# Generic Programming, Partial Evaluation, and a New Programming Paradigm

Christopher Landauer, Kirstie L. Bellman
*Aerospace Integration Science Center, The Aerospace Corporation,*
*P.O. Box 92957, Los Angeles, California 90009-2957*
*E-mail: cal@aero.org, bellman@aero.org*

## Abstract

*We describe in this paper a new approach to Generic Programming that combines our integration results with Partial Evaluation methods for adaptation. Our approach supports Partial Evaluation by providing much more information than is usually available, including explicit meta-knowledge about the program fragments and their intended execution environments. We make some ambitious claims here, so we provide some detail about our methods, to justify our interest and expectations. We are not claiming to have solved the problem; only that we think our methods circumvent some of the known difficulties that were previously identified or encountered in approaches to Generic Programming.*

## 1 Introduction to the Problem

We describe in this paper a new approach to Generic Programming that uses our integration results together with recent progress in Partial Evaluation methods for adaptation. Our approach to partial evaluation provides an adaptation mechanism by having much more information than usual, including explicit meta-knowledge about the programs and their intended execution environments.

Generic Programming is an attempt to get the right level of abstraction in algorithms, so that the essential parts of the algorithm are displayed, and the details that make it conform to an execution environment are added during pre-compile time by a kind of specialization. Of course, it has been a goal of computer programming since the beginning to get the right level of abstraction in algorithms. The long-anticipated result would be that the program as written would contain all and only the essential parts of the algorithm. All of the characteristics of the execution environment that affect the map from the algorithm to the program would then have to be made explicit, but they could be deferred from program construction time to pre-compile time: the algorithm would then instantiated or adapted to make the program conform to the specific environmental characteristics before (or during) compilation. Even after many years of progress in this direction, we still have a distance to go [21].

We have a new approach to this problem that we think is very promising.

*Problem Posing* is a new declarative programming style that unifies all major classes of programming. Programs interpreted in this style do not "call functions", "issue commands", "assert constraints", or "send messages"; they "pose problems". Program fragments are not defined as "functions", "modules", "clauses", or "objects" that do things; they are written as "resources" that can be "applied" to problems.

The Problem Posing Interpretation uses *Knowledge-Based* Polymorphism to map from a problem specification in the ambient context to the computational resources that will provide or coordinate the solution. Any programming language can be interpreted in this new way.

This interpretation is particularly effective in combination with *wrappings*, our computationally reflective knowledge-based approach to integration infrastructure, since wrappings provide the Knowledge-Based Polymorphism that mediates between the posed problems and the applicable resources.

Our approach to Generic Programming is to write programs using *wrex*, the wrapping expression notation, since it explicitly leaves the posed problems in the program, and relies on the selection of computational resources to address the problems. The flexibility of run-time choices may be carried to as much detail as desired, from large-scale "legacy" components to individual arithmetic calculations. Finally, we use Partial Evaluation to make early decisions when possible, in effect "compiling out" the constant decisions, so that there is no run-time cost for decisions that will be made in the same way every time. We describe this

0-7695-0001-3/99 $10.00 (c) 1999 IEEE

new approach to generic programming, especially our integration technology that underlies and powers it.

## 2 Constructed Complex Systems

We have defined *Constructed Complex Systems* [16] [17] to be computer-mediated systems that are too complex for a single model to describe [19]. They tend to be software and hardware systems that have heterogeneous processing requirements or that have to function in complex environments. We believe that designing, building and managing such a system requires explicit attention to the infrastructure, including explicit models of the system, its architecture, and the environment in which it is expected to operate [2], and suitably flexible computer-based design support. Design support systems for constructed complex systems are complex themselves, since they must accommodate many different kinds of model and problem domain. In general, global design management requires integration of multiple criteria, which means that multiplicity is unavoidable because complex tradeoff calculations must be made.

