Exception Handling in the Spreadsheet Paradigm

Margaret Burnett, *Senior Member*, *IEEE*, Anurag Agrawal, and Pieter van Zee

Abstract—Exception handling is widely regarded as a necessity in programming languages today and almost every programming language currently used for professional software development supports some form of it. However, spreadsheet systems, which may be the most widely used type of "programming language" today in terms of number of users using it to create "programs" (spreadsheets), have traditionally had only extremely limited support for exception handling. Spreadsheet system users range from end users to professional programmers and this wide range suggests that an approach to exception handling for spreadsheet systems needs to be compatible with the equational reasoning model of spreadsheet formulas, yet feature expressive power comparable to that found in other programming languages. In this paper, we present an approach to exception handling for spreadsheet system users that is aimed at this goal. Some of the features of the approach are new; others are not new, but their effects on the programming language properties of spreadsheet systems have not been discussed before in the literature. We explore these properties, offer our solutions to problems that arise with these properties, and compare the functionality of the approach with that of exception handling approaches in other languages.

Index Terms—Exception handling, spreadsheets, end-user programming.

1 Introduction

In his landmark paper, Goodenough captured the notion of exception handling as a general purpose tool in the programmer's toolbox [17] and today, exception handling is widely acknowledged to be a device that can improve a program's software engineering characteristics, such as robustness, generality, and modularity. But while exception handling is commonly supported in programming languages today, especially those in the imperative, object-oriented, and functional paradigms, almost no work has been reported on exception handling in programming languages aimed at end users. Perhaps the most widely used of all end-user programming languages are spreadsheet systems. The lack of attention to exception handling and other software engineering aspects of spreadsheet systems may be one reason that a large number of spreadsheets in practical use contain faults (see [37], [38] for summaries of this literature).

Spreadsheets are simply collections of equations. A spreadsheet consists of cells and each cell (the lefthand side of the equation) can be given a formula (the righthand side of the equation). The collection of these formulas completely specifies the "program" logic. In particular, the spreadsheet's creator does not specify execution sequence; rather execution is automatically scheduled by the system in some dependency preserving order.

Exceptions can arise in spreadsheets just as in traditional programs, occurring, for example, in the cases of divisions by zero and dynamic type errors. Yet, incorporating most traditional approaches to exception handling into a spreadsheet system does not seem viable for end users, because doing so would require these users to deal with concepts not usually present in spreadsheet systems, such as control flow or higher-order functions.

The issue of how to support exception handling in spreadsheets is compounded by the fact that only *some* of the creators of spreadsheets are end users. Other creators of spreadsheets include professional programmers creating spreadsheet templates for sale to other users, such as for income tax preparation and *informal programmers* [56]—users who, like professional programmers, write programs for others to use, but who are not fully trained professional programmers. Examples of informal programmers include an office manager creating budget spreadsheets for the office staff using a conventional spreadsheet system such as Excel® [33], a user interface designer creating a user interface using a spreadsheet-based user interface design tool such as C32 [36], and a scientist creating a graphical depiction of complex scientific data produced by a spreadsheet-based visualization system (e.g., [7]). With this mixture of audiences in mind, our goal was to develop a "gentle slope" approach to exception handling for spreadsheets that would be simple enough for end users, yet expressive enough for informal programmers and professional programmers to use for full featured exception handling.

In this paper, we examine an approach to exception handling commonly found in spreadsheet systems called the *error value model* and show that, given suitable

- *M. Burnett is with the Department of Computer Science, Oregon State University, Corvallis, OR 97331. E-mail: burnett@cs.orst.edu.*
- *A. Agrawal is with Qualcomm, Inc., 6455 Lusk Blvd., San Diego, CA 92121. E-mail: anuraga@qualcomm.com.*
- *P. van Zee is with Hewlet Packard, Inc., 1000 NE Circle Blvd., Corvallis, OR 97331. E-mail: pieter_van_zee@hp.com.*

Manuscript received 14 Mar. 1999; revised 13 Dec. 1999; accepted 28 Jan. 2000.
Recommended for acceptance by A. Romanovsky.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 111494.

abstraction mechanisms, the approach can provide as much functionality as many traditional approaches to exception handling, but without altering the spreadsheet paradigm's evaluation model. We begin with a review of literature relevant to spreadsheet systems and exception handling and a brief introduction to Forms/3, the research spreadsheet system in which our work has been prototyped. Characteristics of spreadsheet systems are identified that are not commonly found in other programming languages and the implications of these characteristics on exception handling are discussed, followed by problems with error value exception handling in its basic form and our solutions. Ways are then presented to increase the expressive power of error value exception handling through spreadsheet-oriented abstraction mechanisms and the resulting increase is demonstrated by implementing replacement value exception handling using the error value model. Finally, to evaluate the extent to which the approach might be considered "full featured," the approach is compared with the exception handling approaches of three modern programming languages in practical use.

2 Background and Related Work

This paper uses conventional exception-related terminology: The term *exception handling* means a collection of mechanisms that support the detection, signaling, and after-the-fact handling of exceptions. (Before-the-fact exception-oriented code is better classified as *exception prevention*.) Sebesta defines an *exception* as any unusual event, erroneous or not, that is detectable either by hardware or software and that may require special processing [51]. An exception is *signaled* when it is detected and the *signaler* is the operation being executed when the exception is signaled. An *exception handler* is the special processing code that is executed as a result of an exception being signaled.

2.1 Spreadsheet Languages

We use the term *spreadsheet languages*[1] to refer to all systems that follow the spreadsheet paradigm, from commercial spreadsheet systems to more sophisticated research languages following the spreadsheet paradigm. The essence of this paradigm is expressed well by Alan Kay's *value rule*, which states that a cell's value is defined solely by the formula explicitly given it by the user [27]. There are similarities with the dataflow paradigm, but several features of the spreadsheet paradigm distinguish it from the way the dataflow paradigm is conventionally realized. These differences are in the language design and in the implementation strategies. They arise from the need to simplify language features in order to accommodate end users' lack of programming training and to provide immediate and automatic execution of every visible cell after every formula edit.

In recent years, the spreadsheet paradigm has come under study for its potential in domains beyond budgets and accounting and a goal of our approach to exception handling is to be appropriate for research spreadsheet languages that have been created for these new domains. Many of these research systems have been oriented toward supporting graphics-related programming, primarily for end users and informal programmers. This line of spreadsheet research began with a pioneering language in this domain, NoPumpG [30], followed by its successor NoPumpII [65], two spreadsheet languages designed to support interactive graphics. Our research system, Forms/3 [4], [5], is another spreadsheet language that supports interactive graphics. In addition, it supports abstraction, as will be seen later in this paper. Penguims [25], a spreadsheet language for specifying user interfaces, is similar to Forms/3 in its emphasis on abstraction—it provides the capability to collect cells together into objects—but unlike Forms/3, it employs several techniques that do not conform to the spreadsheet value rule, such as interactor objects that can modify the formulas of other cells and imperative code similar to macros. Action Graphics [26] is a spreadsheet language that supports graphical visualizations and animations—primarily of program or data structures—through functions that cause side effects. Chi et al. recently developed a research spreadsheet language to support high quality visualizations of complex data such as those needed in scientific applications [7].

Diverse experimental spreadsheet languages exploring other novel approaches to spreadsheet development have also been created. For example, Smedley, Cox, and Byrne have incorporated the visual programming language Prograph and user interface objects into a spreadsheet language in order to provide a graphical interface for spreadsheet input and feedback [10], [53]. Wilde's WYSIWYC spreadsheet language [64] aims to improve traditional spreadsheet programming by making cell formulas visible and by making the visible structure of the spreadsheet match its computational structure. C32 [36] is a spreadsheet language that uses graphical techniques along with inference to specify constraints in user interfaces. The spreadsheet language Formulate includes the use of voice, handwriting, and gestures as input modalities for entering standard spreadsheet formulas [29] and also provides extensive intelligent assistance in specifying solutions to matrix manipulation problems [59], [62].

As will be seen later in this paper, an important asset to the expressive power of exception handling in spreadsheet languages is support for abstraction. The spreadsheet languages with the most support for abstraction are Formulate, Forms/3, and Penguims. Formulate provides an explicit approach to procedural abstraction, allowing the user to name spreadsheets and provide textual calling protocols to treat them as first order functions, but does not support data abstraction. Forms/3 supports both procedural abstraction and data abstraction without leaving the spreadsheet paradigm, in a manner that will be described in later sections. Penguims's support for abstraction is extensive and is done in a manner similar to Forms/3's. (However, Penguims does not make use of this feature for exception handling; for example, there is no facility for user-defined exceptions.) These three spreadsheet languages are unusual in their support for abstraction;

1. We have chosen this terminology to emphasize the fact that even commercial spreadsheet systems are, indeed, languages for programming, although they differ in audience, application, and environment from traditional programming languages.

most spreadsheet languages have either no abstraction at all, or require the user to switch to another paradigm (through trapdoors to traditional programming languages or imperative macros) when abstraction is needed. The overall lack of abstraction in spreadsheet languages may be the main reason there has been little work on exception handling in this paradigm.

2.2 Exception Handling Models for Spreadsheet Languages

In the *error value model* of exception handling, a function's or operator's return value, if an exception occurred, is an identifiable error value. For example, in Excel, when a primitive operation detects an exception, it returns one of seven possible error values. Each error value indicates a different cause, such as division by zero. Operations are provided that test for error values, discriminate among them, and generate them. Using these mechanisms, a spreadsheet's creator can detect when an exception has occurred and provide the desired exception handling. When nothing is done in a formula to handle the presence of an error value, the error values propagate to cells that reference the cells containing error values. The error value model is different from the use of status flags in traditional programming languages in that the *primary* return value is an error, which will propagate in turn to every use of that return value; hence it cannot be ignored and lost as can status flags.

The Excel rendition of the error value model is representative of exception handling in commercial and research spreadsheet languages. A strong advantage of the approach is that it is attractively simple. Unfortunately, however, it does not support all of the generally accepted principles of exception handling. For example, as Goodenough was first to point out, exceptions are not necessarily errors [17]; this implies that support for user-defined exception abstractions can expand the generality of an exception handling mechanism. Goodenough further observed that an exception's significance is often known only outside the signaling operation and concluded that the invoker of the signaling operation should have some control as to how the exception should be handled. Investigating these observations further, Yemini and Berry presented potential software engineering advantages in the use of exception handling, from which they derived a set of design guidelines and introduced the *replacement value model*[2] as an approach that follows these guidelines [68]. The guidelines were:

- Handlers should be allowed to have formal parameters. This decreases coupling among different potential signalers and potential invokers (no shared global variables) and increases reusability of the handlers.
- To preserve information hiding, unhandled signals should not automatically propagate along the chain of invokers. If the details of an exception are automatically propagated, information hiding is violated; however, explicit propagation is permissable because it supports information hiding by allowing abstraction of exception information.
- Data and procedural abstractions should be able to include exceptions in their definitions. This improves the fidelity of the definition of such abstractions.
- Exception handling should integrate fully with a language's scope rules and type system.
- Exception handling features should be designed so that their addition to a language does not reduce the language's suitability for formal verification.

The replacement value model is based upon providing a *replacement value* to use in handling exceptions in a manner that follows these guidelines. Although the context of Yemini's and Berry's view of the replacement value model was in the imperative world, the replacement value model has since been adapted to functional languages (e.g., [2]) and forms the basis of some of the work on exception handling in the functional language community. Later in this paper, we will use the replacement value model as a measuring standard, showing that it is possible to meet all, except one, of its guidelines by implementing it using the error value model in the spreadsheet paradigm.

2.3 Exception Handling in Other Applicative Languages

Most research on exception handling has been in imperative [11], [17], [31], [68], object-oriented [14], [34], [48], [66], and functional languages [2], [19], [43], [60]. The approaches to exception handling in the imperative and object-oriented paradigms usually accomplish their purposes in part by allowing the programmer to alter execution sequence. Since spreadsheet formulas do not express sequence (other than what can be derived from the dataflow dependencies in the formulas), these approaches cannot be used in the spreadsheet evaluation model.

Like spreadsheet languages, functional languages are in the applicative language family. Although many of the exception handling approaches devised for applicative languages have been at least partially control-flow based, the error value model has also been used in various ways in some applicative and functional languages (e.g., [15, 19, 60]). However, spreadsheet languages are first-order languages; that is, functions cannot be passed as parameters or returned as results. Hence, without fundamental changes to the evaluation model, spreadsheet languages cannot rely on higher-order functions, such as using continuation-passing style (e.g., [35]) or monads (e.g., [40]), to achieve exception handling.

