*In Proceedings of 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, Sept. 3-6, 1996 (to appear)*

# Retire Superman: Handling Exceptions Seamlessly in a Declarative Visual Programming Language

Pieter van Zee
Hewlett-Packard
piet@cv.hp.com

Margaret Burnett*
Oregon State University
burnett@cs.orst.edu

Maureen Chesire
Mentor Graphics Corporation
maureen@mentorg.com

## Abstract

*Exception handling is widely regarded as a necessity in programming languages today, and almost every programming language in current use supports some form of it. Unfortunately however, most approaches to exception handling involve constructs with unusual powers, and even deviations from the language's evaluation model. To avoid such devices in our declarative visual programming language, we have devised a full-featured approach to exception handling that fits seamlessly into languages that are declarative and visual. Using this approach allows designers of declarative visual programming languages to provide the expressive power previously available only through complex exception handling techniques.*

## 1. Introduction

The American folk hero Superman can be thought of as the personification of programming language mechanisms that have unusual powers. Superman bends steel in his bare hands, runs faster than a locomotive, leaps tall buildings in a single bound, and flies faster than the fastest jet planes. Superman uses these attributes to handle dangerous situations that arise in the city of Metropolis. However, Superman is somewhat unpredictable—sometimes he does not arrive to solve the problem. For example, he cannot solve multiple problems at once, and he cannot solve a problem that arises when he is working on solving another problem. Thus, because it is not possible to predict his problem-solving behavior consistently, the disaster response plans for the city of Metropolis do not rest upon a reliance on Superman.

Unfortunately, approaches to exception handling in most programming languages do require reliance on Superman-like mechanisms—mechanisms that are as different from the other constructs of these languages as Superman is from the other citizens of Metropolis. For example, in imperative languages, ordinary imperative constructs are only invoked via an explicitly defined sequence, but exception handling constructs ("on <exception>," "signal," etc.) allow sets of statements to be *implicitly* invoked. The addition of these new constructs and their deviation from the explicit order of evaluation increase the complexity for a programmer to reason about a program, because in order to do so, the programmer must constantly take into account these "Supermen" who may at any time wrest control from the program's explicit control flow. Such inconsistencies are not unique to exception handling in imperative languages; significant inconsistencies are also present in approaches to exception handling for functional languages.

We believe that exception handling is one of the features needed for scalability in visual programming languages (VPLs). We also believe that if VPLs are to succeed in improving the usability of languages with which people program, inconsistencies such as those found in traditional approaches to exception handling must be avoided. In this paper, we present a new approach to exception handling that, while still providing the features that are generally sought by researchers in the area of exception handling, fits seamlessly into commonly used declarative visual programming paradigms. The contributions of the approach are that (1) it is the first approach to provide full-featured exception handling in a VPL, (2) it does so without an accompanying loss of language simplicity, and (3) it demonstrates that the visual characteristics of a VPL can make a significant difference in the way exception handling can be supported in a programming language.

## 2. Background and Related Work

*Exception handling* refers to the mechanisms that support the detection, signaling, and handling of exceptions. Sebesta defines an *exception* as any unusual

*This work was supported in part by Hewlett-Packard and by the National Science Foundation under grant CCR-9308649 and an NSF Young Investigator Award.

event, erroneous or not, that is detectable either by hardware or software and that may require special processing [10]. An exception is *signaled* when it is detected. An *exception handler* is the special processing code that is executed as a result of an exception being signaled.

### 2.1 Two Models of Exception Handling

Most research on exception handling has been in imperative, object-oriented, and applicative (especially functional) languages. Since our approach is for declarative VPLs, we focus here on applicative languages. For an approach to exception handling to be usable by all applicative languages (and not be restricted to only functional languages), it cannot rely on higher-order functions. Thus, approaches such as continuation-passing style or monadic approaches, which require higher-order functions, cannot be used by applicative languages that do not support higher-order functions, such as most dataflow languages and spreadsheet languages.