Our original motivation was very large space systems [4] [19], but the results have become much more generally applicable: in our "Integration Science" [15], we study better integration methods, which are both more permissive, flexible, and semantically powerful, and more supportive of formality in the analyses of the processes and products of integration.

We describe the organization of this paper as a circular journey through some interesting new research areas in Computer Science. Actually, some of them are not really new, but there is new hope that significant progress can be made.

In Sections 3 and 4, we introduce Generic Programming and Partial Evaluation. In Section 5, we introduce the *Problem Posing* interpretation, which is a new interpretation of any programming language that uses Knowledge-Based Polymorphism. Then in Section 6, we describe the underlying wrapping theory. In Section 7, we show how the Problem Posing interpretation works with wrappings, and describe the corresponding notation *wrex*. Finally, in Section 8, we describe our conclusions and claims for Generic Programming, Partial Evaluation, and Problem Posing.

## 3 Generic Programming

In the most general sense, generic programming is about making programs more flexible, often using more interesting kinds of polymorphism, more interesting kinds of parameters (e.g., programs, types, constructors, etc.), and more interesting kinds of program analysis than compilation. There have been gradual improvements over the last 50 years:

> Machine language $\longrightarrow$ assemblers $\longrightarrow$ compilers $\longrightarrow$ interpreters,

but we are still looking for more effective techniques, since we still have some distance to go [21]. Partial Evaluation, to be described next, looks like a good way to go, because it makes explicit our choices about when to evaluate expressions and when to instantiate program fragments.

We see examples of generic programs in almost every book that describes a computational technique. The algorithms are often written descriptively for explanation, using some kind of pseudo-code. We want those descriptions to be the actual generic programs.

Generic genetic programming algorithm:

```
generate initial population
loop until done
    compute offspring
    retain only the ones with best fitness
```

Generic simulated annealing algorithm:

```
set initial temperature and position
loop until done
    generate potential new position
    check movement criterion
    reduce temperature
```

Generic k-means clustering algorithm:

```
generate initial cluster centers
loop until done
    partition data into nearest-center clusters
    recompute cluster centers
```

Note that each algorithm has a "loop until done" statement, which means very different things in different contexts.

The use of context to assist in the specialization process is very important to our approach. For example, "sanity checks" in computational design models are constraints determined by context. In fact, context provides and organizes _all_ interpretation of symbols

and symbol structures; we use it to collect and organize the symbol structure interpreters. Context determines what representations are important or useful, because it determines what each symbolic expression means or does. In addition, as we try to imbue our symbolic systems with more semantics, we bridge the gap by taking semantics to _be_ the interpretation of syntax, and the interpreters to be provided by the context.

## 4 Partial Evaluation

The basic approach of Partial Evaluation is very simple [22] [9]: if you know a program and some of the parameters, then you can specialize the program, and the resulting specialized program should be (and often is) faster. This is another old dream of computing, from the original high-level languages through extensible languages, very high-level languages, domain-specific languages, and executable specification languages. We particularly want completely automatic techniques, so that we can write our programs in the generic languages and leave the details to the compilation and execution environment. The problem has been that it is very difficult to make it work, though there are some encouraging recent signs [5] [8] [10] [9].

Our wrappings contain much more semantic information about the program components than is usually provided with a program, and therefore much more than can be extracted from a program by itself. It is this extra information that we intend to use in our partial evaluations.

There are also some methods originally developed for use in optimizing compilers that are very useful here, such as symbolic execution, unfolding function calls, partly unrolling iterations and recursions, and "program point specializations", which means replicating parts of the program with different specializations, definition creation, folding, and a kind of memoization of function calls and other program fragments. Replication particularly needs care to avoid exponential blowup.

