A RAND NOTE

A "Propagative" Approach to Sensitivity Analysis

Jeff Rothenberg, Norman Z. Shapiro,
Charlene Hefley

July 1990

RAND

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under RAND's National Defense Research Institute, a federally funded research and development center supported by the Office of the Secretary of Defense and the Joint Chiefs of Staff, Contract No. MDA903-85-C-0030.

Published by The RAND Corporation
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

A RAND NOTE

N-3192-DARPA

A "Propagative" Approach to Sensitivity Analysis

Jeff Rothenberg, Norman Z. Shapiro,
Charlene Hefley

July 1990

Prepared for the
Defense Advanced Research Projects Agency

RAND

A "Propagative" Approach to Sensitivity Analysis

Jeff Rothenberg Norman Z Shapiro Charlene Hefley

The RAND Corporation*
1700 Main Street
Santa Monica, CA 90406-2138

ABSTRACT

Large-scale simulations often involve huge numbers of parameters, making it prohibitive to run more than a tiny fraction of all potentially relevant cases. Sensitivity analysis attempts to show how sensitive the results of a simulation are to changes in its parameters; this is an important tool for promoting confidence in a simulation and making its results credible. However, the computational cost of traditional approaches to sensitivity analysis prevents its use in many cases. We show that this cost is logically unnecessary and can be largely avoided by propagating and combining sensitivities during a computation, rather than recomputing them. We describe this "propagative" approach to sensitivity analysis and present the algorithm we have implemented to explore its potential.

1. Overview

A simulation can be viewed as a single top-level function, which may have thousands or even tens of thousands of parameters. It is generally prohibitive to run such a simulation (i.e., evaluate this function) for more than a tiny fraction of all possible combinations of parameter values. Simulations are therefore typically run for relatively few cases, representing the most important, likely, or promising of these parameter values. Sensitivity analysis attempts to show how sensitive the results of a simulation are to varying the values of its parameters. Since exhaustive evaluation of a simulation for all cases is rarely possible, sensitivity analysis is especially important for promoting confidence in the "robustness" of a model (i.e., showing that its results are independent of minor changes to its parameters) and for indicating which parameter values are the most important ones to validate in order to make the model believable [3], [4].

The simplest approach to sensitivity analysis, which we refer to as "naive perturbation", requires running a simulation many times, while perturbing individual parameters to see how the results differ. The computational cost of this process is prohibitive for all but the most trivial simulations, which is why sensitivity analysis is rarely performed in simulation. Furthermore, faster computers do not solve this problem, since their increased capacity is invariably used to build bigger and more elaborate models rather than to perform sensitivity analysis on existing models. An alternative approach to sensitivity analysis, called Perturbation Analysis [6], [7], applies only to certain kinds of simulation and requires substantial mathematical knowledge about the process being simulated. The inability to perform sensitivity analysis cost-effectively for arbitrary simulations therefore remains an ongoing and critical impediment to the acceptance of these models as credible planning and decision aids. This paper describes research on a computationally feasible way of performing sensitivity analysis in a simulation context [5].

2. Background

Consider the abstraction of a simulation shown in Figure 1, where a top level function, *Sim*, invokes many levels of nested subfunctions,† each of which may be called many times. In most cases of practical interest, the

*This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) under the auspices of RAND's National Defense Research Institute, a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense, under contract No. MDA903-85-C-0030. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official opinion of DARPA, the U.S. government, or any person or agency connected with them.

†In this discussion, the term "subfunction" is used to mean a function that is called by another function, whether or not it is defined within the lexical scope of its caller.

Reprinted from *AI*, Simulation and Planning in High Autonomy Systems, Bernard Zeigler, Jerzy Rozenblit (eds.), March 26–27, 1990, pp. 10–16,