Several problems can arise when exception handling is introduced into functional and other applicative languages [2], [19], [43]. For example, modern applicative programming languages treat same-level arguments as if they are evaluated in parallel; no order of evaluation is specified. But if two or more same-level exceptional points exist within a function call, unless an order of the signals is asserted, referential transparency will not be maintained. Another potential problem is loss of laziness. These problems can also arise in the spreadsheet paradigm.

2. Yemeni and Berry named it the "replacement model." We use the phrase "replacement value model" in this paper because it emphasizes the nature of the approach.

2.4 Exception Handling in Visual Programming Languages

Spreadsheet languages are an example of at least partially visual programming languages (VPLs), a family of languages whose semantics and/or evaluation models rest upon the use of multiple dimensions and/or immediate visual feedback. VPLs are especially relevant to this paper because they are intended for diverse audiences, ranging from end users to informal programmers to professional programmers. Interestingly, unlike traditional languages, declarative paradigms predominate in the visual programming research community, especially rule-based, constraint-based, and dataflow. This would imply that the exception handling research in the VPL community would likely be applicable to the spreadsheet paradigm. However, to date, there has been only a little support for exception handling capabilities in VPLs, whether aimed at end users, professional programmers, or informal programmers.

The visual languages most commonly found in the software engineering research community are languages for modeling and/or specifying some aspect of software or software process. Popular approaches include variations on statecharts, Petri nets, control flow diagrams, data flow diagrams, and hybrids among these [16], [20], [39], [50], [52], [54]. Many such languages are for specification only and are not executable. Of those that are executable, some do support reacting to high-level or domain-oriented events and exceptions that can arise in the domain being modeled, but, due to their high-level focus, they provide little support for handling low-level exceptions that arise in the domain of computation itself, such as divide by zero, unavailability of an I/O device, exhaustion of colors available to a display, etc.

VPLs aimed directly at programming have also been limited in supporting exception handling. For example, Fabrik [32], an early dataflow VPL, is still representative of declarative approaches to exception handling in VPLs. Fabrik supports system-level errors only, under the error value model. In Fabrik, if a component cannot compute, the values on the output pins are invalid and this invalidity is propagated to the connected input pins. Connections carrying invalid values appear as dashed lines. No facility for user-defined exceptions is provided.

The rule-based paradigm is becoming increasingly popular for VPLs aimed at end users (e.g., [22], [42], [45], [46]). Although support for exception handling seems like a natural fit for this paradigm, given the similarity between the "whenever-like" behavior of rules and the behavior needed for invoking handlers, at least some kind of support would be needed for identifying and handling built-in exceptions. To date, our searches through the VPL research literature have turned up no reports of this kind of support.

Descriptions of Prograph-based research [9], [47] include the only detailed discussions on exception handling in programmer-oriented VPLs that we have been able to locate. Prograph is a VPL intended for professional programmers that combines the dataflow, object-oriented, and imperative paradigms. In Prograph, exception handling is control-flow oriented. There are constructs that allow the programmer to explicitly signal exceptions and to handle exceptions through termination and transfer of control to other sections of the program. Computation-generated exceptions, however, such as divide by zero, are not supported in Prograph itself. The latter of the two mentioned Prograph-based projects explores extending the original Prograph syntax to make it Java compatible and translates the visual code to Java. In this project, the control-flow-based, object-oriented exception handling style of Java is supported. This approach is full featured, but its control flow orientation would not be suitable for spreadsheet languages because of its incompatibility with the spreadsheet value rule.

2.5 A Brief Introduction to Forms/3

Forms/3 is the research spreadsheet language in which we have prototyped our approach to exception handling. Forms/3 includes the usual spreadsheet language features (see Table 1), but extends them by providing support for abstraction mechanisms (through a device similar to "linked spreadsheets"); by supporting several visual and gestural mechanisms for specifying and exploring formulas; and by supporting a variety of types, including graphical types [4], [5].

Another difference from commercial spreadsheet systems is that in Forms/3 not all cells need to be locked into a grid; it is also possible to have standalone cells and grids. (This difference is not relevant to support for exception handling.) However, like commercial spreadsheet systems, spreadsheets, called *forms* in Forms/3, are the basic organizational units, and cells are the computational units. Because each cell's value is determined by its formula, a spreadsheet's output is entirely determined by the cells' formulas and by the visible attributes of these cells (such as their location and the visibility of borders).

Most spreadsheet languages are dynamically typed; hence run-time type errors are one group of errors that can signal exceptions. Although we have experimented with static typing systems for Forms/3 [3], [13], they are not a part of this work on exception handling. Our approach to exception handling is presented in the context of the dynamic typing that is usual for spreadsheet languages.

3 The Application of Error Value Exception Handling to Spreadsheets

3.1 Unusual Characteristics of Spreadsheet Languages

Liveness is a term coined by Tanimoto to describe the amount and immediacy of *semantic* feedback provided by a language environment [55]. Tanimoto described four levels of liveness. At level 1 no semantics are implied to the computer and, hence, no feedback about a program is provided to the user. An example of level 1 is an entity-relationship diagram for documentation. At level 2 the user can obtain semantic feedback about a portion of a program, but it is not provided automatically. Compilers support level 2 liveness for final output values and interpreters do so without being restricted to final output values. At level 3, incremental semantic feedback is automatically provided whenever the user performs an incremental program edit and all affected on-screen values are automatically

TABLE 1
The Grammar for the Subset of Forms/3's Formula Language Used in this Paper

formula	::=	*Blank* | expr
expr	::=	*Constant* | ref | infixExpr | prefixExpr | ifExpr | composeExpr | (expr)
infixExpr	::=	subExpr infixOperator subExpr
prefixExpr	::=	nullaryOperator | unaryPrefixOperator subExpr | binaryPrefixOperator subExpr subExpr
ifExpr	::=	IF subExpr THEN subExpr ELSE subExpr |
		IF subExpr THEN subExpr
composeExpr	::=	COMPOSE subExpr AT (subexpr subexpr) |
		COMPOSE subExpr AT (subexpr subexpr) withClauses
subExpr	::=	*Constant* | ref | (expr)
withClauses	::=	WITH subexpr AT (subexpr subexpr) |
		WITH subexpr AT (subexpr subexpr) withClauses
infixOperator	::=	+ | - | * | / | AND | OR | MOD | = | > | < | ...
nullaryOperator	::=	ERROR | ...
unaryPrefixOperator	::=	- | ROUND | ABS | WIDTH | HEIGHT | ERROR? | CIRCLE | GLYPH | ...
binaryPrefixOperator	::=	APPEND | LINE | ...
ref	::=	cellRef | *FormID* : cellRef
cellRef	::=	*CellID* | *GridID* | *GridID* [subscripts]
subscripts	::=	gridSubscript@gridSubscript
gridSubscript	::=	expr

(Figs. 1, 2, 5, 6, 7, and 8 provide examples of formulas following this grammar.) As the top section shows, Froms/3 has the usual spreadsheet formula operators and also some operators supporting computation on grids and on graphics. The bottom section shows cell reference syntax, which includes row/column referencing for cells that are in a grid. There are also four pseudo references—I, J, LASTROW, and LASTCOL—that can be used in grid formulas. Including these in the grammar is straightforward, but tedious and we have omitted them for brevity.

redisplayed. This ensures the consistency of display state and system state if the only trigger for system state changes is user editing. The automatic recalculation feature of spreadsheet languages supports level 3 liveness. At level 4, the system responds to program edits as in level 3 and to other events as well, such as system clock ticks and mouse clicks over time, ensuring that all data on display accurately reflects the current state of the system as computations continue to evolve. Forms/3 is a spreadsheet language that supports time-related computations and provides feedback about them at liveness level 4. In this paper, we will simply use the terms *live* and *liveness* to describe a system supporting level 3 or higher. As will be discussed later, the liveness of spreadsheet languages provides both opportunities and problems for exception handling.

One implication of the liveness of spreadsheet languages is that it means there are multiple output variables (cells) and that these output cells are not distinguished from any other cells. That is, any cell is an output cell if it is on the screen. A further implication of the fact that every on-screen cell is an output cell is that spreadsheet languages involve multiple threads, at least in theory and often in practice.

In spite of the presence of liveness, it is possible for a spreadsheet evaluation engine to use lazy evaluation and some spreadsheet languages do so (e.g., Forms/3 and C32 [36]). In the context of liveness, laziness means that only those cells needed to keep the on-screen cells up-to-date will be calculated. Because of the presence of scrollbars and other event-driven navigation devices, which cells are on-screen may change at any moment. For example, "final" answers may be scrolled off the screen by the user, with intermediate cells such as exception handlers scrolled onto the screen to suddenly become "outputs." This has particular impact on lazy spreadsheet languages, but can also affect exception handling in eager languages. An approach to exception handling for spreadsheet languages must deal with this flexibility and would ideally work for either lazy or eager evaluation strategies.

3.2 Basic Features of Exception Handling in Forms/3

3.2.1 Default Exception Handling of Built-in Exceptions

In the error value model, a built-in exception type can allow instantiation of an exception that is part of the language definition. Instances of built-in exceptions can be signaled by the language's runtime system. Hence, in Forms/3, the default way to handle a built-in exception is to terminate the signaling operator, return a value of type *error*, and resume execution, as in Fig. 1. The error value can be returned at the subexpression level or the cell level and we will discuss the advantages and disadvantages of each possibility in Section 4. Since the error value model focuses on values, not on control flow, there is no transfer of control, either in default handling or in explicit handling that can be provided by the spreadsheet creator. Rather, execution simply resumes at the operator that would have been scheduled next if no exception had occurred.

The analog clock spreadsheet in Fig. 2 demonstrates an application of default handling of built-in exceptions in Forms/3 in a type of application oriented toward graphical output that might have been written by an informal (not professionally trained) programmer. This spreadsheet demonstrates the error value model as found in most spreadsheet languages (although many spreadsheet

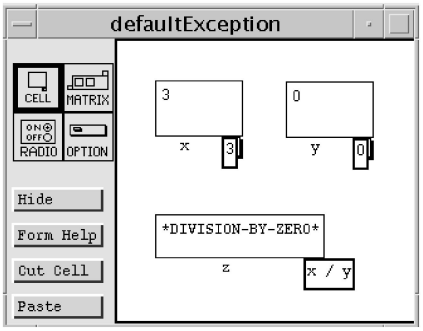


Fig. 1. A Forms/3 spreadsheet with three cells demonstrates the error value model for built-in exceptions. (In Forms/3, the spreadsheet creator can provide optional names for the cells, such as *x*, *y*, and *z*. Also, any cell's formula can be displayed with the cell's value, as has been done here for all three cells simultaneously.)

languages do not allow some other features present in the clock, such as graphical types). The clock spreadsheet takes two integers, representing the time of day in hours and minutes and displays the corresponding analog clock. After testing is complete, those integer input cells would be replaced by references to the actual system clock. The x- and y-positions of the clock's hands are computed by cells *minutex*, *minutey*, *hourx*, and *houry* based on x- and y-axes that each run from -15 to +15, with 0 intended as the location of the clock's pin. Cell *theClock* references the results of the cells *minuteHand*, *hourHand, face,* and *pivot* to assemble the clock components into one unit. In this example, the cell references were entered by pointing and the formula arranging the clock components was demonstrated by dragging the components together and rubber banding the result. (The clock components were then separated for readability of their formulas.) Alternatively, these formulas could have been typed in exactly as they are shown in the figure. The combination of lazy evaluation with liveness in Forms/3 causes execution of formulas to be automatically scheduled for every cell that is currently on the screen as well as for any other cells needed to compute those on-screen cells.

Because the spreadsheet's creator has not provided any exception handling code in the formulas in Fig. 2, all exception handling is done by the system's default handlers.

3.2.2 Explicit Exception Handling: If-Then-Else + Applicative Semantics + Liveness = Constraints

The example so far shows the system signaling exceptions by generating error values. (In Forms/3, these values can also be generated explicitly by spreadsheet creators via the *error* operator; most other spreadsheet languages also offer such a facility.) Spreadsheet creators can capitalize upon the presence of error values by specifying their own exception handlers: ordinary *if-then-else* formulas defining calculations predicated on exceptions having arisen. For example, suppose the spreadsheet creator renames *theClock* to *goodClock*, adds a *badClock* cell containing a sketch of a broken clock (drawn using an ordinary X-Windows bitmap editor and imported using the *glyph* operator with the filename as its argument), and creates a new *theClock* cell with formula:

if (error? (minuteHand) or error? (hourHand))
then badClock else goodClock

(The operation *error?* tests whether a value is of type *error*.) Fig. 3 shows the result of these three changes.