As we describe some of the promise of partial evaluation, we will imagine a program "mix" that can do partial evaluation (our notation takes [p] as an executable form of the program p):

for all programs p, and all inputs in1 and in2,
[[mix] (p, in1)] in2 = [p] (in1, in2),

so mix specializes p to the inputs in1 (such programs do exist for some languages [9]). Partial evaluation provides an adaptation mechanism for generic programs (we can easily imagine that in1 is the specialization information for a particular execution environment).

We turn to some formal reasoning that lies at the heart of the promise of partial evaluation: called the Futamura Projections [9], these are ways to use partial evaluation to produce compilers and compiler generators automatically.

We start with a definition: an *interpreter* is a program interp for which:

for all inputs in, and all source programs s,
[interp] (s, in) = [s] in

We claim that a kind of "compilation target" t for source program s can be computed:

t = [mix] (interp, s).

We call it a compilation target because it has the same computational effect as the original program s, since for any input in,

out = [s] in
= [interp] (s, in)
= [[mix] (interp, s)] in
= [t] in,

so mix specializes the interpreter via part of its input (the source program s). This is the *first* Futamura projection.

Now

comp = [mix] (mix, interp)

turns out to be a compiler for the language interpreted by interp:

for any source program s,
the compilation target t has
t = [mix] (interp, s)
= [[mix] (mix, interp) ] s
= [comp] s.

This is the *second* Futamura projection.

So far,

t = [mix] (interp, s)

is a compiled form of s,

comp = [mix] (mix, interp)

is a compiler for the language interp interprets. Finally,

cogen = [mix] (mix, mix)

is a compiler generator for the language interp is written in:

for any interpreter interp,
source program s, and input in,
[s] in= [interp] (s, in)
= [[mix] (interp, s)] in
= [[[mix] (mix, interp)] s ] in
= [[[[mix] (mix, mix)] interp] s] in,
= [[[cogen] interp] s] in,

so

[cogen] interp = [comp]

is a compiler for the language of s. This is the *third* Futamura projection.

The use of these formal projections is intriguing, but the performance of the programs is important. There is good news here, too. We have shown two ways to produce output with these notions:

out = [interp] (s, in)
= [s] in

t = [mix] (interp, s)
= [comp] s

comp= [mix] (mix, interp)
= [cogen] interp

cogen= [mix] (mix, mix)
= [cogen] mix

The second way is often something like 10 times faster (this is well-known for the first pair above, in which interpreters in general are slower than compilers). The first way is usually much easier to write (once the program mix is written). We want to use this fact to write faster programs more easily. The hard part here, pretty clearly, is writing the program mix in the first place. In the next part of the paper, we describe our knowledge-based integration infrastructure, which we believe will lead to much better understanding of some of the issues required for effective partial evaluation.

# 5 Problem Posing Programming Paradigm

In this section, we very briefly describe our Problem Posing Programming Paradigm [13] [14] [15] [16] [17] [18], which underlies our approach to generic programming. We have defined *Problem Posing* as a new declarative interpretation of programs that unifies all major classes of programming. It uses what we have called *Knowledge-Based* Polymorphism to map from problem specifications to the computational resources that will provide or coordinate the solution. Any programming language can be interpreted in this new way. Problem Posing can therefore be viewed as a new programming paradigm that changes the semantics, not the syntax, of any programming language.

The basic expressive notion of this interpretation is the "posed problem", and the basic computational component is the "resource". They can be connected by the "wrappings" as defined in the next section, or by any other method that uses processes and associated knowledge bases that convert a posed problem into coordinated collections of resources that can address the problem.

We have demonstrated the conceptual utility of "problem posing" in our own descriptions of systems [13] [16]. The "Problem posing" interpretation unifies all major classes of programming. Programs interpreted in this style do not "call functions", "issue commands", "assert constraints", or "send messages"; they "pose problems" (these are information service requests). Program fragments are not written as "functions", "modules", "clauses", or "objects" that do things; they are written as "resources" that can be "applied" to problems (these are information service providers). Any programming language can be interpreted in this new way.