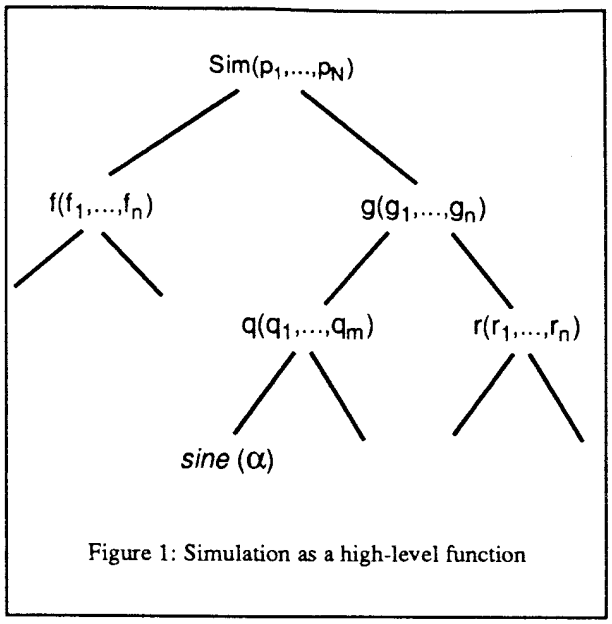

Figure 1: Simulation as a high-level function

function *Sim* will have a large number of parameters p_k (k = 1,..,N) relative to the number of parameters of most of its subfunctions. The naive perturbation approach to sensitivity analysis attempts to approximate a local partial derivative for each parameter p_k by perturbing p_k in the neighborhood of its value at some point of interest, $P = \{p_k\}$. The usual way to approximate the partial derivative would be to compute the Cauchy ratio: $\frac{Sim(p_1, \ldots, p_k + \Delta, \ldots, p_N) - Sim(p_1, \ldots, p_N)}{\Delta}$, which for convenience we will write $(Sim(P+\Delta_k) - Sim(P))/\Delta_k$. This approach will execute *Sim* once to compute $Sim(P)$ plus one or more times for each parameter p_k, each time executing each subfunction as many times as it is called. Each subfunction f will therefore be executed on the order of N times, where N is the number of parameters of *Sim* (the exact number of times f will be executed is a function of the number of times a single parameter must be perturbed in order to approximate a partial derivative and the number of times f is invoked by *Sim*).

Now consider some subfunction $g(g_1,\ldots,g_n)$ of *Sim* that has many fewer parameters than N (i.e., n<<N). Assuming that g is reasonably well-behaved, finding its sensitivity to variations in its parameters should not require anywhere near N evaluations: g is simply not complicated enough to warrant this much expenditure of effort! In fact, it should be possible to fully characterize the sensitivity of g by evaluating it on the order of n (rather than N) times, resulting in a tremendous performance improvement. Similarly, if g calls $q(q_1,\ldots q_m)$, where q has many fewer parameters than g (i.e., m<<n<<N), then this improvement can be reaped recursively. Furthermore, in some cases the sensitivity of a given subfunction may be known in detail from analytic knowledge; for example, the *sine* function shown in Figure 1 should not require perturbation at all, since its derivative is known.

In general then, naive perturbation wastes considerable effort in computing the sensitivities of the subfunctions of *Sim*. This observation forms the basis for the approach to sensitivity analysis proposed below.

3. A new approach to sensitivity analysis

We have designed a new **propagative** approach to sensitivity analysis that propagates and combines the sensitivities of functions during a computation. This is based on the observation that it is possible to compute a function's sensitivity as part of the process of computing its value [2]. The approach is motivated by the chain rule of the differential calculus, which defines the partial derivative of a composite function as a combination of the partial derivatives of its subfunctions (assuming these subfunctions are differentiable with respect to the parameters of interest). The chain rule justifies the "propagation" of sensitivity information in the sense that the sensitivity of a function f (i.e., its partial derivatives) in a given neighborhood can be used to approximate the value of f in that neighborhood. A geometric interpretation of this process for a function $g(x)$ of one parameter, is that the partial derivative g' of g in the neighborhood of a point x_0 represents the slope of g at x_0, which can be used to approximate the value of $g(x_0+\Delta)$ by means of the linear approximation $g(x_0+\Delta) = g(x_0)+g'(x_0)\Delta$.

Our propagative approach computes a representation of the sensitivity of each subfunction the first time it is executed and propagates that sensitivity information (in the above sense) rather than recomputing it each time it is needed. In cases where a subfunction's partial derivatives are known analytically, they can be declared as part of its definition, providing a direct representation of its sensitivity. In general, however, a numerical approximation to a subfunction's partial derivatives must be computed, for example by making it perturb itself once for each of its own parameters.* This approach is applied recursively to the subfunctions of each subfunction; any subfunction that

*Note that for simplicity, this paper focuses on sensitivity analysis in which one parameter at a time is varied, though the propagative approach applies equally well (and has even greater potential payoff) when combinations of parameters are varied together (i.e., when higher-order partial derivatives are required).

has more parameters than its caller will not benefit from this propagative approach and so is simply handled as in naive perturbation.