This example illustrates an important characteristic of spreadsheet languages that can be exploited to allow exception handling under the error value model: The ordinary *if-then-else* conditional construct, when paired with the continuous demands for output of a live environment, provides the constraint-based or rule-based functionality that is at the heart of exception handling. That is, explicit exception handling is a definition of computations needed *whenever* an exception arises, just as are constraints and rules. (In fact, the term *one-way constraints* coined by Myers is now frequently used to describe the spreadsheet evaluation model [36].) This compatibility with constraint-based semantics is because of the combination that 1) the variables' definitions (cells' formulas) entirely define all the relationships in the spreadsheet and 2) wherever output is produced, the system must automatically maintain all values contributing to the output. As these two characteristics emphasize, the sequence of execution is always "soon enough"; that is, if a cell's formula (constraint) is relevant, it will automatically be scheduled to execute before any cell affected by it.

A noteworthy implication of this whenever-like *if* in spreadsheet languages is that the large difference that exists in most programming languages between exception handling (after an exception occurs) and exception prevention (before an exception occurs) is reduced. In traditional programming languages, exception prevention is normally done using *if* statements or expressions to test for conditions that could cause exceptions to be signaled and this can seem attractive to programmers—especially those who are relatively inexperienced—since they already know the *if*. On the other hand, traditional exception handling often requires special-purpose constructs (such as *on, try*, and/or *catch*) and deviations from the usual evaluation model. With the error value model set in a spreadsheet language however, exception prevention and exception handling are both done using the *if*, requiring only the understanding of what the error type is and how to detect its presence with the *error?* operator. Hence, the difference between prevention and handling (and the size of the learning curve required to move from prevention to handling) becomes small.

One reason exception handling is promoted by language researchers over exception prevention is modularity: Exception handling code is usually more easily separated from the main algorithm than is exception prevention code. This capability is also present in the approach to exception handling discussed here, because the exception handling cells can be placed well away from the "logic" cells, since the physical locations of these cells do not affect the spreadsheet's behavior.

3.2.3 Expressiveness of Explicit Exception Handlers

In many languages, one explicit exception handler can handle all exceptions of a certain type. The approach described here is not as expressive as that. Instead, in the examples so far, an exception handler monitors activity in the particular cell(s) referenced in the *if* predicate, which is

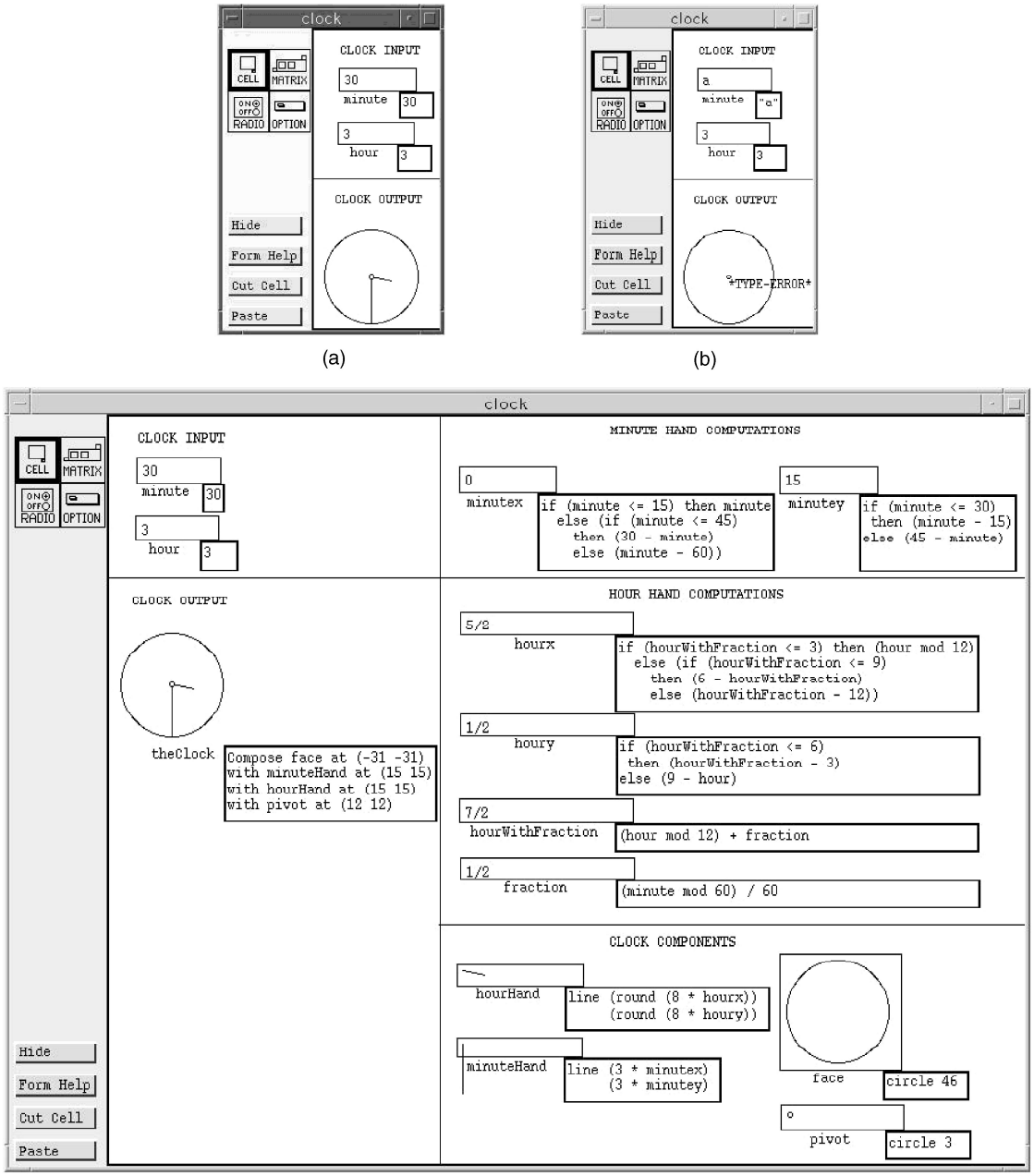


Fig. 2. A user's view of the clock spreadsheet does not include the formulas or the cells the spreadsheet's creator has chosen to hide. Here, two user views are shown: (a) with correct input and (b) with erroneous input. (c) The creator's view: The spreadsheet creator did not include any exception handling code in this spreadsheet; it simply defaults to the default exception handling provided by the system.

more like handling a particular instance of an exception than an entire category of exceptions. As will be discussed in the next few sections, it is not possible in this approach for a single handler to handle all exceptions of a certain type, but it is possible for a handler to handle multiple instances of an exception, given certain abstraction capabilities.

4 Issues Raised

Except for the inclusion of graphics, the approach to exception handling we have described to this point is representative of that found in most spreadsheet languages. Approaches like this (also sans graphics) have also been included in some other applicative languages. However, even though the model is simple, several issues are raised by its incorporation into the spreadsheet paradigm. In this section, we consider these issues and offer our solutions to them.

4.1 Issue 1: Under What Conditions Should Type Exceptions Be Signaled?

Commercial spreadsheet systems, which are dynamically typed, have traditionally been more forgiving than other kinds of languages—even those that are also dynamically typed—in overlooking type errors. For example, when summing columns that contain nonnumeric entries, Excel

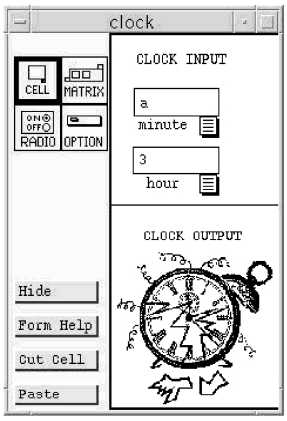


Fig. 3. The user's view of the clock after the spreadsheet's creator has specified an exception handler.

simply treats those entries as zeros. However, this forgiving approach allows type errors to slip by unnoticed. For example, consider the following example of an end-user spreadsheet: a conventional grades spreadsheet kept by a teacher, as in Fig. 4. If the *Course* column's formulas were something like:

(HW1 + HW2 + HW3 + Midterm + Final)/ 5

then a *HW2* entry for one of the students (say, Smith) such as "*N/A (ill)*" would be treated as a zero instead of being omitted from the calculations. Since it is fairly common for teachers to have special circumstances arise for individual students such as this, it seems important for a type exception to be raised if the teacher forgot to remove the *HW2* reference from Smith's *Course* formula. To solve this problem, in Forms/3 and in some other spreadsheet languages, inappropriate types are never ignored; rather, a type exception is raised. In this example, Smith's *Course* cell would display *Type-Error*. However, this decision affects exception propagation.

4.2 Issue 2: How Far Should Exceptions Be Propagated?

Consider a formula for some cell X such as:

$$(3/0) + 5.$$

In evaluating this expression, the subexpression in parentheses evaluates to a *Division-By-Zero* value (of type *error)*. From this point, one possibility would be to continue evaluation, trying to add *Division-By-Zero* to five. In this case, X's final answer would be

Division-By-Zero + 5 = Type-Error

since the system cannot add an error to a number. This solution would have the advantages of being consistent with the resolution just presented to the previous section's issue and of being easily done without altering the evaluation model. The problem with it is that the result would not be very informative. Since the exception occurred inside a subexpression, no notification of the *Division-By-Zero* would ever be displayed. Instead, the *Type-Error* exception displayed would be different from the *Division-By-Zero* exception that actually occurred, which would be misleading. Since a cornerstone of spreadsheet languages is immediate visual feedback, this misleading display would be a serious problem.

Another possibility would be to return *Division-By-Zero* as the formula's result. This would solve the misleading display of the first approach. It would in essence be a redefinition of the semantics of each operator so that, if one of its arguments were an error, it would behave as an identity function on that argument, returning the incoming error value. However, this fix could cause new problems. One of these problems is with information hiding.

When Yemini and Berry wrote about exception handling, they asserted that "automatic propagation of unhandled exceptions may compromise information hiding" (through the exposure and propagation of implementation details),

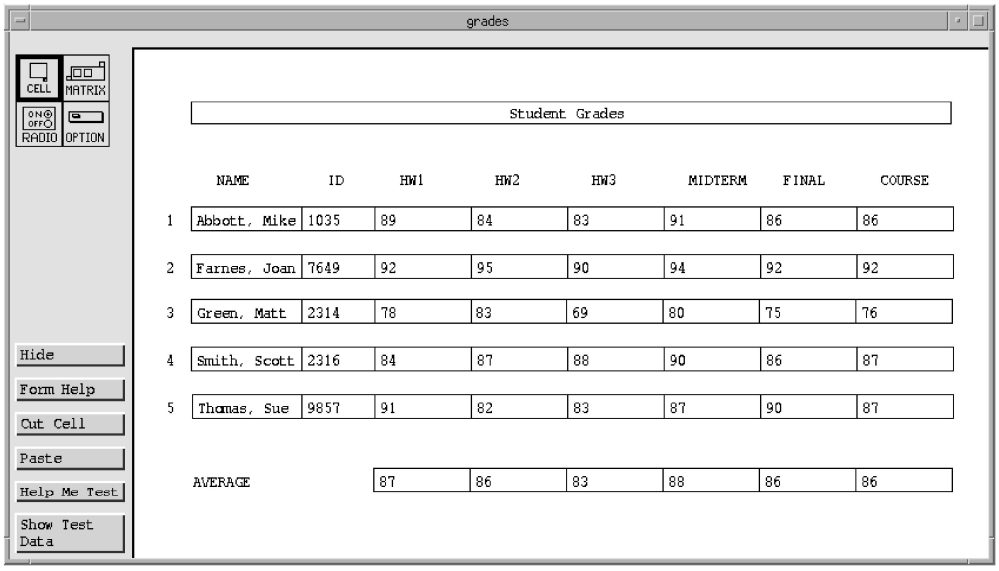


Fig. 4. Forms/3 spreadsheet for calculating student grades.

recommending instead that "explicit propagation ... be used to properly rename propagated exceptions" [68]. Languages that do not automatically propagate exceptions (e.g., ALEX [2]) support this recommendation.