# 6 Integration Infrastructure: Wrapping

The Problem Posing interpretation is particularly effective in combination with "wrappings", our computationally reflective knowledge-based approach to integration infrastructure [14] [15] [17] (and references therein). In this section, we give a brief overview of the wrapping approach; many more details are elsewhere.

## 6.1 Wrapping Overview

The advantages of our knowledge-based integration technology are (1) a simplifying uniformity of description, using the meta-knowledge organized into *Wrapping Knowledge Bases*, and (2) a corresponding simplifying uniformity of processing that meta-knowledge using algorithms called *Problem Managers*, which are active integration processes that use the meta-knowledge

to organize the system's computational resources in response to problems posed to it by users (who can be either computing systems or humans). In particular, since the entire process is recursive [12], wrappings provide a general way to allow specialized methods to participate, in contexts for which they are appropriate.

The wrapping theory has four basic features.

1. ALL parts of a system architecture are *resources* that provide an information service, including programs, data, user interfaces, architecture and interconnection models, and everything else.

2. ALL activities in the system are *problem study*, (i.e., all activities *apply* a resource to a *posed problem*), including user interactions, information requests and announcements within the system, service or processing requests, etc.. We therefore specifically separate the problem to be studied from the resources that might study it.

3. *Wrapping Knowledge Bases* contain *wrappings*, which are explicit machine-processable descriptions of all of the resources and how they can be applied to problems to support what we have called the *Intelligent User Support* (IUS) functions [3]:

   - *Selection* (which resources to apply to a problem),
   - *Assembly* (how to let them work together),
   - *Integration* (when and why they should work together),
   - *Adaptation* (how to adjust them to work on the problem), and
   - *Explanation* (why certain resources were or will be used).

   Wrappings contain much more than "how" to use a resource. They also help decide "when" it is appropriate, "why" you might want to use it, and "whether" it can be used in this current problem and context.

4. *Problem Managers (PMs)*, including the *Study Managers(SMs)* and the *Coordination Manager (CM)*, are algorithms that use the wrapping descriptions to collect and select resources to apply to problems. They use implicit invocation, both context and problem dependent, to choose and organize resources. The PMs are also resources, and they are also wrapped.

The most important conceptual simplifications that the wrapping approach brings to integration are the uniformities of the first two features: the uniformity of treating everything in the system as resources, and the uniformity of treating everything that happens in the system as problem study. The most important algorithmic simplification is the Computational Reflection [1] [11] [14] provided by treating the PMs as resources themselves: we explicitly make the entire system reflective by considering these programs that process the wrappings to be resources also, and wrapping them, so that all of our integration support processes apply to themselves, too. It is this ability of the system to analyze and modify its own behavior that provides the power and flexibility of resource use.

### 6.2 SM and CM

The wrapping processes are active coordination processes that use the wrappings for the Intelligent User Support functions. They also provide overview via perspective and navigation tools, context maintenance functions, monitors, and other explicit infrastructure activities.

The alternation between problem definition and problem study is organized by the *Coordination Manager* (CM), which is a special resource that coordinates the wrapping processes.

The CM runs a sequence of steps shown in Figure 1 (written in the *wrex* notation, which we describe in Section 7), that manages the overall system behavior.

The basic problem study sequence is monitored by a resource called the *Study Manager* (SM). which organizes problem solving into a sequence of basic steps that we believe represent a fundamental part of problem study and solution.

The SM is organized into a sequence of basic steps that we believe represent a fundamental part of problem study and solution. The default SM step sequence is shown in Figure 2 in the *wrex* notation.