Conceptually, the approach proceeds as follows. In order to compute the sensitivity of *Sim*, each of its parameters p_k is perturbed in turn (as in naive perturbation) to compute the Cauchy ratios $(Sim(P+\Delta_k) - Sim(P))/\Delta_k$, for all k. In the process of evaluating *Sim* during these perturbations, the first time a particular subfunction f is called, make f compute its own sensitivity (by repeating this process recursively)* and return this information to *Sim* along with its normal return value. For all subsequent calls to f during the perturbation of *Sim*'s parameters, *Sim* uses the sensitivity information returned by f to approximate the value of f instead of re-executing it. This process is described in further detail in Section 5.

Our initial attempt to analyze the expected payoff of this approach resulted in a program that performed symbolic analyses of computations to evaluate the advantages of the new approach. We soon realized, however, that this payoff analysis was even more computationally intensive than the sensitivity analysis it was attempting to analyze! It became clear that it would be more efficient to analyze the payoff by actually implementing a propagative environment and using it to perform sensitivity analysis while collecting performance statistics. We therefore next implemented a novel, stand-alone computational environment (in LISP) to support the propagation and combination of sensitivities. This environment has allowed us to try our approach on a number of computations and to analyze its payoff.

Although the purely-applicative nature of LISP lends itself reasonably well to the propagative approach (since every function consists simply of subfunction calls), the approach would work equally well in a procedural language, where subfunction calls are embedded in a sea of in-line computation. In addition, not all functions in a given computation are of interest for sensitivity analysis: for example, it may not make sense to analyze the sensitivity of built-in functions (such as the LISP conditional function "cond"). Our computational environment therefore allows the user to designate which functions are "candidates" to be analyzed; these functions are automatically instrumented by the environment to keep track of how often they are called. The user further identifies whether candidate functions have return values that are discrete or continuous (though this could be done automatically). The environment then automatically modifies each candidate function to return a representation of its sensitivity as well as its normal return value.

*Though it is an important issue, we defer to a later paper the question of how to choose an appropriate value of Δ for each perturbation.

4. Locality and "occurrences" of functions

The description of the propagative approach presented so far ignores one major problem by implying that a given subfunction f has the same sensitivity each time it is called. This may be true for functions whose actual partial derivatives are known analytically, but in general the sensitivity information returned by a function cannot be assumed to represent more than a local, numerical approximation to its partial derivatives in the neighborhood of its invocation. To properly reflect this *locality assumption*, we introduce a novel concept, which we call an "occurrence" of a function. Consider the procedural pseudo-code shown in Figure 2, in which Function F1 appears twice in function H and F2 appears five times. During a single invocation of H, F1 will be evaluated twice, once with argument Z and once with argument r. We say that there are two "occurrences" of F1 in H (labelled F1:1 and F1:2 in Figure 2). In the absence of further information about Z, r, and F1, these two occurrences must be assumed to have entirely independent sensitivities, since Z and r may represent quite different neighborhoods of the domain of F1. The sensitivity of each occurrence of F1 must therefore be computed separately, as if each occurrence were a distinct function; the sensitivity information for each occurrence of F1 (as well as for every other occurrence of every subfunction called by H) is stored in an "approximation table" in H.

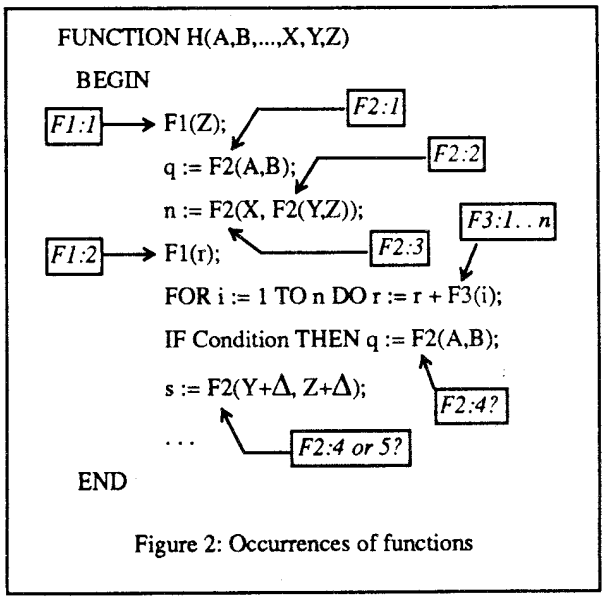

Figure 2: Occurrences of functions

If H now invokes itself again with one of its parameters perturbed, each occurrence of F1 will remain in the same neighborhood of its domain (assuming that the value of r is a continuous function of the parameters of H), so that the sensitivities previously computed for F1:1 and F1:2 (which were saved by H in its approximation table) should serve as valid approximations for these occurrences of F1 in the new, perturbed invocation of H.