There is no support (and, some might argue, not even a need) for information hiding in commercial spreadsheet systems, but it is an issue for research spreadsheet languages, some of which include abstraction mechanisms. (Forms/3 is one such language and does provide facilities for information hiding.) Thus, this issue is worth consideration, especially since some spreadsheet creators are professional programmers trying to create spreadsheets and templates that will be robust and maintainable over time.

To support Yemini's and Berry's recommendation in our approach to exception handling, the first possibility mentioned above—computing further instead of returning the original exception—would be a closer match than the second, although neither is a perfect fit. To try to achieve the closer match of the first possibility, it might seem that a compromise between the two possibilities is needed, such as propagating exceptions only up to the cell level and then allowing formulas accessing that cell to compute as normally as they are able, given an incoming exception for one argument. For example, cell X above would have as its value *Division-By-Zero*, which still addresses the lack of informativeness and then some other cell Y adding ten to that value would result in *Type-Error*. This, however, would destroy referential transparency, as we discuss next. Instead, in many spreadsheet languages, including Forms/3, the approach taken is to propagate, for most operators, exception arguments at every level, despite the resulting lack of information hiding. Since most other languages today likewise do not provide language support for Yemini's and Berry's information hiding recommendation, instead automatically propagating unhandled exceptions, one can at least argue that the spreadsheet paradigm is in good company in ignoring it.

4.3 Issue 3: Maintaining Referential Transparency

Referential transparency, which is sometimes described as "the concept that equals can be substituted for equals" [1], is a property of much use to users and implementers of programming languages. Its presence prevents many kinds of subtle, hard-to-explain bugs and facilitates informal reasoning about a program and this is important to those attempting to use the language. It also facilitates optimizations in the compiler or interpreter. The spreadsheet paradigm of cells and formulas without exception handling (and without extra-applicative devices such as macros) potentially has the property of referential transparency. For spreadsheet languages in which referential transparency is present, it would be advantageous for the approach to exception handling not to undermine this property.

That loss of referential transparency can occur when exception handling is introduced into other applicative languages is a well-known problem [19] and this suggests that care must be exercised if the problem is to be avoided in spreadsheet languages. In our use of the error value

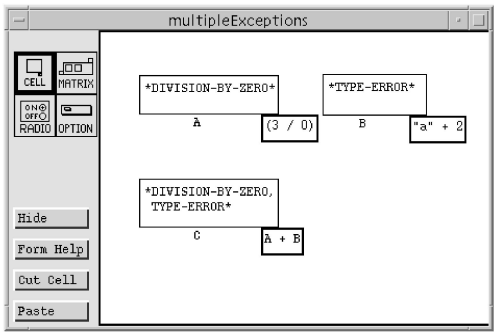


Fig. 5. A *multipleException* is a set of exceptions in Forms/3.

model, the possibility of propagating only up to the cell value would have caused loss of this property because the answers would not be the same if the same formula were split between two cells. For example, we stated that under this possibility, X above would have the value *Division-By-Zero*. Yet, if cell A had the formula:

$$3/0$$

and cell B had the formula:

$$A + 5,$$

then the result of B, which is mathematically the same as X, would have been *Type-Error* instead of *Division-By-Zero*.

The problem in this particular example was avoided by our choice to propagate at every granularity, not just up to the cell level. But problems with referential transparency caused by this choice still remain in the presence of multiple exceptions. For example, consider the spreadsheet in Fig. 5. A evaluates to *Division-By-Zero* and B evaluates to *Type-Error*. If C propagated only one of the exceptions (as is true in other spreadsheet languages), it would not return the same answer as it would if its formula were the mathematically equivalent

$$B + A.$$

The functional languages ALEX [2] and ML [21], [57] define this problem away by proposing left-to-right semantics. This is also the solution used in Excel. However, this restriction is not particularly compatible with parallel execution or with the dynamic execution scheduling common in spreadsheet languages. In Gerald [43], exceptions are given different priorities that determine which one of the exceptions will be returned, but the information about the other exceptions is lost. Forms/3 introduces a new solution to this problem that avoids such difficulties, propagating the collection of *all* the exceptions that arise in a computation via a *multipleException*, as in Fig. 5. Varying orders in which the exceptions are discovered are not an issue, because a *multipleException* is a set of exceptions, which takes advantage of the orderlessness of sets.

4.4 Issue 4: Preserving Laziness

Our solution above to the multiple-exceptions problem seems to cause a new problem: It seems incompatible with laziness, because it appears to require all arguments to be evaluated to gather up all the exceptions. Since even eager

TABLE 2
Semantics of an *if*-Expression of the Form: *if* predicate *then* then-expr *else* else-expr

predicate=true	predicate=false	predicate is an exception	predicate is some other non-Boolean
then-expr	else-expr	exception	exception

TABLE 3
Semantics of an *and*-Expression of the Form: arg1 *and* arg2

	arg1=true	arg1=false	arg1 is an exception	arg1 is some other non-Boolean
arg2=true	true	false	exception	exception
arg2=false	false	false	false	false
arg2 is an exception	exception	false	multiple exception	multiple exception
arg2 is some other non-Boolean	exception	false	multiple exception	multiple exception

TABLE 4
Semantics of an *or*-Expression of the Form arg1 *or* arg2

	arg1=true	arg1=false	arg1 is an exception	arg1 is some other non-Boolean
arg2=true	true	true	true	true
arg2=false	true	false	exception	exception
arg2 is an exception	true	exception	multiple exception	multiple exception
arg2 is some other non-Boolean	true	exception	multiple exception	multiple exception

spreadsheet languages use some laziness through short circuit evaluation of nonstrict[3] operators such as *if*, this problem is not confined to only lazy spreadsheet languages.

Loss of laziness is another of the well-known problems that can arise in attempts to combine exception handling with applicative languages. The most common solutions have been to isolate the loss to particular constructs, or to limit the propagation of exceptions, thereby isolating the loss to particular sections of a program, neither of which entirely supports referential transparency.

Govindarajan's work follows a different strategy, namely defining semantics for the nonstrict *if* and the nonstrict list operators that support both laziness and referential transparency [19]. Our approach extends this strategy by showing how it can be used for *and* and *or*, which is key to supporting laziness while still preserving referential transparency in the presence of multiple exceptions.

The invariant that must be maintained in this strategy is that, for nonstrict operators, if a subexpression in an expression does not need to be evaluated to produce the expression's answer, then that subexpression should be ignored. Hence, even if an unneeded subexpression coincidentally does get evaluated, perhaps because of some particular implementation mechanism in the evaluation engine, the answer should be the same as if it had not been evaluated.

Forms/3 is a lazy language. Its three nonstrict operators are *if*, *and*, and *or*. Tables 2, 3, and 4 define semantics for these operators that maintain referential transparency while still preserving laziness. Our definition of *if* in Table 2 is similar to that in Govindarajan's work, but in contrast to his, our *and* and *or* are nonstrict. Implementing the strategy for these, the built-in nonstrict operators, allows more complex expressions constructed using the built-ins to also retain their laziness.

For example, consider the following two cell formulas:

A: true or $(x > y)$
B: $(x > y)$ *or true*

By the semantics defined in Table 4, in Forms/3 the result of A and the result of B are both true, regardless of whether $(x > y)$ is ever executed and regardless of whether it signals an exception.

Govindarajan's work also defines semantics that explicitly cover exceptions for nonstrict list operators such as the equivalents of *head* and *tail*. Most spreadsheet languages have grids (matrices) instead of lists and do not have

3. The term *nonstrict* means that not all the arguments will necessarily be evaluated. For example, *if* is nonstrict because it will never require all three of its arguments. For example, if the first (the predicate argument) is true it will evaluate the second (the then-expression) but not the third (the else-expression).

explicit accessing operators such as *head* and *tail*. However, functionality similar to that of such operators associated with lazy lists is present if references to groups of grid elements are supported, even when no explicit operators are involved. For example, the functionality of *head* can be achieved by a reference to the first cell in a grid, which is a capability present in all spreadsheet languages, and the functionality of *tail* can be achieved by a reference to a group of grid cells, which is a capability found in some spreadsheet languages including Forms/3.

Surprisingly, although *head* and *tail* require special semantics to maintain laziness in the presence of exceptions in Govindarajan's work, this is not necessary in spreadsheet languages. To see this, consider grid accessing capabilities as they exist in Forms/3. In Forms/3, an entire grid can have a formula, which can access another entire grid of cells. For example, the formula for some grid M might be a reference to some grid N. This is simply treated as a shortcut for individual references between the corresponding cells of the two grids, i.e., each M component cell M_i's formula being a reference to N component cell N_i. Since individual cells are evaluated lazily, no problems arise with laziness: Just as with any cell in Forms/3, if M_i is visible on the screen, it generates demands for N_i and otherwise it does not. Treating grid accesses in this way handles all the exception cases of Govindarajan's list accessor semantics. That is, if for some reason M_i cannot validly reference N_i—regardless of whether it is because N_i does not exist (due to an exception arising from grid N's formula), or because N_i's value is an exception—then M_i's value will be an exception.

4.5 Issue 5: Missing Exceptions

The above solution handles most of the potential problems related to laziness in this paradigm, but one still remains. Suppose there is an exception that, even though it is not tied to any on-screen cell, still needs to be reported. Such cases would be unlikely in traditional business-oriented spreadsheets, but could arise in less traditional applications. For example, consider an application such as a home security system. In such an application, the on-screen spreadsheet might consist of security information, but unusual situations not directly tied to the main status display may also need to be monitored in off-screen spreadsheets, such as loss of security data, due perhaps to a communication interruption. Unless the cell whose formula checks for this situation is in the dataflow path of an on-screen cell, in a lazy language the cell would not be demanded and, hence, the exception would not be discovered. In Forms/3, a spreadsheet's creator must avoid this problem by making sure cells like this have some tie to output, such as by including them in the dataflow path of some on-screen cell. A spreadsheet language could further facilitate this by allowing a spreadsheet's creator to cause specific cells to be monitored eagerly, such as by allowing them to be placed on a distinguished "always active" spreadsheet that is always considered to be on the screen.

4.6 Issue 6: Liveness Itself Can Generate Exceptions

The fact that spreadsheet languages are fully live introduces an interesting situation. In a live language, whether lazy or eager, everything on the screen is needed for output. This is different from traditional languages, in which the programmer has control of which variables will be output and which will not. As a result, when the spreadsheet is being developed, extra exceptions can occur simply because *all* the program's on-screen cells, even those producing only intermediate values, are being calculated. This can lead to exceptions that would not be signaled in traditional languages.

For example, some spreadsheet languages have a built-in factorial operator, although Forms/3 does not. For languages having such an operator, a formula for some cell *answer* such as the following is possible:

if (x > 1) then fact (x) else 1

If x is a positive number, this will not signal any exceptions, regardless of whether the spreadsheet language being used is lazy or eager (with short circuit evaluation). But by the principle of referential transparency, it should be possible to remove the factorial subexpression to an intermediate cell z. Now z's formula would be:

fact (x)

and *answer*'s formula would be:

if $(x > 1)$ *then* z *else 1*

If z is on the screen, liveness will force it to be executed regardless of what x's value is.

One potential solution might seem to be for the system to apply techniques from program optimization research, reasoning about *if*-expressions to find entire groups of cells to consider as an atomic unit; however such an approach would violate the value rule for cell z. Fortunately, such measures are not necessary; the error value model can deal easily with this situation, provided that the usual spreadsheet convention is followed of built-in operators returning an error value (instead of terminating the system or locking up the system) whenever something goes awry in the operator's execution. For example, in Excel, the above results in the error value $\#NUM!$ for cell z when cell x is too large or too small. Because the result of the errant calculation is simply a value, not a change in control flow, any spurious exception that would not have arisen had an *if* expression not been distributed over multiple cells will simply be filtered out by the *if*. For example, both of the versions of *answer*'s formula above result in value one when x is negative, regardless of whether the *then* expression was embedded in a subformula or was a reference to on-screen cell z.

This straightforward way in which the error value model deals with irrelevant exceptions signaled by built-in operators can also be applied to problems of nontermination in spreadsheets. There are fewer opportunities for nontermination errors in spreadsheet languages than in traditional languages. Nontermination errors in programming languages can be subcategorized into nontermination that is due to cycles versus acyclic nontermination. Cycle-related nontermination in other languages that stems from

control flow looping does not arise in spreadsheet languages because there is no support for explicit control flow constructs, but cycle-related nontermination can arise from cyclical data dependencies in spreadsheet languages. To deal with this situation, some spreadsheet languages treat dependency cycles as constraint problems and attempt to solve them in a constant number of iterative tries, signaling (resulting in) an exception if the constant is exceeded. Forms/3 simply detects dependency cycles at formula entry time and signals (results in) a **formula-contains-cycles** exception for the cell. Thus, any cell referring directly or transitively to this cell will also result in an exception, but cells not referring to a cell in the cycle terminate normally.