#### 6.2.1 SM Recursion

The SM, as described thus far, is a (very) simple planner. We make the system Computationally Reflective by making the CM and SM recursive in a "meta-direction", that is, every step in their definition is a posed problem. The recursive use of the SM to pose problems that are part of its own and the CM processing prevents them from being just another general planner or recursive problem solver. The basic steps and their ordering in the CM and SM form a default case that can be superseded by any more clever mechanisms for particular problem contexts.

```
define
CM [ <user> ]:
        [
        Find context [ <user> ],

        for ever :
                [
                <problem> = Pose problem [ <user> ],
                <result> = Study problem [ <user>, <problem>, <context> ],
                Present results [ <user>, <result> ]
                ]
        ],
```

Figure 1: Coordination Manager (CM) Step Sequence, in *wrex*

```
define
SM-simple [ <poser>, <probname>, <data>, <context> ]:
        [
        <candidates> = Match resources [ <probname>, <data>, <context> ],
        if (<candidates> == empty): then return FAIL_NO_CANDS,

        <candidate applications> = Resolve resources [ <candidates>, <probname>, <data>, <context> ],
        if (<candidate applications> == empty): then return FAIL_NO_APPLS,

        /* choose one of the remaining candidate resources */
        <selection> = Select resource [ <candidate applications> ],

        <resource application> = Adapt resource [ <selection>, <probname>, <data>, <context> ],

        <resp> = Advise poser [ <poser>, <resource application>, <probname>, <data>, <context> ],
        if (not <resp>): then return FAIL_NO_APPLS,

        <result> = Apply resource [ <poser>, <resource application>, <context> ],

        <success flag> = Assess results [ <result>, <probname>, <data>, <context> ],
        return <result>
        ],
```

Figure 2: Study Manager (SM) Step Sequence, in *wrex*

In particular, the "study problem" step in the CM means that the SM is chosen in the same way as any other resource: resources that can address a "study problem" problem are selected using the default SM. That means that the SM we described above is only a default that occurs at the bottom of the recursion: there are other SMs that have slightly more interesting algorithms. More advanced versions of matching, selecting and so forth will be implemented by resources that are chosen like any others, using the same recursive steps.

Therefore, the posed problem approach makes everything a resource, as far down as one cares to go into the implementation, and the WKBs help select whether to further the meta-recursion by invoking the SM, or to stop it by going to a specially coded resource.

We are going to make essential use of this reflection in our partial evaluations: since the system has access to its own computational resources, it has a much better chance to make appropriate simplifications and specializations than if it had no knowledge of its own structure.

## 7 Wrapping and Problem Posing

In this section, we show how the problem posing interpretation works with wrappings, and describe the *wrex* notation we can use to express problems.

The basic expressive notion of wrapping is the "posed problem", and the basic computational component is the "resource" (this research led directly to the Problem Posing Interpretation). They are connected by the "wrappings", which consist of processes and associated knowledge bases that convert a posed problem into coordinated collections of resources that can address the problem. We have developed this notion into the *wrex* notation, which extends the application of wrapping all the way down to the data access and expression evaluation level of detail. For example, the CM and SM were written in *wrex*, in Figure 1 and Figure 2.

### 7.1 The Wrapping Expression Notation *wrex*

The wrapping expression notation *wrex* [13] is one possible way to use the Problem Posing paradigm. It is a "Problem Posing" notation intended both for programming "in-the-large", i.e., as an architecture description language for resources written in other more ordinary programming languages, and for programming "in-the-small", i.e., as a language in which to write some of the (perhaps less time-constrained or more experimental) computational resources. We think that both kinds of programming are necessary for building reliable systems because they allow us to make explicit the interconnection between architectural models and the component models [20].

The *wrex* notation extends the application of wrapping all the way down to the data access and expression evaluation level of detail. In this notation, a program does not issue commands, impose constraints, assert facts, invoke functions, or even send messages; it *poses problems*. The CM and SM step sequences in Figure 1 and Figure 2 are written in *wrex*.