Note that this concept of an occurrence of a function lies somewhere between an invocation and a lexical appearance. It is less dynamic than an invocation, in that there is a strong association between the occurrence F1:1 during the first invocation of H and the corresponding occurrence F1:1 during the second, perturbed invocation of H; this association allows the sensitivity computed for F1:1 during the first invocation of H to be used to approximate the value of F1:1 during the second, perturbed invocation of H. In addition, an occurrence is relative to the calling function (H), whereas an invocation of a function is generally independent of its caller. (Note, for example, that calling a recursive function represents only a single occurrence in the caller.)

An occurrence is more dynamic than a lexical appearance of a function, as illustrated by other functions in Figure 2. For example, F2 appears five times in H, but it may only occur 4 times during a given invocation of H, depending on the conditional test. Similarly, the number of occurrences of F3 depends on the dynamic upper limit of the FOR loop. Note that a given flow of control through H results in a particular set of occurrences of its subfunctions. To summarize, an occurrence of a function F in H corresponds to an invocation of F in a particular neighborhood of its domain, during the execution of a particular flow of control through H. To our knowledge, this concept is not supported by any programming language, including LISP and Prolog.

The occurrence defines the temporal scope of the sensitivity information for a function. That is, the sensitivity information for a given occurrence of a function F in H must persist from the time that H first invokes F to compute F's sensitivity through the subsequent invocations of F by H that use this sensitivity information to approximate F. This information cannot be stored in any single invocation of F (since it must persist across several such invocations), nor can it be stored as "own" information of F (since it must be associated with a particular flow of control through a particular calling function, H). The sensitivity information for a given occurrence of F must therefore be stored in the corresponding invocation of H (in the approximation table of H, as described above). Furthermore, this invocation of H must itself persist throughout the temporal scope of all occurrences of its subfunctions; that is, the initial evaluation of H and its subsequent perturbations must all be part of a single invocation of H. This motivates the algorithm described in the next Section.

5. An algorithm for the propagative approach

Whenever a candidate function is called normally during a computation (which we refer to as a "primary" call), our environment first causes the function to compute its normal return value and then causes it to place recursive "secondary" calls to itself, perturbing each of its arguments in turn. By so doing, the function computes its sensitivity and returns this information to its caller (along with its normal return value). During these secondary calls to itself, the function approximates the return values of its candidate subfunctions, rather than recomputing them for each secondary call. The default approximation technique is to apply the sensitivity of each called subfunction as a linear approximation of its value (where the sensitivity of each subfunction is computed recursively by this same process and returned to its caller). This process hinges on the notions of primary and secondary calls, which we illustrate with a simple example, shown in Figure 3.

Consider a top-level function H that calls a candidate function F, which in turn calls another candidate subfunction G. For simplicity, suppose that there is only a single occurrence of G in F, that F calls no other candidate subfunctions besides G, and that G calls no candidate subfunctions at all. Further, suppose both F and G are continuous-valued, differentiable functions. Since F is a candidate function, the propagative environment automatically interprets H's call to F as a **primary call** (shown as "p-call"). Upon receiving this primary call, F initializes its approximation table and proceeds to compute its normal return value, calling G in the process. Since G is also a candidate function, the environment interprets this as a primary call as well. Upon receiving this primary call, G proceeds to compute its normal return value (since G calls no candidate subfunctions, it does not need to create an approximation table). Having computed its value—and before returning from its primary call—G must compute its sensitivity, to be returned to F along with G's value.

To compute its sensitivity, G perturbs each of its own arguments in turn, placing a recursive **secondary call** (shown as "s-call") to itself for each perturbation. For example, if G has only a single formal argument, whose