Experimenting with the error value and nontermination in Forms/3, we have also used the error value model to deal with acyclic nontermination problems. Acyclic nontermination can arise when features available to user formulas can generate dynamic creation of too many additional things to do, seen in spreadsheet languages as too many additional data values to compute. Examples of such features include dynamically-sized grids (grid size determined by formula), dynamic creation of statically-specified animations (number of frames determined by formula), and dynamically-created recursive spreadsheet instances; in fact, these example features all exist in Forms/3. In Forms/3, when the underlying implementation runs out of space for these additional values, it simply signals an exception (returns an error value) for the cell being computed, such as **stack-overflow**. (Even if aggressive space reclaiming mechanisms were used in a manner that prevented space utilization from actually being physically exhausted, a counter could be used to measure the number of "virtual" cells computed so far, with an exception raised when some limit is reached.) These devices provide the benefit of protecting working portions of the spreadsheet from nontermination errors in other portions of the spreadsheet.

An interesting aspect of these situations is that, though the error value model handles them easily, other approaches to exception handling would run into difficulty with them in the context of spreadsheet languages. For example, an approach in which a handler was set up to detect *any* exception within a section of the spreadsheet, perhaps modeled after the scoped catch mechanism used by Java, Lisp, and Haskell, would branch to handle the irrelevant exceptions and/or potential nonterminations instead of allowing the rest of the spreadsheet not dependent on these exceptions to proceed normally. Of course, a spreadsheet creator could circumvent some such difficulties, such as by carefully avoiding distribution of formulas' intermediate subexpressions across separate cells to prevent extraneous nontermination errors, but this "solution" would be fragile, relying upon a spreadsheet creator's discipline and understanding of this subtlety.

5 Exception Composition and Abstraction

In programming languages, one technique often employed to enhance expressive power is abstraction. Abstraction allows composition of related information into a single package, thereby providing programmers the ability to abstract low-level details away into higher-level concepts and to use these abstractions multiple times. In Forms/3, abstraction of exceptions is possible using the ordinary operators and, for more sophisticated users, via language support for data abstraction.

5.1 Exception Composition through Ordinary Operators

Since Forms/3 treats all instances of the error type just like any other value, any combination of values—errors or not—can be combined to identify an exception using ordinary operators such as *if-then-else*. For example, if the clock spreadsheet referenced the system's clock rather than user input, the spreadsheet creator might add an 8:10 alarm using an *alarm* cell:

if (hourHand = 8) and (minuteHand = 10)
then true else false

Other cells in the spreadsheet could then refer to this cell in their own formulas (e.g., "*if alarm then ...*"). Such uses of *if-then-else* can involve arbitrarily complex combinations and can result in values of any type. This way of composing low-level details into higher-level exceptions is almost invisible, since it uses only ordinary operators and works with any kind of exceptions, whether of type *error* or not.

This composition mechanism is another example of the implications of the liveness property of spreadsheet languages. Due to this property, all operators define constraints, and the system guarantees maintenance of all of these constraints throughout the user's formula edits and display manipulations. Hence, any operator can compose exceptions in this way, not just the *if-then-else* operator. For example, the above formula for the *alarm* cell could have been written instead using simply the *and* operator:

(hourHand = 8) and (minuteHand = 10)

5.2 User-Defined Exceptions through Data Abstraction

For some situations, what is needed is the ability to define new kinds of exceptions that are still of type *error*. This would allow the spreadsheet creator to differentiate between exceptions that are errors versus those that are not, even in the case of user-defined exceptions. Forms/3 provides this capability through its support for abstract data types [4], [5].

Forms/3 supports abstract data types as follows: Types are defined on type definition spreadsheets. Attributes of a type are defined by formulas in cells and an instance of a type is the value of an ordinary cell which can be referenced just like any other cell. Built-in types are provided in the language implementation but are otherwise identical to user-defined types. For example, type *error* is actually a built-in abstract data type that is described by the spreadsheet in Fig. 6. The value of cell *newError* is determined by cells defining its error message and error type (bottom half of the spreadsheet) and the public attributes of any error can be accessed by changing the formula of *someError* to refer to the erroneous value and by then referencing cells *message?* and/or *errorType?* (top half of the spreadsheet).

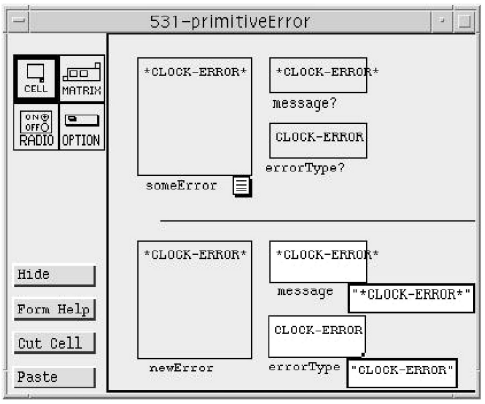


Fig. 6. Type *error*'s attributes can be set and queried using copies of this spreadsheet. (That this particular one is a copy can be seen by the fact that most of its background is gray; the white cells are the ones for which the spreadsheet creator has provided new formulas for this copy's cells.) Here, the spreadsheet creator has defined cell *errorType*'s formula to be the string " CLOCK-ERROR" and cell *message*'s formula to be the string "*CLOCK-ERROR*."

There is an apparent similarity between some programming environments' "property sheets," which allow maintenance of properties of objects and the spreadsheet in Fig. 6, but this similarity does not go beyond the surface. One difference is that, because the sheet in Fig. 6 is a spreadsheet, its cells can have formulas that specify arbitrarily complex relationships, not just constant values as in property sheets. Another difference is information hiding: In Forms/3 it is possible for the spreadsheet creator to make cells visually (and logically) invisible, the information-hiding equivalent of private methods and fields in other languages.

Each instance of a type is defined on its own copy of the spreadsheet for that type, which can be displayed upon demand. A spreadsheet creator can make new copies by pushing a *Copy* button; the system can also automatically make new copies as needed behind the scenes, as will be explained in Section 5.3.

Thus, in signaling an exception via an instance of type *error*, although the spreadsheet creator may use the textual *error* operator as in the earlier examples (*if (minuteHand > 60) then error else...*), this operator is actually a shortcut for a reference to cell *newError* on a copy of the built-in *primitiveError* spreadsheet that defines type *error*. This spreadsheet allows spreadsheet creators to create instances of the error type to define their own kinds of errors, as in Fig. 6.

5.3 Replacement Value Exception Handling via the Error Value Model

While the variation of the analog clock that references the system clock is a real world software application, it represents only the class of software in which a single spreadsheet stands alone. To support more reusable spreadsheets, such as libraries of spreadsheets that can be linked with a variety of present and future spreadsheets, an approach that supports information hiding and structured communication between the callee and caller is required. This was the basic point made by Yemini and Berry when they first introduced the replacement value model of exception handling [68].

In this section, we show how combining the error value model with Forms/3's abstraction mechanisms can achieve the functionality of the replacement value model described in Section 2 (with the exception of the propagation policy, as was discussed in Section 4). Using the error value model to achieve replacement value functionality has two advantages over other possible ways to achieve replacement value functionality. First, it maintains the compatibility featured by the error value model with the spreadsheet evaluation model. Second, it provides full featured exception handling suitable for libraries of reusable spreadsheet "modules" without encountering the problems that have arisen in more control-flow-oriented approaches to replacement value exception handling.

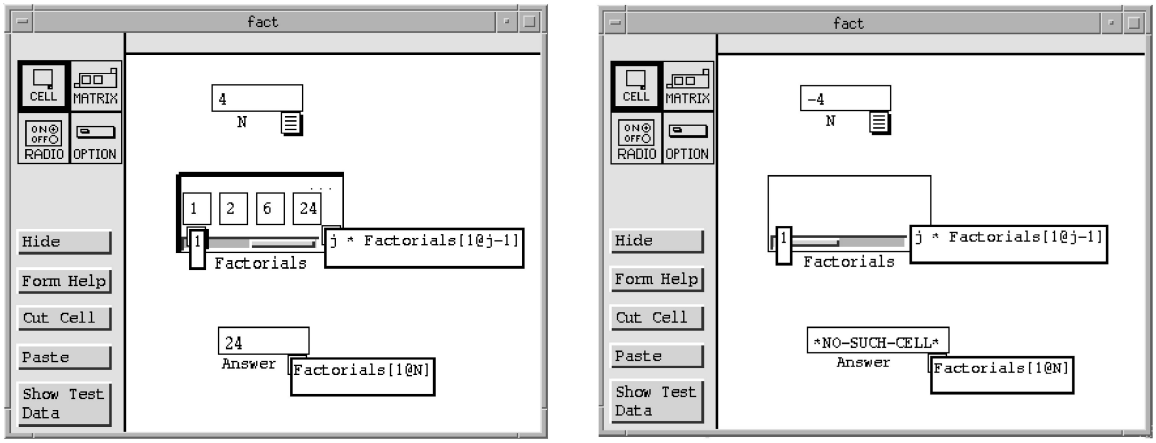
5.3.1 Abstraction in Forms/3

Abstraction is needed to support replacement value exception handling. In Forms/3, just as a spreadsheet can be used to define new types, a spreadsheet also can be used as a unit of procedural abstraction. A spreadsheet R can "call" another spreadsheet S by referring to a cell on S. (This is similar to the linked spreadsheets found in commercial spreadsheet systems.) To set up multiple such "calls" to a single spreadsheet S in Forms/3, the spreadsheet creator makes copies of spreadsheet S and changes some of the cells' formulas on these copies (for parameter passing functionality). For example, in Fig. 7, the cell whose formula would be changed for parameter passing functionality would be N. Forms/3 allows a spreadsheet creator to "lock in" the formulas for cells that should not be changed by making the formula tab invisible. This provides the spreadsheet creator a way to enforce the desired parameter protocol and to protect the spreadsheet's nonparameter cells from modification.

As the factorial example demonstrates, this approach provides functionality similar to both a (first-order) function and an instance of that function (i.e., an activation record). However, at first glance, the approach does not seem to afford as much generality in expressiveness as is usual with approaches to procedural abstraction in programming languages. The formulas for the cells in the example all refer to values that the spreadsheet creator explicitly instantiated, either by entering them explicitly via constant formulas, or by referring to cells on forms (activation records) that he or she manually created. In contrast to this, conventional first-order functions' parameters can automatically generate the needed activation records at runtime.

The Forms/3 solution to providing as much generality as is present with conventional first-order functions is though automatically generalizing formulas through deductive reasoning. The system automatically deduces the relationships specified in this concrete way [67], allowing additional invocations to be generated automatically by the system as needed.

An approach to generalization in programming languages can be either explicit or implicit. In an explicit approach, the user provides the generalized interpretation



(a) (b)

Fig. 7. (a) A Forms/3 spreadsheet to calculate N factorial. Cell N is intended for input and the fact that its formula tab is showing is intended to communicate this fact to users. (N's formula is the same as its value, the constant 4). Grid *Factorials* has two parts: The first, which has only the first cell in it, has the formula 1; the second contains all remaining cells and has the formula shown at the right. (b) The error value model's default behavior is resume semantics. Hence, without explicit exception handling code, an unusable input for N causes *Answer* to return an error value.

explicitly. Forms/3 follows an implicit approach, deriving the generalized version automatically. If an implicit approach for generalization employs inference,[4] there is a possibility of guessing wrong. Still, the probability of doing so is often reasonable in certain domains, such as user interface generation, in which the number of possibilities are relatively small and a number of domain-specific languages have successfully employed this technique. (See [12] for several examples.) However, this kind of inference has not been viable in general purpose languages, because the probability of guessing wrong is too high.

Fortunately, in spreadsheet formulas, the operators are already fully general; only the operands must be generalized. Further, there is enough information present in the dataflow graph of a spreadsheet's relationships to allow generalized operands to be implicitly[5] deduced. In fact, the operand is already partly general: The cell part of the operand as specified by the spreadsheet creator is either already fully general (as in *Answer*'s formula in Fig. 7) or is trivially generalizable via row/column relationships in a grid (as in *Factorials*). The only aspect of an operand that needs further deduction is a specification of how to generate an appropriate copy of a spreadsheet when needed.