Object-oriented languages like Smalltalk [6] gain at least part of their impressive conceptual simplicity and power of expression from having implicit notations for sending messages, and interpreting all expression evaluation as sending messages of one kind or another. The expression syntax of *wrex* reflects the central role of problem study even as the syntax for Smalltalk reflects the central role of message passing. We should not be nearly as interested as programmers in the selected resource (the analogue of message destination) as in the posed problem (the analogue of message type), since many different resources can satisfy an information service request, and specifying otherwise is one of the architectural rigidities that leads to reuse difficulties.

Since the fundamental activity of wrapping is "apply resource to problem", and the Study Manager (SM) is the resource usually applied to study a problem, we make that operation implicit, and use explicit text in the program for problem specifications and plans.

The analogue of message transmission in Smalltalk, in which juxtaposition is the implicit operator for sending messages, is the posing of problems (in some appropriate syntax):

<posed problem> ::= <problem name>
'[' <problem specification> ']'
[ 'by' <resource> ]
[ 'in' <context> ]

(using <resource> = default SM if omitted, which is usually, and <context> = poser context if omitted, which is almost always), in which juxtaposition is the implicit operator for studying problems. The <resource> is the one expected to interpret the problem posed, not the one expected to apply to the problem. The problem data is part of the <problem specification>, and is usually considered to be the same as the specification. The differences are unimportant for the purposes of this note. In general, we can pass the values that specify the problem either as problem data or in the context. The former is more useful in

case the data is problem specific or transient, and the latter when it might also be useful later on in the program.

Every problem has some amount of problem data (possibly empty). Every resource application has a success or failure result (which is made available to other problem specifications). All control issues are problem grouping and ordering issues. Control statements in *wrex* look almost the same as they do in current programming languages: sequencing with ',', grouping with blocks and procedures, and even grouping with loops and conditionals.

Nearer the low end in granularity, expression evaluations are also posed problems. We can use any usual syntax here, too, if we change the interpretation, so that, for example,

a + b

is the problem '+', with data 'a' and 'b'. It is easy to make compilers convert the infix notation properly.

Finally, the innermost part of expression evaluation is read-style data access, and the last part of assignment evaluation is write-style data access. The basic access mechanisms are reading and writing values, and the only real question is what kinds of value structures (i.e., data types) we want to allow. This question is application specific, and not part of the generic notation.

### 7.2 Syntax of *wrex*

We describe the syntax of *wrex* briefly in this subsection; most of the details should be clear from the examples. The syntactic items of *wrex* are Variables, Constants, and three kinds of grouping structures. Variables consist of multiple words, delimited by '<' and '>'. Constants are words, other symbols (such as operator symbols and delimiters), numbers (integers and otherwise), or strings (both single and double quoted).

The three Constructors for grouping items into structures are List, Set, and Group. A List contains multiple items, separated by commas, and delimited by '[' and ']'. A Set contains multiple items, separated by commas, and delimited by '{' and '}'. Finally, a Group also contains multiple items, separated by commas, and delimited by '(' and ')', but Groups not in other Groups can omit the parentheses. Syntax recognition of these structures is extremely easy.

To facilitate readability, we allow separated problem names, such as

find [ ... ] in [ ... ] using [ ... ],

. which has 'find ... in ... using' as a problem name. Programs are sequences of these syntactic items (i.e., Groups without parentheses). The simple posed problem is a Group whose last element is a List.

We describe these syntactic items using the features of the CM in Figure 1 on Page 6: The entire resource is one group (with a comma at the end to separate it from the following ones):

'define' 'CM' '[ ... ]' ':' '[ ... ]'.

'Find context'

is a separated problem name. So is

'define' ... ':' '[ ... ]'.

'for ever : [ ... ]'

is a posed problem with *wrex* statements as data

'<problem> = Pose problem [ <user> ]'

is also a group without parentheses. It is also a posed problem with problem name '='. Finally, 'if' and 'for' expressions are defined concepts, not primitive. So is '='.

### 7.3 Decision Times and Performance

Finally, we want to make a point about performance issues, and claim that the flexibility of *wrex* need not cost too much. Here we rely on the fact that partial evaluation has much more information to use than is usually the case.