The two models of exception handling upon which our approach builds are the error value model and the replacement value model. In the *error value model*, uniquely identifiable error values are used to indicate after the fact that an exception has occurred. Some applicative and functional languages follow this model (c.f. [5, 11]), and it can be approximated in imperative languages via distinguished values (e.g., procedure return values and status flags). Most commercial spreadsheets also follow the error value model.

The error value model is attractively simple, but it does not support all of the generally accepted principles of exception handling. For example, Goodenough's landmark paper on exception handling [6] points out that exceptions are not necessarily errors; 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*[1] as an approach that follows these guidelines [13]. 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 permissible because it supports information hiding by allowing abstraction of exception information.
- Data and procedural abstractions should be able to include exceptions in their definition. 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 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 [1], and forms the basis of much recent work in exception handling in the functional language community.

However, researchers have identified several problems that can arise when replacement value exception handling is introduced into functional and other applicative languages [1, 9, 14]. 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. Although this problem and other related problems can be addressed by changing the evaluation model to avoid the problems or by adopting specific semantic conventions to resolve them, doing so does not address the true cause of these difficulties, which lies in the fact that adding special-purpose constructs to applicative languages to support exception handling usually results in incompatibilities with the evaluation models of those languages.

### 2.2 Exception Handling in VPLs

To date, there has been little support for exception handling capabilities in VPLs. The few VPLs that do support exception handling provide only low-level support with no abstraction capabilities for exceptions. For example, Fabrik [7], a dataflow VPL, supports system-level errors only, under the error value model. In Fabrik, if a component cannot compute, the values on the output

[1] Yemeni and Berry name it the "replacement model". We use the phrase "replacement value model" in this paper because it emphasizes the nature of the approach.

pins are invalid and this invalidity is propagated to the connected input pins. Connections carrying invalid values appear as dashed lines.

Commercial spreadsheet programs, which share many characteristics of VPLs, also follow the error value model. For example, in Microsoft™ Excel® [8], when a primitive operation detects an exception, it returns one of seven possible error values. Operations are provided that test for error values and can discriminate among them and also generate them. Using these mechanisms, a programmer can detect when an exception has occurred and provide the desired exception handling.

The description of Prograph2 [4], a version of Prograph, includes the only detailed discussion on exception handling we have been able to locate in VPL literature. Prograph2 combines the dataflow, object-oriented, and imperative paradigms. Exception handling is provided through 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.

# 3. A New Approach to Exception Handling

In this paper, we combine the error value model and the replacement value model and show that, given suitable abstraction mechanisms, this combination can fit seamlessly with a declarative VPL. We had two goals: to support full-featured exception handling, and to do so seamlessly. To be more precise about what we mean by seamlessness, we state a *seamlessness constraint*:

> For an approach to fit seamlessly with a language, it must be implementable simply by *appending* (0 or more) new operators to the language's dictionary. No other changes to the language or its operators are allowed, and no deviations from the language's normal evaluation model to support the new operators are allowed.

## 3.1 A Brief Introduction to Forms/3

Forms/3 [2], the language in which we have prototyped our approach to exception handling, is a general purpose, declarative VPL. Its goal is to provide computational and expressive power in a language featuring a simple, concrete programming style with immediate feedback. Programming in Forms/3 follows the spreadsheet paradigm; the programmer uses direct manipulation to place cells on forms, and then defines a formula for each cell. Forms are the basic organizational units, and cells are the computational units. Because each cell's value is determined by its formula, a program's behavior is entirely determined by the cells' formulas. Forms/3 is *fully live*, which means that it automatically re-evaluates on-screen values whenever a formula is changed or new data arrives. As will be discussed later, liveness in a VPL provides both opportunities and problems for exception handling.