Suppose S is a spreadsheet, S_i is a copy of S instantiated directly by a spreadsheet creator pressing the *Copy* button and $DefSet_i$ is a set of elements of format "$X.\text{formula} = \phi$," where each X is a cell on S_i whose formula has been edited to be ϕ. (ϕ can be any spreadsheet formula, such as a reference to some other cell, not just an example value.) It is possible for the system to abstractly specify copy S_i by noticing how its cells' formulas differ from those in S:

$$S_i = S \text{ modified by } DefSet_i.$$

In Forms/3, the system derives generalized formulas by replacing all references to specific copies of S with abstract descriptions such as the one above. For most cases, these descriptions are sufficient for the system to generate new copies automatically at runtime if needed. For example, if there is another spreadsheet *Commissions* containing a grid, each element of which needs to use *Factorials* in its calculation, the spreadsheet creator would enter the formula of one of the grid cells in terms of the relationship between that cell and a copy of *Factorials* and the resulting generalized formula would contain enough information to automatically generate the copies needed for the other cells in the grid. The cases in which this mechanism is not sufficient involve nontraditional call-return relationships, but even these can be handled by further analysis of the dataflow graphs involved. (See [67] for details.)

With this approach to abstraction, the expressive power of the error value model in spreadsheets increases over that presented in Section 3.2.3. In that section we pointed out that, without abstraction, it is necessary to provide exception handling formulas that refer to each *use* of a "function:" in essence a handler for every instance of an exception. In contrast to this, when abstraction is added, then exception handling can be provided with the *definition* of a function instead. In spreadsheet terms, a function definition is a spreadsheet (S) and uses of the function are references to a cell on S or on any copy of S. Hence, all instances of an exception that arise when referring to any copy of S will be handled by the handling formulas on S. This increases the expressive power of the approach and is also key to the ability to provide replacement value functionality using the error value model.

4. In much of the artificial intelligence literature, the term *inference* includes both sound reasoning techniques such as deduction and techniques employing guesswork. However, in literature about demonstrational programming languages, the term is normally used to mean only reasoning techniques employing guesswork. In this paper, we follow this latter convention.

5. Commercial spreadsheet languages are partially explicit; the user must enter a special character (often a "$") with a cell reference to make it an "absolute" reference. Further, since commercial spreadsheet languages are locked into a grid, generalization can be based solely on spatial relationships. Nongrid spreadsheet languages have not replaced the grid model with any alternative well-defined model of spatial relationships and, thus, must base generalization logic solely on logical relationships.

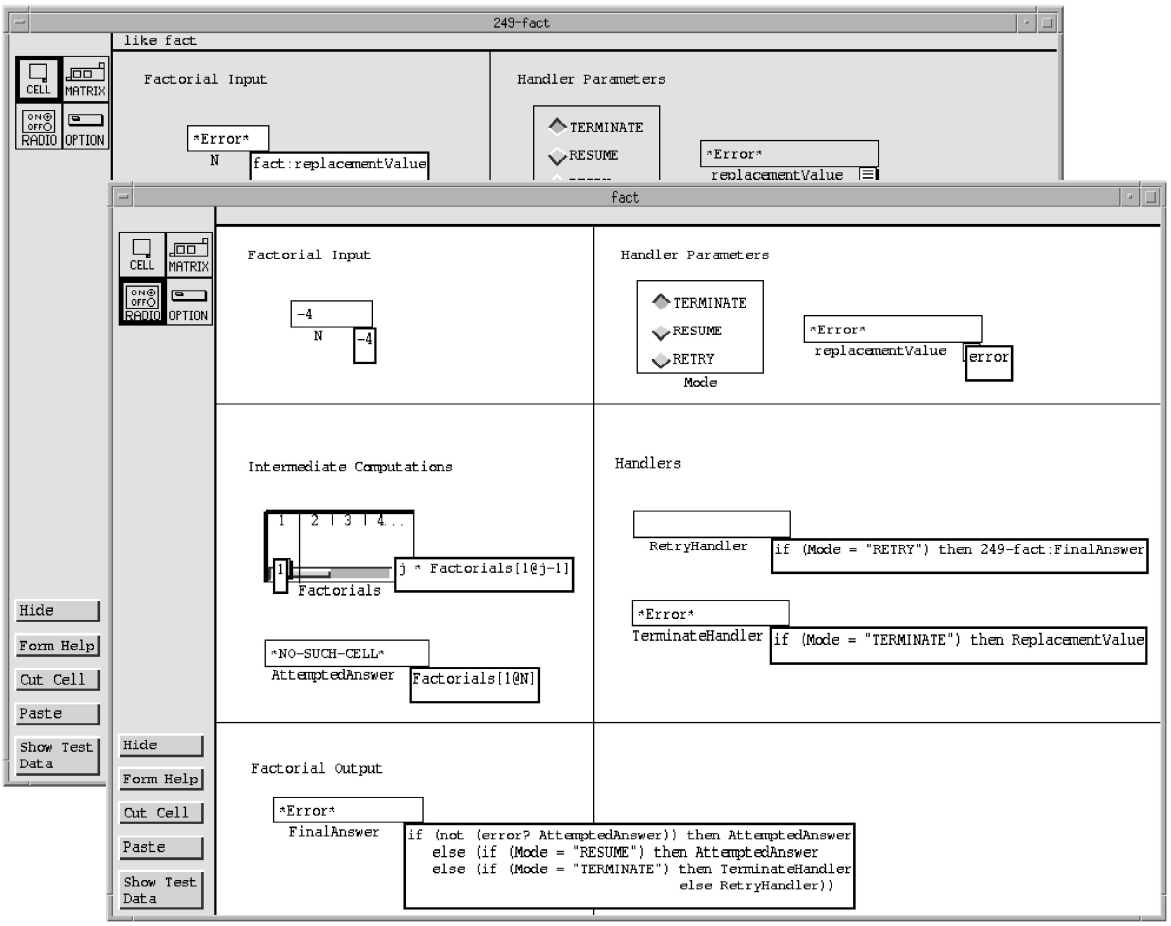


Fig. 8. The cells in the input, output, and parameter areas (top and midleft) are the interface to this routine. Replacement value exception handling (right half) is accomplished with formulas containing ordinary operators. (The lines and labels are simply borderless, nameless cells with formulas that evaluate to lines and strings.) Cell *RetryHandler* contains the result of a recursive call to a copy of *fact* in which *N*'s formula is a reference to *fact*'s *replacementValue*.

5.3.2 The Replacement Value Model in a Library Spreadsheet "Module"

Fig. 7's factorial spreadsheet includes only default error value exception handling. To improve this spreadsheet's functionality as a general purpose library module, the spreadsheet creator will add replacement value exception handling.

In Fig. 8, the spreadsheet creator has added cells to provide replacement value exception handling for *fact*. (Note that the formulas contain only the same ordinary operators as in the error value model examples.) In replacement value exception handling, handlers are defined inside the call*ee*, as the figure shows; this promotes cohesion. The call*er* sets up the parameters that dictate the handlers' behavior; this is because the caller (the application) knows more than the callee (the library routine) about the significance of the exception to the application. Because the handling cells have been placed on the *fact* spreadsheet, these handlers will deal with exceptions arising from *any* invocation of *fact*, which is, in our view, a necessary improvement from the individual-instance power of handling that was shown in earlier examples.

Cell *replacementValue* contains a value to be used if an exception is signaled and cell *Mode* specifies the desired handling mode. Future spreadsheet creators who make use of the factorial library routine will copy the *fact* spreadsheet, will provide a formula for input cell N, will provide a formula for the *replacementValue* cell and will choose an exception handling mode by pushing one of the radio buttons. (A radio button group in Forms/3 is a robust shortcut for a cell whose formula is intended to be one of an enumerated set of constants.)

Except for the above parameter-like cells and the output cell *Answer*, all the other cells on the spreadsheet will eventually be hidden; they are internal and, thus, of concern only to the creator of the factorial library routine. The spreadsheet creator did not choose to provide an explicit handler for resume mode handling in this example, because the underlying error value model automatically handles, using resume semantics, any exceptions not covered by the explicit handlers.

5.3.3 Exception Handling Modes Supported

In programming language literature, an exception handler's behavior after it takes corrective action is typically categorized into one of five modes: terminate execution, resume execution, retry execution, propagate the exception, or transfer control to a new location. Fig. 8 demonstrated the error value model's ability to support the first three of these. In regarding the fourth, propagation of exceptions has already been discussed at length. The fifth, transfer of control, is not possible in the error value model.

TABLE 5
The Behavior of Library Spreadsheet *fact* under Different Exception Handling Modes

Handler parameters	Inputs and results
Terminate, replacement value formula of *error*	N = -4 *Answer* = **Error**
Resume	N = -4 *Answer* = **No-Such-Cell**
Retry, replacement value of *round (abs N)*	N = -4 *Answer* = results of second invocation below
	Second invocation with N = 4 *Answer* = 24

The behavior of terminate handling under the replacement value model is that, if an exception is signaled, the replacement value is used as the value of the *final* output cell(s) on the spreadsheet, as illustrated by the second *else* in *FinalAnswer*'s formula. Resume mode, as we have already pointed out, is the default for the error value model; when coupled with a replacement value in the replacement value model, a cell's formula could refer to the replacement value if an exception arises instead of to an *intermediate* cell (the cell that caused the exception). The behavior of retry handling under the replacement value model is that, if an exception is signaled, the replacement value is referenced by the *initial* input cell(s) on another invocation of the spreadsheet, as illustrated by the reference in *248-fact*'s N to *fact*'s *replacementValue*. Table 5 shows example behavior of the factorial spreadsheet under these three exception handling modes.

6 Functionality Comparison with Modern Practical Languages

Although we have presented the exception handling features of our approach primarily in the context of those described in the research literature, it is also instructive to compare the approach with modern languages that are actually in wide use. In this section, we compare the Forms/3 exception handling approach with approaches found in three programming languages that are in use for building software used in the real world. The three we discuss are Excel, because it shows the way exception handling exists in a widely used commercial spreadsheet system; Java, because its approach to exception handling is thought by many to be the most advanced in wide use; and Haskell, because it is one of the few (nonspreadsheet) applicative languages in practical use whose approach to exception handling does not interfere with referential transparency. Our purpose is to focus on the differences in functionality, not on the differences in mechanisms used to achieve the functionality.

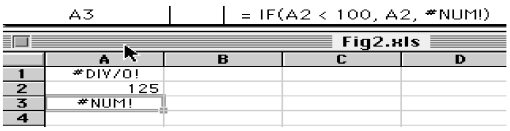


Fig. 9. Exceptions in Excel.

6.1 Comparison with Excel Exception Handling

Excel [33] is representative of conventional commercial spreadsheet systems' error value exception handling, and is in practical use by a huge number of users. In Excel, if a formula includes a reference to a cell that contains an error value, that formula also produces an error value (except for the operators *iserr*, *iserror*, or *isna*, which are used for exception handling). For example, in Fig. 9, the formula for cell *A1* is $5/0$, which evaluates to $\#DIV/0!$. The spreadsheet creator can also explicitly signal an instance of these error types by including it in a cell's formula. For example, the formula for cell *A3* in Fig. 9 is:

if (A2 < 100, A2, #NUM!)[6]

Excel's approach to exception handling follows the same basic model as that in Forms/3. Although there are several subtle differences (discussed in our solutions to the issues in Section 4), the largest difference between Forms/3 and Excel exception handling functionality is tied to Forms/3's support for abstraction. Using this mechanism, Forms/3 is able to provide a customizable error type (using the *primitiveError* spreadsheet). This provides the ability to create user-defined exceptions. It also can be used to define exception handling at the "function" granularity, as Fig. 8 showed for "function" *fact* and at the type granularity via the same technique, namely including exception handling formulas on the defining spreadsheet for the type. Since Excel has no abstraction capabilities, features such as user-defined exceptions, function-level handlers, and type-level handlers are not possible in Excel.