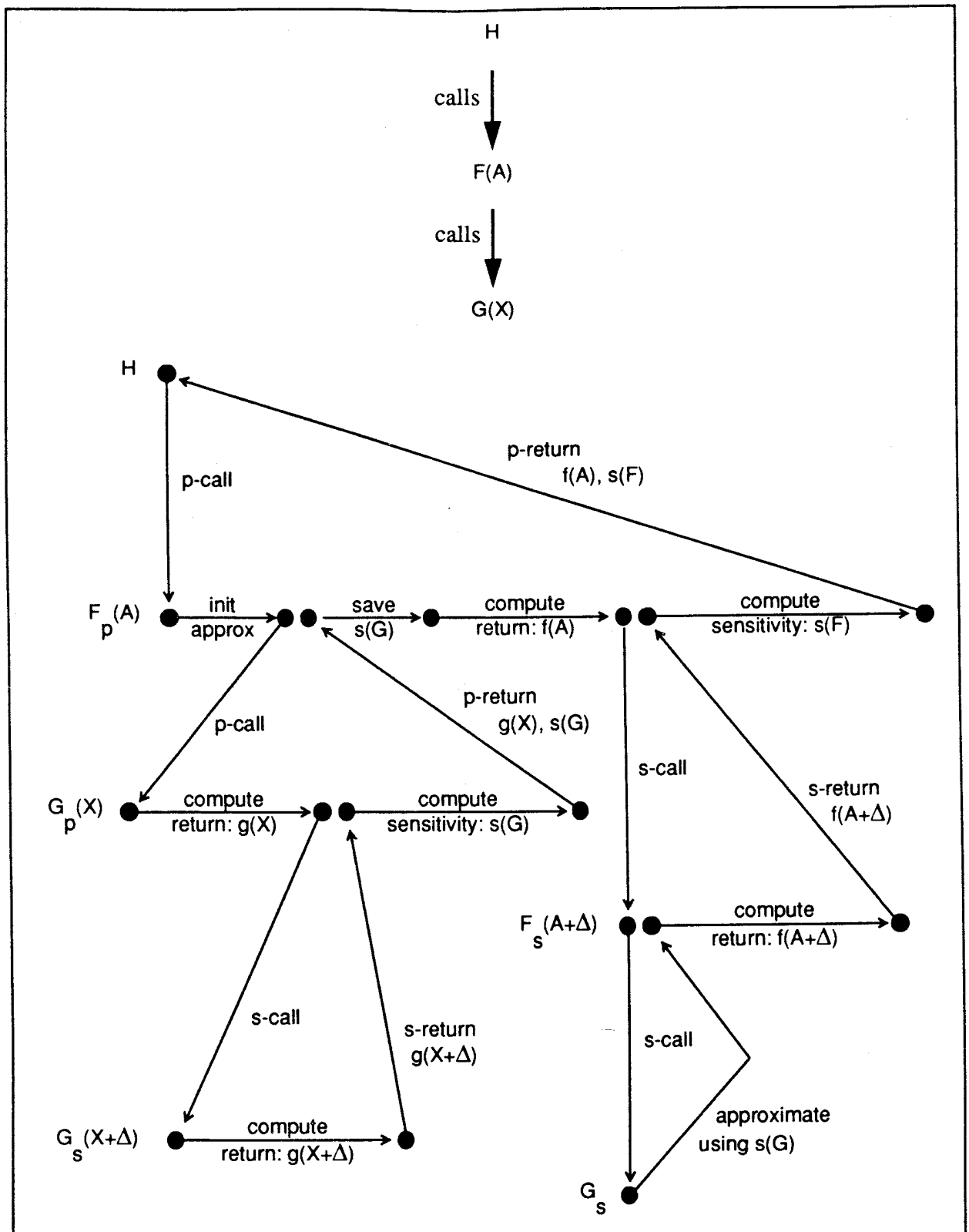

Figure 3: Propagative sensitivity analysis

value (as supplied in F's primary call to G) is X, then G will place a secondary call to itself with an argument of $(X+\Delta)$, where Δ provides the perturbation. This recursive secondary call to G returns the value $G(X+\Delta)$ to the primary invocation of G. Assuming, for simplicity of exposition, that a single perturbation is enough to produce a linear approximation to its "partial" derivative, G computes the Cauchy ratio $(G(X+\Delta) - G(X))/\Delta$ as its sensitivity information (shown as "s(G)"), and returns this to F along with G's normal return value (shown as "g(X)"). The value of the Cauchy ratio represents the sensitivity of G to its argument (in the neighborhood of the point X); this is used subsequently by F as a linear approximation to the value of G. If G had several arguments, it would perturb each one, generating secondary calls to itself to compute a Cauchy ratio for each partial derivative, and would return the collection of the resulting coefficients as its sensitivity information.