## 3.2 Error Value Exception Handling

The foundation of our approach lies in the error value model. The analog clock program in Figure 1 demonstrates this model of exception handling in Forms/3. The clock program takes two integers, representing the time of day in hours and minutes, and displays the corresponding analog clock. The x- and y-positions of the clock's hands are computed by cells *minutex*, *minutey*, *hourx*, and *houry*. Cell *theClock* references the results of the cells *minuteHand*, *hourHand*, *face* and *pivot* to assemble the clock components into one unit. The cell references were indicated 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.) 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 programmer has not provided any exception handling code in the formulas in Figure 1, the program defaults to the exception handling automatically provided by the system under the error value model. The error value exception handling model is implemented in Forms/3 using a distinguished *error* type. For example, if a character is entered as the formula for the *minute* cell instead of an integer, the operators invoked by formulas in the cells *minutex* and *minutey* will detect and signal exception conditions by returning values of type *error*.

### *3.2.1 If-then-else + declarative semantics + output = rules*

The example so far shows the system signaling exceptions by generating *error* values. (These values can also be generated explicitly by programmers via the *error* operator.) Programmers can capitalize upon the presence of error values by specifying their own exception handling "rules": ordinary if-then-else formulas defining calculations predicated on exceptions arising. For example, suppose we rename *theClock* to *goodClock*, add a *badClock* cell containing a sketch of a broken clock (drawn using an ordinary X-Windows bitmap editor), and create 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*.) Figure 2 shows the result of these three changes.

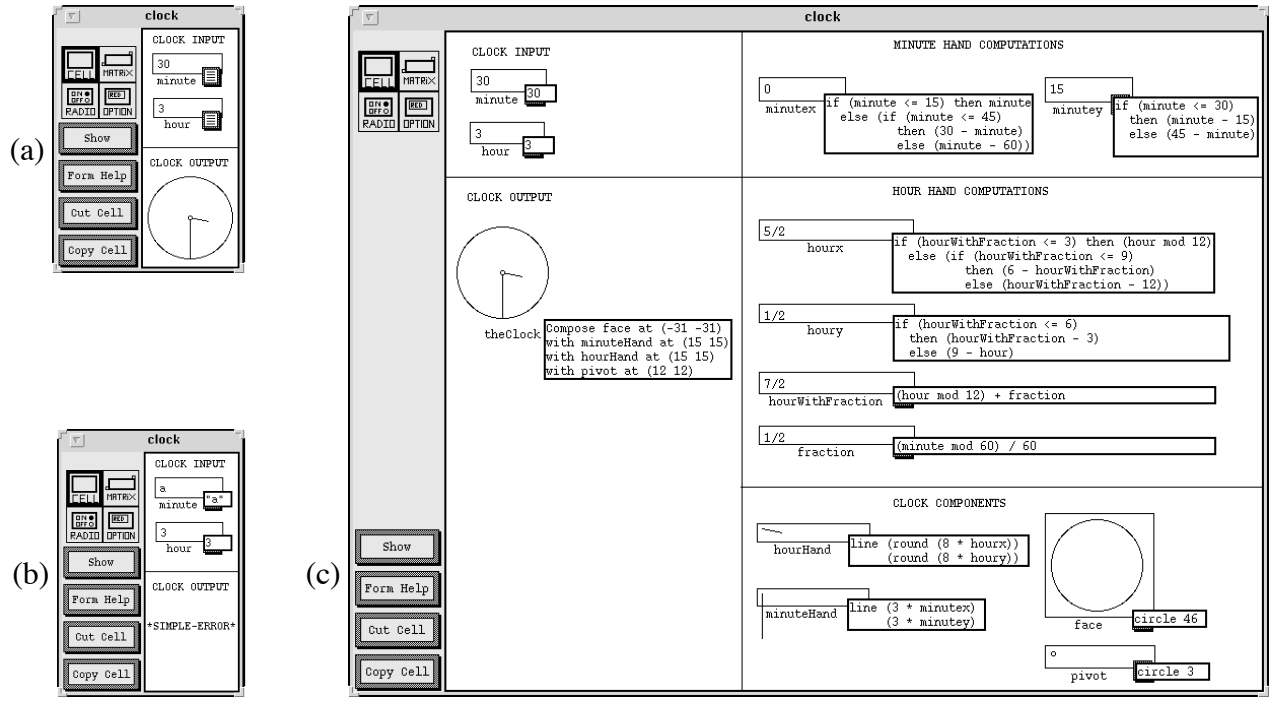


**Figure 1**: The user's view of the clock program does not include the formulas or the cells the programmer has chosen to hide. Here two user views are shown (a) with correct input, and (b) with erroneous input. The tabs indicate where users can enter input formulas. (c) The programmer's view: There is no exception handling code in this program; it simply defaults to the error value exception handling automatically provided by the system.