Decisions about program structure are made at many different times

- Language Design time
- Compiler Generation time
- Program Generation time
- Compile time
- Link time
- Load time (for operating systems, this is boot time)
- Run time (for operating systems, this is multi-user time)

Requiring too many decisions to be made at Run time is as rigid as restricting them to any other times, and it also costs greatly in performance. If it is provided, a program can use knowledge of the execution environment with partial evaluation to reduce the time considerably [11]. When it has available the context and collection of available resources, it can reason about which

resource selections are made. If it can deduce that a posed problem is only addressed by one resource, it can avoid the full search and the SM recursion entirely.

In particular, in considering the partial evaluation of *wrex*, there are many more kinds of input available than for programs in most other programming languages:

- Program and input data (as usual),
- Wrappings and other semantic information as context data, and
- Collection of available resources.

Much greater specialization will result, and we hope much better programs: both faster AND more easily changed.

## 8 Conclusions and other Claims

We have now come full circle, back to the implications of these results to Generic Programming. The common theme of this paper has been to make the times at which design decisions must be made more flexible, without costing too much in performance. For Generic Programming, we can defer many decisions from Program Generation, Compile, and even Link time, to Run time. Actually, with Domain-Specific Languages and Architectures [7], we can make those decisions earlier as well as later. Because of Partial Evaluation, we can defer many decisions from Language Design and Compiler Generation time to Run time.

Our main claim for this research is that the perennial goal of programming, to make programs easier to write and more reliable, will be easier to achieve this way than others.

Using Partial Evaluation, the Problem Posing interpretation can produce regular compiler code, instead of leaving the decisions all to Run time. We don't know how fast it will be, but we're very hopeful about it. We want programs that are BOTH easier to write AND faster to run. Of course, before this can happen, we need to write some partial evaluators like "mix" for *wrex*, which we expect to be hard.

The Knowledge-Based Polymorphism with wrappings that we get with Problem Posing has interesting reuse and system re-engineering implications, such as Reuse without modification, and the Problem Posing interpretation also provides a kind of interoperability for systems written in (almost) any programming language. To use *wrex* as a coordination language, the various components need to be written in languages that can accept unsolicited information from outside the program, which is hard for some.

Finally, none of this solves the hard modeling problems in any particular domain (they are still hard). However, it does allow them to be solved more separately, and the integration of even partial solutions to be more automatic or at least better supported by the system.

## References

[1] Harold Abelson, Gerald Sussman, with Julie Sussman, *The Structure and Interpretation of Computer Programs*, Bradford Books, now MIT (1985)

[2] Kirstie L. Bellman, "The Modelling Issues Inherent in Testing and Evaluating Knowledge-based Systems", pp. 199-215 in Chris Culbert (ed.), *Special Issue: Verification and Validation of Knowledge Based Systems*, *Expert Systems With Applications Journal*, Volume 1, No. 3 (1990)

[3] Kirstie L. Bellman, "An Approach to Integrating and Creating Flexible Software Environments Supporting the Design of Complex Systems", pp. 1101-1105 in *Proceedings of WSC'91: The 1991 Winter Simulation Conference*, 8-11 December 1991, Phoenix, Arizona (1991); revised version in Kirstie L. Bellman, Christopher Landauer, "Flexible Software Environments Supporting the Design of Complex Systems", *Proceedings of the Artificial Intelligence in Logistics Meeting*, 8-10 March 1993, Williamsburg, Va., American Defense Preparedness Association (1993)

[4] Kirstie L. Bellman, April Gillam, Christopher Landauer, "Challenges for Conceptual Design Environments: The VEHICLES Experience", Revue Internationale de CFAO et d'Infographie, Hermes, Paris (September 1993)