When the primary call to G returns to F, F separates the return value g(X) from the sensitivity information s(G), which it stores as its approximation table entry for this occurrence of G. F continues executing its own primary call (from H), using the value g(X) as needed to compute its own return value. Having computed its value (and before returning from its primary call), F computes its sensitivity (to be returned to H along with F's value, as the result of H's primary call to F). To compute its sensitivity, F perturbs each of its own arguments in turn, placing a recursive secondary call to itself for each perturbation. The results of these recursive secondary calls are used to compute the Cauchy ratios for F's own partials, as was done for G. However, during these secondary calls, whenever F would normally call G, it now "places secondary calls to G", by which we mean that it uses its approximation table entry for G to approximate the value of G rather than actually calling G. The environment ensures that the approximation table that F constructed during its primary call is available for use by these secondary, recursive invocations of F.

To summarize, a **primary call** to a candidate function F calculates its normal return value, in the course of which it places primary calls to its candidate subfunctions, each of which returns its sensitivity, as well as its normal return value. The calling function F stores this sensitivity information in an approximation table for later use. The primary call then initiates a series of recursive secondary calls by F to itself, to compute its sensitivity by perturbing its parameters. During these secondary calls, the value of each candidate subfunction is approximated using the sensitivity information previously stored in F's approximation table entry for that subfunction. It is the ability to approximate these return values (rather than recomputing them) that allows the propagative approach to outperform naive perturbation. When F has completed perturbing its parameters via recursive secondary calls, it will have derived its own sensitivity to its parameters: This information is returned by F to its caller to serve as F's entry in its caller's approximation table.

A **secondary call** by a function F to a candidate subfunction G essentially replaces the evaluation of G with an approximation, using the entry for G in F's approximation table (which was returned to F by G, when F placed its primary call to G).

One final point is that the flow of control through H must not change when it perturbs its parameters. This would invalidate the locality assumption (that secondary calls occur in the same neighborhood as their corresponding primary call), which justifies the use of approximations for subfunctions during secondary calls. In order to guard against this, an "occurrence-counter" counts subfunction occurrences during the primary call of H and records the value of the counter in the approximation table entry for each occurrence. During a secondary call of H, if the stored value for a particular subfunction occurrence does not match the dynamic value of the occurrence-counter or the parameter values for the call are not within the same neighborhood, then the control flow for this secondary call has deviated from that of the corresponding primary call; in this case the environment reports that the locality assumption has been violated.

6. Results

Our initial results indicate that this propagative approach has tremendous potential, reducing a combinatorial process to a linear one. In addition, we note that the approach has implications beyond sensitivity analysis: it suggests a novel computational paradigm in which functions replace themselves by approximations when they are first called, and these approximations are used for the remainder of a computation, e.g., to improve performance. Sensitivity analysis is simply one instance of this approach, using linear approximations based on partial derivatives; however, the approach—and the computational environment we have implemented—allow arbitrary approximations to be used.

Ultimately, we would like our environment to support standard, procedural languages such as C or Ada. This

requires modifying a compiler or building a preprocessor to automatically transform user-supplied functions into the form required by our propagative algorithm, with a minimum of declarative overhead for the user. In addition, further research is needed before this approach can be integrated into a realistic simulation environment. For example, although Boolean derivatives and a Boolean version of the chain rule can be defined [1], the general case of symbolic-valued functions requires further thought. Our computational environment allows functions of this sort, but only applies the propagative approach to those functions that are differentiable in the usual sense, performing naive perturbation for all others. However, even without such extensions, we believe this new approach holds great promise for the feasibility of sensitivity analysis in simulation.

REFERENCES

[1] Blanning, R. W., "Sensitivity Analysis in Logic-based Models," *Decision Support Systems*, Vol. 3, 1987, pp. 343–349.

[2] Boehm, B., personal communication, c. 1970.

[3] Bowen, K. C., "Analysis of Models," University of London, Department of Mathematics (unpublished), 1978.

[4] Miser, H. J., and E. S. Quade, *Handbook of Systems Analysis*, Elsevier, New York, 1988.

[5] Rothenberg, J., Narain, S., Steeb, R., Hefley, C., and Shapiro, N. Z., *Knowledge-Based Simulation: An Interim Report*, The RAND Corporation, N-2897-DARPA, July 1989.

[6] Suri, R., "Infinitesimal Perturbation Analysis for General Discrete Event Systems", *JACM*, Vol. 34, No. 3, July 1987, pp. 686-717.

[7] Suri, R., "Perturbation Analysis: The State of the Art and Research Issues Explained via the GI/G/1 Queue", *Proceedings IEEE*, Vol. 77, NO. 1, January 1989.