This example illustrates the key characteristic of declarative languages that can be exploited to allow seamless exception handling under the error value model: The ordinary if-then-else conditional construct in a declarative language, when paired with a demand for output, provides the same functionality as rule-based semantics. This is because (1) the declarative nature of the language says that the variables' definitions (in Forms/3, these are the cells' formulas) entirely define all the relationships in the program, and (2) wherever output is produced in such a language, the system must automatically maintain all values contributing to the output. This combination provides exactly the evaluation model needed for exception handling, because programmer-supplied exception handlers are the equivalent of rules that must be followed whenever the associated exceptions arise.

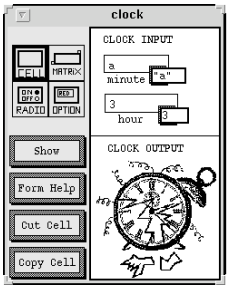


**Figure 2**: The user's view of a clock program for which the programmer has specified an exception handling "rule".

### *3.2.2 Exception Composition and Abstraction*

In programming languages, one technique often employed for scalability 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. In Forms/3, abstraction of exceptions is inherent in both the if-then-else and in our approach to data abstraction [2].

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 the ordinary if-then-else construct. For example, if the clock program referenced the system's clock rather than user input, we might add an 8:10 alarm using an *alarm* cell:

*if (hourHand = 8) and (minuteHand = 10)*
*then TRUE else FALSE*

Other cells in the program 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, not just Booleans. This way of composing low-level details into higher-level exceptions is almost invisible, since it

uses only the ordinary if-then-else construct, and works with any kind of exceptions, whether errors or not.

However, for some situations, what is needed is the ability to compose new kinds of exceptions that are still of type error. This would allow the programmer to differentiate between exceptions that are errors versus those that are not. Forms/3 provides this capability because it supports composition of abstract data types.

Like every type in our system, type *error* is an abstract data type. To signal an exception with an instance of this type, the Forms/3 programmer may use the *error* operator (*if minuteHand > 60 then error else...*). The *error* operator is actually a shortcut for a reference to cell *newError* on a copy of the primitive error form that defines type *error*. This form allows programmers to insert arbitrarily complex data into instances of the error type to define their own kinds of errors. See Figure 3.

### 3.3 Replacement Value Exception Handling

While the variation of the analog clock that references the system clock is a real-world software application, it represents only the small class of software in which a module is a standalone program. To support more reusable software, such as library routines to be used in a variety of present and future programs, 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.

In this section we show how the error value model and abstraction mechanisms, when combined with visual techniques, can achieve the functionality of the replacement value model. Achieving replacement value model functionality in this way has two advantages over traditional approaches to exception handling. First, it maintains the simplicity and seamlessness of the error value model. Second, it provides full-featured exception handling suitable for software such as library routines, without encountering the problems (given in Section 2) that arise in other approaches to replacement value exception handling.

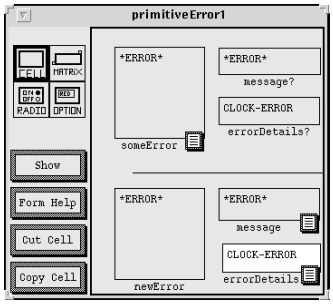


**Figure 3**: Type *error*'s attributes can be set and queried using copies of this form. Here the programmer has inserted a string into an error by defining cell *errorDetails'* formula to be the string "CLOCK-ERROR".