[5] C. Consel, O. Danvy, "Tutorial Notes on Partial Evaluation", *Proceedings 20th PoPL: The 1993 ACM Symposium on Principles of Programming Languages*, Charleston, SC (January 1993)

[6] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley (1983)

[7] Barbara Hayes-Roth, Karl Pfleger, Philippe Lalanda, Philippe Morignot, Marka Balabanovic, "A Domain-Specific Software Architecture for Adaptive Intelligent Systems", *IEEE Transac-*

*tions on Software Engineering*, Volume SE-21, No. 4, pp. 288-301 (April 1995)

[8] P. Hudak, N. D. Jones (eds.), "Partial Evaluation and Semantics-Based Program Manipulation", *SIGPLAN Notices*, Volume 26, No. 9 (1991)

[9] N. D. Jones, "Partial Evaluation", *Computing Surveys*, Volume 28, No. 3 (September 1996)

[10] N. D. Jones, C. K. Gomard, P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall (1993)

[11] Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow, *The Art of the Meta-Object Protocol*, MIT Press (1991)

[12] Christopher Landauer, Kirstie L. Bellman, "The Role of Self-Referential Logics in a Software Architecture Using Wrappings", *Proceedings of ISS '93: the 3rd Irvine Software Symposium*, 30 April 1993, U. C. Irvine, California (1993)

[13] Christopher Landauer, Kirstie L. Bellman, "The Organization and Active Processing of Meta-Knowledge for Large-Scale Dynamic Integration", pp. 149-160 in *Proceedings 10th IEEE International Symposium on Intelligent Control, Workshop on Architectures for Semiotic Modeling and Situation Analysis in Large Complex Systems*, 27-30 August 1995, Monterey (August 1995)

[14] Christopher Landauer, Kirstie L. Bellman, "Knowledge-Based Integration Infrastructure for Complex Systems", *International Journal of Intelligent Control and Systems*, Volume 1, No. 1, pp. 133-153 (1996)

[15] Christopher Landauer, Kirstie L. Bellman, "Integration Systems and Interaction Spaces", pp. 161-178 in *Proceedings of the First International Workshop on Frontiers of Combining Systems*, 26-29 March 1995, Munich (March 1996)

[16] Christopher Landauer, Kirstie L. Bellman, "Constructed Complex Systems: Issues, Architectures and Wrappings", pp. 233-238 in *Proceedings EMCSR 96: Thirteenth European Meeting on Cybernetics and Systems Research, Symposium on Complex Systems Analysis and Design*, 9-12 April 1996, Vienna (April 1996)

[17] Christopher Landauer, Kirstie L. Bellman, "Wrappings for Software Development", pp. 420-429 in *31st Hawaii Conference on System Sciences, Volume III: Emerging Technologies*, 6-9 January 1998, Kona, Hawaii (1998)

[18] Christopher Landauer, Kirstie L. Bellman, "Problem Posing Interpretation of Programming Languages", (to appear) in *32nd Hawaii Conference on System Sciences*, 5-8 January 1999, Maui, Hawaii (1999)

[19] Christopher Landauer, Kirstie L. Bellman, April Gillam, "Software Infrastructure for System Engineering Support", *Proceedings AAAI '93 Workshop on Artificial Intelligence for Software Engineering*, 12 July 1993, Washington, D.C. (1993)

[20] Mary Shaw, David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall (1996)

[21] Mary Shaw, William A. Wulf, "Tyrannical Languages *still* Preempt System Design", pp. 200-211 in *Proceedings ICCL'92: The 1992 International Conference on Computer Languages*, 20-23 April 1992, Oakland, California (1992); includes and comments on Mary Shaw, William A. Wulf, "Toward Relaxing Assumptions in Languages and their Implementations", *ACM SIGPLAN Notices*, Volume 15, No. 3, pp. 45-51 (March 1980)

[22] Leon Sterling, Ehud Shapiro, *The Art of Prolog*, MIT (1986)