6.2 Comparison with Java Exception Handling

Java [18] is the first widely used object-oriented programming language in which the exception handling constructs themselves are in wide use. This is due at least in part to the fact that in Java, use of exception handling is enforced;[7] the Java compiler requires that if the code within a method can signal an exception, then the method must either provide handlers for that exception, or it must include the exceptions in the method declaration (although

6. The syntax of Excel's *if* function is *if* (condition, value_if_true, value_if_false).

7. The C++ exception handling mechanism [28] is similar in many ways to the Java exception handling mechanism. However, in C++ built-in exceptions are not supported and there is no enforcement of the use of exception handling.

```
class Grumpy       extends Exception {}
class TooHot       extends Grumpy {}
class TooTired     extends Grumpy {}
class TooStressed  extends Grumpy {}
...
  try {
    if ( temp > 40) throw (new TooHot () );
    if ( sleep < 8 ) throw (new TooTired () );
    x = x / y;                          // could signal a division by zero
    …
  } catch (TooHot g) {
    {System.out.println ("caught too hot!"); return;
  } catch (TooTired g) {
    {System.out.println ("caught too tired!"); return;
  } catch (Grumpy g) {
    {System.out.println ("caught some other flavor of grumpy"); return;
  } catch (Exception g) {
    {System.out.println ("caught some other kind of exception"); return;
  }
```

Fig. 10. Exception handling in Java (adapted from a program given in [18]).

there are some cases in which this requirement does not hold). This is a significant difference from Forms/3, which does not enforce use of exception handling.

Java exceptions are instances of class *Exception*. To create new, user-defined exception types, a Java programmer may create new subclasses under this class, as in Fig. 10. Java programmers can signal exceptions explicitly via a *throw* statement and can catch either user-defined or built-in exceptions using *catch*es, as shown in the figure. The type of exception signaled determines which handler is invoked by matching types of the signaled exception with those in the handlers. In this case, "matching" means that a handler with a formal parameter of type T matches a signaled exception of type T or any subclass of T. (Parameter passing is used to pass the exception object into the handler.) If there is no matching handler, the method or block terminates and the exception is propagated to the caller and then to *its* caller and so on. If no handler is ever found in the call tree, the exception is printed and the program terminates. Java's control-flow orientation is very different from the applicative model followed by Forms/3, but we will ignore this issue as much as possible in considering exception handling functionality.

The number of exceptions one handler is intended to handle is different in Java than in Forms/3. A Java *catch* block for a subclass that was an exact match, such as the first *catch* in Fig. 10, handles all exceptions that are instances *of* a particular type. This is not possible in Forms/3, since it is not particularly compatible with liveness, as was discussed in Section 4. Java exception handling also leverages the power of inheritance, allowing a *catch* block such as the third one in the figure to handle multiple types of exceptions within an entire subtree of the inheritance hierarchy. In Forms/3, this also is not possible. However, in Forms/3 (and in Java), it is possible to include exception handlers as part of an abstract data type, which catch all exceptions that happen *to* a particular type.

Aside from the control-flow versus data orientation aspect then, the main differences in functionality between the Java approach and the Forms/3 approach are: 1) the number of exceptions handled by one handler is greater in Java than in Forms/3; 2) Forms/3 provides default handling of all exceptions based on resume semantics, whereas Java's default handling of exceptions is based on terminate semantics; and 3) exception handling is enforced in Java, whereas it is not in Forms/3.

6.3 Comparison with Haskell Exception Handling

Haskell [40] is a general purpose, "pure" functional programming language. Like other functional programming languages, it supports higher-order functions. It is representative of the newest functional languages in its support for static polymorphic typing and data abstraction. It has also received wide attention for maintaining "purity" while still incorporating practical features, such as support for GUI I/O, that allow it to be used for real world applications.

Haskell makes a distinction between errors and exceptions. In Haskell terminology, the term "exception" applies only to I/O errors, which can be caught and handled. Other kinds of exceptions are termed "errors" in Haskell and cannot be handled. Exceptions are related to the IO monad,[8] which is an abstract data type representing I/O actions. The built-in I/O exceptions are of type *IOError*, and built-in exceptions can be signaled only by system primitives, not by user written code.

The *try/catch* notation of Java is combined in Haskell's *catch* function. The *catch* function associates an exception handler with an action or set of actions to be tried. The type signature of the *catch* function is:

8. Monads are a relatively recent advance in functional programming languages, and allow sequencing and state modification to be performed in controlled sections of the program without destroying referential transparency [41]. For example, one operation on IO actions is "sequentially order," which is used to solve the long-standing problem of convenient sequencing of I/O in functional languages.

catch :: IO a -> (IOError -> IO a) -> IO a

In Haskell notation, "::" means "is of type," and the types of the parameters are separated by arrows. To the right of the last arrow is the return type. Hence, if *catch* receives an I/O action (of type *IO a*) and an exception handler (of type *(IOError -> IO a)*), it will return an I/O action (of type *IO a*). The logic works as follows: If the incoming action succeeds, its result is returned without invoking the handler. If an I/O exception occurs, it is passed to the handler as a value of type *IOError* and the handler is then invoked, returning an action of type *IO a*.

There are several exception-oriented functions built into the language that a programmer can use in writing such a handler. Examples include *isEOFError*, to query whether the exception was an end-of-file and *fail*, which explicitly propagates the exception. In the following example (adapted from [24]), the function *getChar'* returns a character from the standard input. In case an exception is signaled by the function *getChar*, it is caught and passed on to the handler function *eofHandler*. If the exception was an end-of-file exception, the function returns a newline. Otherwise, the exception is propagated to the caller.

```
getChar'   = catch getChar eofHandler where
               eofHandler e = if isEOFError e   then return '\n'
                                                else fail e
```

It is also possible to build limited forms of user-defined exceptions via function *userError:*

userError :: String -> IOError

Although Haskell supports relationships among types similar to inheritance hierarchies, there is no hierarchy of exception types and, hence, no object-oriented type matching mechanism is used to select from a variety of handlers. However, Haskell's support for abstract data types allows exception handling to be provided for all (I/O-related) exceptions that happen *to* a particular type, as in Forms/3.

Haskell's monadic mechanism for exception handling is significantly different from the error value model. Still, ignoring the differences in mechanisms and focusing on the functionalities achieved by each, the main differences between the functionality of Haskell's exception handling and Forms/3's exception handling are 1) Haskell supports only I/O-related built-in exceptions and limited user-defined I/O-related exceptions, whereas Forms/3 supports all built-in exceptions and arbitrarily complex user-defined exceptions and 2) unlike Forms/3 (and like Java), Haskell's default handling of exceptions is based on terminate semantics: the program simply terminates with an error message.

7 Discussion

As the comparison in Section 6 shows, while the approach to exception handling presented here does not have as much expressive power as Java's, it exceeds the expressive power of Excel's approach and that of Haskell's approach. This is rather surprising, given the minimalism of the error value model.

The factorial example also demonstrates that the approach can be used in accordance with all of Yemini's and Berry's software engineering guidelines (listed in Section 2) except their information hiding recommendation. The most unconventional of the ways the guidelines are satisfied are in the support for parameterized handlers and in the compatibility with the language's scope, type, and verifiability characteristics. For parameterized handlers, a spreadsheet creator specifies the desired parameters on a copy of the "callee" spreadsheet and refers to that copy's final answer from the "caller" spreadsheet. Regarding scope, type, and verifiability compatibility, the approach does not impact these properties because it avoids changes to the evaluation model; hence, no new scope, type, or verifiability issues arise.

Correctness of exception handling code can be a problem in some languages, but the concreteness of the spreadsheet paradigm and the presence of liveness's immediate visual feedback may offer some help with this. For example, the use of widgets like radio buttons can promote robustness in parameterizing the handlers. When employing a spreadsheet with replacement value exception handling, the spreadsheet creator need not memorize or look up the codes for the different exception handling modes; he or she simply needs to push the appropriate button. Interactive visual characteristics and liveness can also add testing and documentation functionality. For example, in Forms/3, the creator of the library spreadsheet module can test exception handling behavior interactively while developing it. Later, creators of calling spreadsheets can learn about the module's exception handling behavior by trying it with sample inputs, without having to refer to separate documentation or the spreadsheet implementation details to understand it. These features are not a panacea, but they do provide some support for incremental testing and debugging. We have recently been working on ways to incorporate these features into an integrated assistance mechanism for testing and debugging in spreadsheet languages [6], [44], [49], [61], and have a series of empirical studies underway to understand how such a mechanism affects users' abilities to test and debug [8], [49], [63].

8 Conclusion

Programming languages that have been associated at least partially with end-user programming, such as spreadsheet languages, have been seriously under studied in terms of their programming language and software engineering properties. This is a serious omission because these languages are being used to create production software upon which real decisions are being based. Further, the spreadsheets created with these languages can be large and complex, such as for income tax preparation, and could benefit from mechanisms that help deal with complex applications. For these reasons, it is important to provide support for mechanisms such as exception handling that can aid both in the reliability of answers produced by these spreadsheets and in the spreadsheets' maintainability over time. However, if an exception handling mechanism is not compatible with the spreadsheet evaluation model, the associated learning curve could be an effective barrier to its

use, especially by the end users and informal programmers who create many of these spreadsheets.

To explore this issue, we have investigated the properties of error value exception handling in the spreadsheet paradigm. We have shown that the error value model is particularly well suited to the spreadsheet paradigm, due to characteristics such as the following:

- Under the error value model, spreadsheet creators require only ordinary language operators, such as *if-then-else*, for both exception handling and prevention;
- The error value model can fully preserve both referential transparency and laziness; and
- The error value model is entirely compatible with liveness.

Perhaps most important, despite the error value model's minimalism, when paired with a few straightforward abstraction capabilities, this simple model provides a surprisingly high degree of exception handling functionality, as was demonstrated both by the fact that it can support the replacement value model and by the results of comparing its functionality with the exception handling mechanisms in several modern programming languages.

ACKNOWLEDGMENTS

The authors would like to thank the members of the Oregon State University Visual Programming Language Research Group for their work on the Forms/3 implementation and for their feedback on the exception handling approach. Also, they would like to thank John Launchbury for helpful conversations about the issue of referential transparency and thank the anonymous reviewers for their insightful suggestions. Finally, they would like to acknowledge Maureen Chesire's contributions to some of the programming examples in this paper. This paper includes some material that appeared in the Proc. the 1996 IEEE Symp. Visual Languages [58].

This work was supported in part by Hewlett Packard, by Harlequin, by Pictorius, and by the National Science Foundation under CCR-9308649, CCR-9806821, and NSF Young Investigator Award CCR-9457473. A publicly available implementation of Forms/3 can be downloaded from http://www.cs.orst.edu/~burnett/Forms3/forms3.html.

REFERENCES

[1] H. Abelson, G. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs,* second ed., Cambridge, Mass.: MIT Press, 1996.

[2] M. Bretz and J. Ebert, "An Exception Handling Construct for Functional Languages," *Proc. Second European Symp. Programming,* Mar. 1988.

[3] M. Burnett, "Types and Type Inference in a Visual Programming Language," *Proc. IEEE Symp. Visual Languages,* pp. 238-243, Aug. 1993.

[4] M. Burnett and A. Ambler, " Interactive Visual Data Abstraction in a Declarative Visual Programming Language," *J. Visual Languages and Computing,* vol. 5, no. 1, pp. 29–60, Mar. 1994.

[5] M. Burnett and H. Gottfried, " Graphical Definitions: Expanding Spreadsheet Languages through Direct Manipulation and Gestures," *ACM Trans. Computer-Human Interaction,* vol. 5, no. 1, pp. 1–33, Mar. 1998.

[6] M. Burnett, A. Sheretov, and G. Rothermel, " Scaling Up a 'What You See is What You Test' Methodology to Spreadsheet Grids," *Proc. IEEE Symp. Visual Languages,* pp. 30–37, Sept. 1999.

[7] E. Chi, J. Riedl, P. Barry, and J. Konstan, "Principles for Information Visualization Spreadsheets," *IEEE Computer Graphics and Applications,* pp. 30–38, July/Aug. 1998.

[8] C. Cook, K. Rothermel, M. Burnett, T. Adams, G. Rothermel, A. Sheretov, F. Cort, and J. Reichwein, "Does Immediate Visual Feedback About Testing Aid Debugging in Spreadsheet Languages?," Technical Report 99-60-07, Oregon State Univ., 1999.

[9] P. Cox and T. Pietrzykowski, "Using a Pictorial Representation to Combine Dataflow and Object-Orientation in a Language Independent Programming Mechanism," *Proc. Int'l Computer Science Conf.,* pp. 695–704, 1988.

[10] P. Cox and T.J. Smedley, "Using Visual Programming to Extend the Power of Spreadsheet Computation," *Proc. Advanced Visual Interfaces,* pp. 153–161, June 1994.