#### *3.3.1 The replacement value model in a library routine*

Figure 4 displays a factorial program that includes only default error value exception handling. The error value model is often sufficient for small standalone programs, but to improve this form's functionality as a general-purpose library routine, we will add replacement value exception handling.

Abstraction is a key element needed to support replacement value error handling. In Forms/3, a form is the unit of procedural abstraction. In Forms/3's concrete programming style, parameterized calls are defined by copying a form and providing formulas (the arguments) for cells that are modifiable (the formal parameters), thereby modeling the form's actual invocation. An example is cell *N* on form *Fact1* in the figure. The system automatically generalizes the relationships specified in this concrete way [12], allowing invocation of other calls such as the additional recursive calls in the factorial example.

In Figure 5, cells have been added that provide replacement value exception handling. (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.

The handler parameters are cells *replacementValue* and *Mode*. Cell *replacementValue* specifies the expression to be used as a substitute value if an exception is signaled. Future programmers who use the factorial library routine

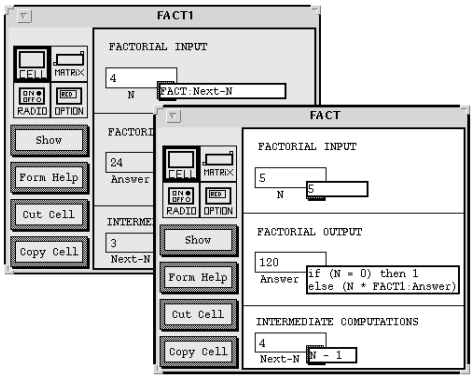


**Figure 4**: A recursive factorial program with default error value exception handling. An inappropriate value for *N* would cause *Answer* to return an error. Even a negative *N* would result in an error value, since stack overflow recovers and produces an error value.

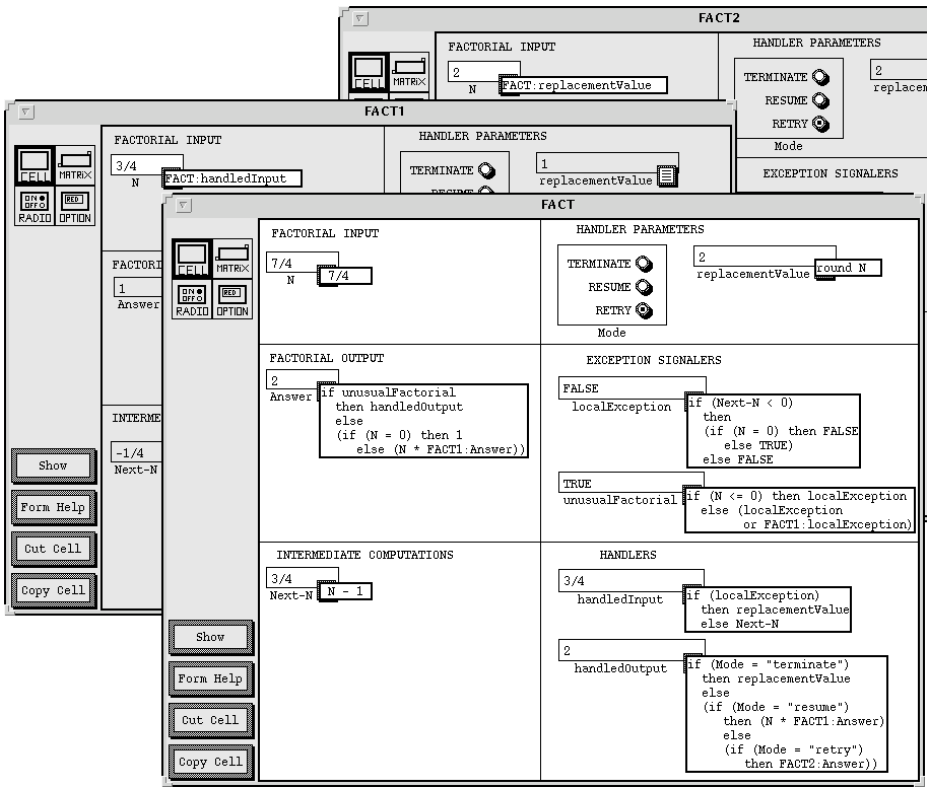


**Figure 5**: The cells in the input, output, and parameter areas (top and mid-left) are the interface to this routine. Replacement value exception handling (right half) is accomplished with formulas containing ordinary operators. Because there are no operators with "special powers," no deviations from the language's evaluation model are needed.

