers, and two methods of resolving the inconsistencies that arise. These techniques and methods have been successfully used to relax the synchronization rigidity imposed by the constraints of an operational application. By using the Pollution Markers as guards in the report generators, the application was able to tolerate the temporary inconsistencies introduced, and allow its users to asynchronously initiate their portion of a multi-party compound update and rely on the information dissemination method to coordinate the involvement of the remaining parties to complete the compound update and resolve the inconsistency.

We've also argued that our approach is applicable to a wide range of systems — including software development environments. In fact, we plan to use this technique in our own software development environment, FSD [Bal86], to tolerate temporary inconsistency in formal/actual interface matching, undefined functions, and the arity of relations, and to maintain an agenda of "pending user actions" to resolve those inconsistencies.

We compared and contrasted our approach with previous work in exceptions (both class and individual based) and with Split Transactions. Our approach appears to be sufficiently modular and local in scope to be combined and used in conjunction with any or all of these efforts.

The main limitations of our work are that the goal of minimizing the number and duration of the tolerated inconsistencies is informal and implicit — though by explicating the inconsistency, its resolution can be tracked and handled within the system (a portion of which we've implemented to notify users about inconsistencies they might help resolve) — and the manual determination of which parts of the application are sensitive to the inconsistency and how they should be adapted.

References

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[Bal85] R. Balzer. Automated enhancement of knowledge representations. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 203–207. Morgan Kaufmann Publishers, 1985.

[Bal86] R. Balzer. Living in the next generation operating system. In *IFIPS Proceedings of the 10th World Computer Congress*, pages 283–291, 1986.

[Bor85] A. Borgida. Language features for flexible handling of exceptions in information systems. *ACM Transactions on Database Systems*, 10(4):565–603, December 1985.

[Bor88] A. Borgida. Modeling class hierarchies with contradictions. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*. ACM SIGMOD, 1988.

[BW85] A. Borgida and W. Williamson. Exceptions to constraints in databases: Live and learn! In *Proceedings of the 11th VLDB Conference*, 1985.

[Coh89] D. Cohen. Ap5 manual. Technical Report ISI-TM-89-368, Information Sciences Institute, March 1989.

[Dij76] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Joh88] L. Johnson. Deriving specifications from requirements. In *Proceedings of the 10th International Conference on Software Engineering*, pages 428–438. Computer Society Press, 1988.

[MBW80] J. Mylopoulos, P.A. Bernstein, and H.K.T. Wong. A language facility for designing interactive database-intensive systems. *ACM Transactions on Database Systems*, 5(2):185–207, June 1980.

[PKH88] C. Pu, G. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the 14th VLDB Conference*, pages 26–37, 1988.

[SFL81] J.M. Smith, S. Fox, and T. Landers. Reference manual for adaplex. Technical Report CCA-81-02, Computer Corp. of America, January 1981.