[11] Q. Cui and J. Gannon, "Data-Oriented Exception Handling," *IEEE Trans. Software Eng.,* vol.18, no. 5, pp. 393–401, May 1992.

[12] *Watch What I Do: Programming by Demonstration,* A. Cypher ed., Cambridge, Mass.: MIT Press, 1993.

[13] R. Djang, M. Burnett, and R. Chen, "Static Type Inference for a First-Order Declarative Visual Programming Language with Inheritance," *J. Visual Languages and Computing,* Apr. 2000.

[14] C. Dony, "Exception Handling and Object-Oriented Programming: Towards a Synthesis," *Proc. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 322–330, Oct. 1990.

[15] J. Feo, D. Cann, and R. Oldehoeft, "A Report on the Sisal Language Project," Technical Report UCRL-102440, Rev. 1, Lawrence Livermore Nat. Lab., 1990.

[16] H. Giese, J. Graf, and G. Wirtz, "Seamless Visual Object-Oriented Behavior Modeling for Distributed Software Systems," *Proc. IEEE Symp. Visual Languages,* pp. 156–163, Sept. 1999.

[17] J. Goodenough, "Exception Handling: Issues and Proposed Notation," *Comm. ACM,* vol. 18, no. 12, pp. 683–696, 1975.

[18] J. Gosling, B. Joy, and G. Steele, "The Java Language Specification," Aug. 1996, http://java.sun.com:80/docs/books/jls/html/index.html.

[19] R. Govindarajan, "Exception Handlers in Functional Programming Languages," *IEEE Trans. Software Eng.,* vol. 19, no. 8, pp. 826–834, Aug. 1993.

[20] D. Harel, "Statecharts: A Visual Approach to Complex Systems," *Science of Computer Programming,* vol. 8, no. 3, pp. 231–274, 1987.

[21] R. Harper, A. Milner, and M. Tofte, "The Definition of Standard ML, Version 3," Internal Report ECS-LFCS-89-91, Dept. of Computer Science, Edinburgh Univ., Edinburgh, Scotland, 1989.

[22] N. Heger, A. Cypher, and D. Smith, "Cocoa at the Visual Programming Challenge 1997," *J. Visual Languages and Computing,* vol. 9, no. 2, pp. 151–169, Apr. 1998.

[23] P. Hudak et al., "Report on the Programming Language Haskell, a Non-Strict, Purely Functional Language, Version 1.2," *ACM SIGPLAN Notices,* vol. 27, no. 5, May 1992.

[24] P. Hudak, J. Peterson, and J. Fasel, "A Gentle Introduction to Haskell, Version 1.4," Mar. 1997, http://haskell.org/tutorial/index.html.

[25] S. Hudson, "User Interface Specification Using an Enhanced Spreadsheet Model," *ACM Trans. Graphics,* vol. 13, no. 3, pp. 209–239, July 1994.

[26] C. Hughes and J. Moshell, "Action Graphics: A Spreadsheet-Based Language for Animated Simulation," *Visual Languages and Applications,* T. Ichikawa, E. Jungert, R. Korfhage, eds., New York: Plenum Publishing, pp. 203–235, 1990.

[27] A. Kay, "Computer Software," *Scientific Am.,* pp. 53–59, Sept. 1984.

[28] A. Koenig and B. Stroustrup, "Exception Handling for C++," *J. Object-Oriented Programming,* pp. 16–33, July 1990.

[29] J. Leopold and A. Ambler, "Keyboardless Visual Programming Using Voice, Handwriting, and Gesture," *Proc. IEEE Symp. Visual Languages,* pp. 28–35, Sept. 1997.

[30] C. Lewis, "NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery," *Visual Programming Environments: Paradigms and Systems,* E. Glinert, ed., pp. 526–546, 1990.

[31] B. Liskov and A. Snyder, "Exception Handling in CLU," *IEEE Trans. Software Eng.,* vol. 5, no. 6, pp. 546–558, Nov. 1979.

[32] F. Ludolph, Y. Chow, D. Ingalls, S. Wallace, and K. Doyle, "The Fabrik Programming Environment," *Proc. IEEE Workshop on Visual Languages,* pp. 222–230, Oct. 1988.

[33] *Microsoft Excel 4.0 User's Guide 1 and Microsoft Excel 4.0 Function Reference.* Microsoft Corporation, 1992.
[34] R. Miller and A. Tripathi, "Issues with Exception Handling in Object-Oriented Systems," *Proc. European Conf. Object-Oriented Programming,* June 1997.
[35] J. Monin, "Exceptions Considered Harmless," *Science of Computer Programming,* vol. 26, nos. 1-3, pp. 179–196, May 1996.
[36] B. Myers, "Graphical Techniques in a Spreadsheet for Specifying User Interfaces," *Proc. ACM Conf. Human Factors in Computing Systems,* pp. 243–249, Apr./May 1991.
[37] R. Panko and R. Halverson, "Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks," *Proc. 29th Hawaii Int'l Conf. System Sciences,* Jan. 1996.
[38] R. Panko, "What We Know About Spreadsheet Errors," *J. End User Computing,* pp. 15–21, 1998.
[39] F. Paulo, P. Masiero, and M. Oliveira, "Hypercharts: Extended Statecharts to Support Hypermedia Specification," *IEEE Trans. Software Eng.,* vol. 25, no. 1, pp. 33–49, Jan./Feb. 1999.
[40] J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. Peyton-Jones, A. Reid, and P. Wadler, *Haskell 1.4: A Non-Strict, Purely Functional Language,* Apr. 1997, http://haskell.systemsz.cs.yale.edu/report/.
[41] S. Peyton-Jones and P. Wadler, "Imperative Functional Programming," *Proc. ACM Symp. Principles of Programming Languages,* pp. 71–84, Jan. 1993.
[42] J. Pfeiffer, "Altaira: A Rule-Based Visual Language for Small Mobile Robots," *J. Visual Languages and Computing,* vol. 9, no. 2, pp. 127–150, Apr. 1998.
[43] A. Reeves, D. Harrison, A. Sinclair, and P. Williamson, "How to Make a Lazy Functional Language Exceptional," *Proc. IEEE TENCON '89 in Functional Programming Languages: Theory & Applications,* pp. 179–185, Nov. 1989.
[44] J. Reichwein, G. Rothermel, and M. Burnett, "Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging," *Proc. Conf. Domain-Specific Languages,* pp. 25–38, Oct. 1999.
[45] A. Repenning, "Bending the Rules: Steps Toward Semantically Enriched Graphical Rewrite Rules," *Proc. IEEE Symp. Visual Languages,* pp. 226–233, Sept. 1995.
[46] A. Repenning and A. Ioannidou, "Behavior Processors: Layers between End-Users and Java Virtual Machines," *Proc. IEEE Symp. Visual Languages,* pp. 402–409, Sept. 1997.
[47] C. Risley and T. Smedley, "Visualization of Compile Time Errors in a Java Compatible Visual Language," *Proc. IEEE Symp. Visual Languages,* pp. 22–29, Sept. 1998.
[48] A. Romanovsky, J. Xu, and B. Randell, "Exception Handling in Object-Oriented Real-Time Distributed Systems," *Proc. IEEE Symp. Object-Oriented Real-Time Distriuted Computing,* Apr. 1998.
[49] G. Rothermel, L. Li, C. DuPuis, and M. Burnett, "What You See is What You Test: A Methodology for Testing Form-Based Visual Programs," *Proc. Int'l Conf. Software Eng.,* pp. 198–207, Apr. 1998.
[50] S. Sauer and G. Engels, "Extending UML for Modeling of Multimedia Applications," *Proc. IEEE Symp. Visual Languages* pp. 80–87, Sept. 1999.
[51] R. Sebesta, *Concepts of Programming Languages,* fourth ed., Reading, Mass.: Addison-Wesley, 1999.
[52] T. Shepard, S. Sibbald, and C. Wortley, "A Visual Software Process Language," *Comm. ACM,* vol. 35, no. 4, pp. 37–44, Apr. 1992.
[53] T. Smedley, P. Cox, and S. Byrne, "Expanding the Utility of Spreadsheets through the Integration of Visual Programming and User Interface Objects," *Proc. ACM Workshop Advanced Visual Interfaces,* pp. 148–155, May 1996.
[54] A. Sowmya and S. Ramesh, "Extending Statecharts with Temporal Logic," *IEEE Trans. Software Eng.,* vol. 24, no. 3, pp. 216–231, Mar. 1998.
[55] S. Tanimoto, "Towards a Theory of Progressive Operators for Live Visual Programming Environments," *Proc. IEEE Workshop Visual Languages,* pp. 80–85, Oct. 1990.
[56] C. Traynor and H. Goodell, "End User Programming/Informal Programming," *SIGCHI Bulletin,* Oct. 1999.
[57] J. Ullman, *Elements of ML Programming.* Prentice Hall, 1998.
[58] P. van Zee, M. Burnett, and M. Chesire, "Retire Superman: Handling Exceptions Seamlessly in a Declarative Visual Programming Language," *Proc. IEEE Symp. Visual Languages,* pp. 222–230, Sept. 1996.
[59] G. Viehstaedt and A. Ambler, "Visual Representation and Manipulation of Matrices," *J. Visual Languages and Computing,* vol. 3, no. 3, pp. 273–298, Sept. 1992.
[60] P. Wadler, "How to Replace Failure by a List of Successes: A Method for Exception Handling, Backtracking, and Pattern Matching in Lazy Functional Languages," *Functional Programming Languages and Computer Architecture,* pp. 113–128, Sept. 1985.
[61] R. Walpole and M. Burnett, "Supporting Reuse of Evolving Visual Code," *Proc. IEEE Symp. Visual Languages,* pp. 68–75, Sept. 1997.
[62] G. Wang and A. Ambler, "Solving Display-Based Problems," *Proc. IEEE Symp. Visual Languages,* pp. 122–129, Sept. 1996.
[63] E. Wilcox, J. Atwood, M. Burnett, J. Cadiz, and C. Cook, "Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems?," *Proc. ACM Conf. Human Factors in Computing Systems (CHI '97),* Mar. 1997.
[64] N. Wilde, "A WYSIWYC (What You See Is What You Compute) Spreadsheet," *Proc. IEEE Symp. Visual Languages,* pp. 72–76, Aug. 1993.
[65] N. Wilde and C. Lewis, "Spreadsheet-Based Interactive Graphics: From Prototype to Tool," *Proc. ACM Conf. Human Factors in Computing Systems,* pp. 153–159, Apr. 1990.
[66] J. Xu, A. Romanovsky, and B. Randell, "Coordinated Exception Handling in Distributed Object Systems: From Model to System Implementation," *Proc. Int'l Conf. Distributed Computing Systems,* May 1998.
[67] S. Yang and M. Burnett, "From Concrete Forms to Generalized Abstractions through Perspective-Oriented Analysis of Logical Relationships," *Proc. IEEE Symp. Visual Languages,* pp. 6–14, Oct. 1994.
[68] S. Yemini, D. Berry, "A Modular Verifiable Exception-Handling Mechanism," *ACM Trans. Programming Languages and Systems,* vol. 7, no. 2, pp. 213–243, Apr. 1985.

Margaret Burnett received the BA degree in mathematics from Miami University, Oxford, Ohio, the MS and the PhD degrees in computer science from the University of Kansas. She worked for several years for large and small companies before beginning her academic career as an assistant professor at Michigan Technological University. She is currently an associate professor in the Department of Computer Science at Oregon State University. Her research interests are in programming languages, visual programming languages, and how programming language and software engineering research can be applied to support end-user programming.

Dr. Burnett has been a recipient of the National Science Foundation's Young Investigator Award. She has served on the papers program committee for the ACM Conference on Human Factors in Computing Systems; on the program committee for the Visual End User Workshop, the IEEE Symposium on Visual Languages, and the ACM Conference on Programming Language Design and Implementation; on the steering committee for the IEEE Symposium on Visual Languages; and on the editorial board of the Journal of Visual Languages and Computing. She is a senior member of IEEE and a member of ACM and the IEEE Computer Society.

Anurag Agrawal no picture or bio available.

Pieter van Zee received the BSE degree in EE/CS from Princeton University. He worked for five years on image processing systems for Siemens A.G. in Munich, Germany, before earning the MS degree in computer science from Oregon State University. Mr. van Zee has worked for Hewlett-Packard Company since 1993 on 3D graphics toolkits, online information systems, and internet technologies. He is currently a senior architect for software and internet systems. Mr. van Zee's research interests are in visual programming languages, e-publishing, digital imaging, and consumer computing appliances and applications.