will provide a formula for input cell *N*, and will set the *replacementValue* parameter by entering the desired formula on their copy of the Fact form. Cell *Mode* is a radio button group used to specify the exception handling mode. (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). A programmer sets the mode parameter simply by pushing one of these buttons.

Except for the above input/parameter cells and the output cell *Answer*, all the other cells on the form will eventually be hidden; they are internal, and thus of concern only to the programmer developing the factorial library routine. These include the exception signalers (*localException* and *unusualFactorial*) and handlers (*handledInput* and *handledOutput*). (Note that these exceptions are not error values.) The underlying error value model will automatically handle any exceptions that are not covered by the explicit signalers and handlers.

### *3.3.2 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. Our approach explicitly supports the first three of these. The fourth, propagation, is superseded because the handler can itself signal an exception. The fifth, transfer of control, is not applicable to declarative approaches. The behaviors of the three explicitly-supported modes are:

Terminate: If an exception is signaled, use the replacement value as a substitute for the *final output cell(s)* on the form.

Resume: If an exception is signaled, use the replacement value as a substitute for the *cell that caused the exception.*

Retry: If an exception is signaled, use the replacement value as a substitute for the *initial input cell(s)* on another invocation of the form.

Table 1 shows example behavior of the factorial routine under each exception handling mode.

## 4. Unusual Features of the Approach

The three characteristics needed for a declarative language to effectively support this approach to exception handling are (1) there must be language-level support for the error value model to consistently provide the broad "safety net" that deals with all exceptions not explicitly

| Handler parameters | Initial invocation | Recursive calls |
|---|---|---|
| Terminate, replacement value of *error* | N = 7/4<br>Next-N = 3/4<br>Answer = *Error* | N = 3/4<br>Next-N = -1/4: exception signaled.<br>Answer = *Error* |
| Resume, replacement value of 1 | N = 7/4<br>Next-N = 3/4<br>Answer = 7/4 * 3/4 *1 | N = 3/4<br>Next-N = –1/4: exception signaled.<br>Answer = 3/4 * (a call to Fact with N=1) = 3/4 |
| Retry, replacement value of (round N) | N = 7/4<br>Next-N= 3/4 | N = 3/4<br>Next-N = –1/4: exception signaled |
| | Try N=2:<br>Next-N= 1<br>Answer = 2 | N = 1:<br>Next-N = 0<br>Answer = 1 |

**Table 1**: The behavior of library routine FACT under different exception handling modes. Internal calculations are shaded; interface cells (parameters and results) are unshaded. The use of an error value in terminate mode shows a combined use of the error value model and replacement value model.

handled by the programmer; (2) the language must have suitable abstraction facilities to enable the modularity and parameter-passing functionality that characterize replacement value model semantics; and (3) the language should be visual. We have already illustrated the first two points; in this section we will discuss the third, as well as other aspects of the approach that are unusual.

### 4.1 What Visualness Adds to the Approach

What advantages do a language that is visual provide this exception handling mechanism? Perhaps most important is the fact that, without the visual aspects, the structure needed for proper support of the replacement value model would be hard to create. For example, use of descriptive labels like “Exception handler” and lines to partition the form into conceptual regions separating exception handling cells from program logic cells helps create a stylized interface to the form to promote modularity/cohesion of related cell groups on the form. (In our system, such decorative devices are just borderless, nameless cells that have been arranged as desired.) In textual languages, such structure is instead achieved using specially empowered handling and signaling operations, handler specification sections, and the like.

In addition, the advantages that come to other kinds of programming through visual mechanisms also impact exception handling programming. For example, the use of widgets like radio buttons can promote robustness in parameterizing the handlers. When calling a form with replacement value exception handling, the programmer 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 programmer of the library routine can test exception handling behavior interactively while developing it. Later, programmers of calling forms can learn about the library routine's exception handling behavior by trying it with sample inputs, without having to refer to separate documentation or the routine's implementation details to understand it.

A debugging challenge in textual exception handling approaches is to identify the original source of an error, but visual mechanisms can help with this task. For example, Forms/3 provides tiny glyphs, general purpose dataflow arrows, and (soon) special error arrows, to help programmers track down sources of errors. Tiny glyphs can be made to appear in cells' formulas, showing the values of referenced cells along with the cell names referenced, to save the programmer the effort of searching for them. As in some other systems, in Forms/3 dataflow arrows to the selected cell can be automatically drawn. We are currently working on a new feature in which a different mouse button, when clicked on any cell, draws a red arrow directly from the original source of the error (the first cell in the selected cell's dependency chain with an error value).

### 4.2 What the Approach Adds to VPLs

As more VPL designers look toward scalability, the software engineering characteristics of their languages become increasingly important [3]. The replacement value model promotes these characteristics in several ways. For example, it allows implementation details for detecting, signaling, and handling exceptions to be hidden from callers. At the same time, it enhances the generality of a library routine, because the routine need not define the application-specific course of action to be taken when exceptions are signaled, since the desired exception handling behavior is specified by the callers.

The factorial example demonstrates how our approach follows all of Yemini's and Berry's software engineering guidelines (listed in Section 2). 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/verifiability characteristics. For parameterized handlers, a programmer specifies the desired parameters on a copy of the routine's form and refers to that copy's final answer from the "caller" form. For scope/type/verifiability compatibility, our strategy is

simply to satisfy the seamlessness constraint, which avoids any new scope, type, or verifiability issues.

The combination of the replacement value model with the error value model also adds robustness, because the latter model provides default error handling for conditions not handled explicitly by programmer-provided handlers. This is an unusual feature; it is more usual for programming languages to simply abort to the O/S if an exception is left unhandled or if a handler itself generates an unhandled exception.

### 4.3 Return Values and Exception Notifications

One inconvenience with traditional approaches to exception handling in declarative languages is that when exceptions are raised during function evaluation, the handled result is the function's only return value. In some situations, it may be useful to have both the handled answer and an exception notification returned as outputs of a function. This functionality can be simulated in various ways, such as with by-reference parameters in imperative languages, or with multiple return values packaged into a list in functional languages, but these ways often detract from the convenience and consistency of a program. However, in Forms/3 a procedure (form) can have multiple outputs (cells). For example, we could add to the clock form in Figure 2 two additional visible cells, *minuteError* and *hourError*, whose visibility would notify the user whenever an exception arises; also, these cells could be referenced, just like other visible cells, by other formulas in the program.

### 4.4 Liveness Versus Termination

The fact that our language is fully live introduced an interesting challenge. Forms/3 uses lazy evaluation, which means that values are only computed if they are needed to produce a program's output. However, in a live environment, everything on the screen is needed for output. As a result, when the program is being developed, extra exceptions can occur simply because *all* the program's calculations, even the intermediate values, are being displayed. This can lead to non-termination. For example, intermediate calculations of recursive programs that will be averted by exception handling logic in the final program will be executed during program development simply because they are on the screen.

The problem stems from the fact that the edit-time feedback provided by liveness's continuous execution does not wait until the program is completed. To solve this problem, the system needed to be able to detect a non-terminating calculation and return an error value. To approximate a solution to this, we decided to simply return an error value in the event of stack overflow. This solution need not impact the program's runtime efficiency after program development is complete, since explicit exception handling logic can prevent non-terminating calculations from ever being demanded, as in the Factorial example seen in Figure 5.

## 5. Conclusion

By seamlessly integrating replacement value exception handling into the simple error value model approach as found in spreadsheets, we have shown that exception handling need not add new factors that would compromise the ease of reasoning about a declarative VPL. Rather, we have exploited such a language's declarative and visual characteristics to produce the following new features for exception handling in VPLs:

- programmers use only ordinary language operators, such as if-then-else, for all exception signaling and handling;
- visual aspects of the language are exploited to allow full-featured exception handling without employing complex, Superman-like mechanisms; and
- the approach is entirely compatible with liveness.

Most important of all, because the approach to replacement value exception handling simply combines error value exception handling with abstraction, a programmer can use either model or both together with complete flexibility. In fact, as the examples show, error value exception handling is enough to provide an acceptable level of exception handling without the programmer writing any exception handling code at all in some cases. What adding the replacement value model via the error value model contributes is software engineering characteristics that are important for scalability, without the increase in language complexity that usually accompanies the introduction of exception handling capabilities into a language.

## Acknowledgments

We would like to thank the members of the Oregon State University VPL Research Group for their work on the Forms/3 implementation and for their feedback on the exception handling approach.

## References

1. Bretz, M. and J. Ebert, "An exception handling construct for functional languages", *2nd European Symposium on Programming*, Nancy, France, *LNCS 300*, Springer-Verlag, March 1988.
2. Burnett, M. and A. Ambler, "Interactive Visual Data Abstraction in a Declarative Visual Programming Language", *Journal of Visual Languages and Computing* 5(1), March 1994, 29-60.

3. Burnett, M., M. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee, "Scaling Up Visual Programming Languages", *Computer*, March 1995.
4. Cox, P. and T. Pietrzykowski, "Using a Pictorial Representation to Combine Dataflow and Object-Orientation in a Language Independent Programming Mechanism", *International Computer Science Conference*, 1988, 695-704.
5. Feo, J., D. Cann, and R. Oldehoeft, "A Report on the Sisal Language Project", Lawrence Livermore Nat. Lab. Report UCRL-102440 Rev. 1, 1990.
6. Goodenough, J., "Exception Handling: Issues and Proposed Notation", *Communications of the ACM* 18(12), 1975, 683-696.
7. Ludolph, F., Y. Chow, D. Ingalls, S. Wallace, and K. Doyle, "The Fabrik Programming Environment", *1988 IEEE Workshop on Visual Languages*, Pittsburgh, PA, Oct. 10-12, 1988, 222-230.
8. *Microsoft Excel 4.0 User's Guide 1* and *Microsoft Excel 4.0 Function Reference*, Microsoft Corporation, 1992.
9. Reeves, A., D. Harrison, A. Sinclair, and P. Williamson, "How to Make a Lazy Functional Language Exceptional", *IEEE TENCON '89* , Bombay, Nov. 1989, 179-185.
10. Sebesta, R., *Concepts of Programming Languages*, *3rd ed.*, Addison-Wesley, Reading, MA, 1996.
11. Wadler, P., "How to Replace Failure by a List of Successes: A Method For Exception Handling, Backtracking, and Pattern Matching in Lazy Functional Languages", in *Functional Programming Languages and Computer Architecture*, Nancy, France, *LNCS 201*, Springer-Verlag, Sept. 16-19, 1985, 113-128.
12. Yang, S. and M. Burnett, "From Concrete Forms to Generalized Abstractions through Perspective-Oriented Analysis of Logical Relationships," *1994 IEEE Symposium on Visual Languages*, St. Louis, MO, Oct. 4-7, 1994, 6-14.
13. Yemini, S., D. Berry, "A Modular Verifiable Exception-Handling Mechanism", *ACM TOPLAS* 7(2), Apr. 1985, 213-243.
14. van Zee, P., M. Burnett, "Exception Handling in the Visual Programming Language Forms/3", TR 95-60-1, Oregon State Univ. Computer Science Dept., Mar. 1995.