JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 22, 337-356 (2006)

Process and Planning Support for Iterative Porting*

PRADEEP VARMA
IBM India Research Laboratory
Block 1, Indian Institute of Technology
Hauz Khas, New Delhi 110016, India
E-mail: pvarma@in.ibm.com

We present a general framework for software maintenance activities such as porting. The framework comprises a software engineering process with tool-based planning support. The process is designed to be iterative, with individual iterations identifying and fixing distinct porting issues and building and testing of software taking place regularly through the iterations. Overall planning of the iterations is formalized as path- planning problem in an abstract Cartesian space of program versions characterized by dialect variables. An optimal solution to the problem is derived based on its translation to a restricted Integer Linear Program capturing the problem constraints as a set of simultaneous linear equations. The solution allows individual dialect vertices to be visited more than once, but not the same edge, thereby ruling out looping behaviour. Problems without solution are identified as infeasible. Reliability of the software engineering process is enhanced by the ability to characterize program testability in different parts of the Cartesian space and to guide planned migration through more testable spaces while identifying the corresponding (effort) tradeoffs. Iteration planning uses the overall plan and related post-processor support to allow user latitude in local decisions and in fine-tuning the overall plan.

Keywords: reliable software engineering process, iterative software maintenance, software porting, planning, testability, analyze, fix, test, debug, quality assurance

1. INTRODUCTION

One of the most common software maintenance activities is software porting. Little attention however has been paid to deriving a detailed process and planning support for it. In a preliminary work of ours presented at the Software Engineering track of the ACM Symposium on Applied Computing, 2005, [21], we had described how eXtreme Programming (XP) practices could provide a detailed process for quality porting. In this paper, we focus on the iterative aspects of the software engineering process and present a general and detailed, tool-based planning framework for it. We crystallize the notion of migration planning as a path planning problem through a Cartesian space of program versions labeled by dialect variables. Since the Cartesian space is too large to encode or use directly in a program, developing a tractable abstraction for it is crucial, which we carry out in this paper, keeping in mind the need for the abstraction to allow a natural and general expression of the planning problem, relationships among different degrees of

Received July 1, 2005; accepted November 24, 2005.
Communicated by Sung Shin.
* The preliminary version of this paper was presented at the Software Engineering Track of the ACM Symposium on Applied Computing, Santa Fe, New Mexico, March 13-17, 2005.

freedom (synchrony constraints), notions of tool support in the problem space etc. Although the concrete notion as developed is specific to software migration/porting, the iterative process and planning problem is applicable to other software maintenance tasks where the program versions may be labeled differently than by language dialect variables. In order to ensure a reliable, high-quality software engineering process, our framework makes it possible to specify which regions of the problem space are more testable than others and to guide migration traversal through such spaces. In comparison to the preliminary work presented at SAC 2005, [21], the present work develops a more precise planning space abstraction (synchrony constraints, tool/testability support) and provides a general framework for the planning problem based on its translation to a restricted Integer Linear Programming (ILP) problem. In comparison to the earlier work, the present work provides an optimal solution for overall planning, as well as the ability to prove the infeasibility of a given problem, in case there is no solution at all for it due to inconsistent constraints.

We discuss porting in the exemplary context of source-to-source porting of C/C++ code-bases. This choice is dictated by commercial significance, as well as the hardness of the problem mandating the need for process and planning support. The languages' desired closeness to machine architecture, memory model (Endian issues, pointer issues/analysis) and concomitant philosophy ("Make it fast, even if it is not guaranteed to be portable", page 3, [19]) contribute to hardness. Longstanding evolution contributes its set of versions and deceptive problems like "quiet changes" [19]. Conformance definition creates a multitude of dialects by allowing vendor divergence on "implementation-defined behavior," "undefined behavior," "unspecified behavior," and "locale-specified behavior" (see pp. 487-513, Annex J, [10]).

Fig. 1 shows a sample C code containing a number of porting issues. We point out how the porting issues can be sorted according to transitions between C language dialects so that their execution can be organized as iterations from dialect settings to dialect settings.

```
    …
1   struct complex {double real, imaginary;} c = {0, 0};
2   extern float X[10];
3   union {int data; char bytes[4];} a;
    …
4           for (long i = 1; i < 10; i++)
5             {c.real = X[i]     //* divide */ i
6                 + c.real;};

7           c.imaginary = X[i];

8           a.data = c.real;
9           if (*((int *) &i) != ((int) i))     /* big endian */
10                 printf("a's msb: %d \n," a.bytes[0]);
11          else     /* little endian */
12                 printf("a's msb: %d \n," a.bytes[3]);
```

Fig. 1. Porting example.

As listed in the figure, the code fragment contains an old-style for loop and constructs from the ISO C90 standard (e.g. comment style /* … */). The code fragment is taken from a 32-bit, Little Endian machine, and is planned to be migrated to a modern C99 compatible setting, on a 64-bit, Big Endian platform. The migration can thus be viewed as a program conversion from one dialect of C to another, where the source dialect has settings: for-loop = old style, standard = C90, bits = 32, and Endian = Little. The target dialect has settings: standard = C99, bits = 64, Endian = Big. The knowledge of such dialect settings can be availed from the build flags used to compile a program (arguments to the cc command). The dialect variables involved in characterizing the porting problem are: for-loop, standard, bits, and Endian. The migration can be carried out variable by variable (e.g. four iterations with one variable changing from source to target setting in one iteration), or by doing more than one variable at a time. Each iteration's fixes can be succeeded by a testing phase in which the program is compiled and tested so that testing does not accumulate and build up as a biggest cost in the porting exercise. The decision regarding the specific set of iterations and their sequencing depends on code metrics, e.g. the specific porting issues found in the code. The specific porting issues in Fig. 1 are: line 1 – shift to C99's support for complex numbers for improvement; line 3 – potential Endian problem, depends upon union's use; line 4 – old style for statement with lexical scope beyond the loop body; lines 5-6 show a "quiet change" [19] with no static error manifestations but different run-time behavior – C99's C++ style comments specify $c.real = X[i] + c.real$ while the older dialect specifies $c.real = X[i]/i + c.real$. Line 7 references variable *i* declared in the old-style for scope of line 4, outside the loop body; line 8 is an implicit cast – a downcast from floating to integral with differing compiler warnings. Line 9's predicate is a faulty Endian platform detection test, which does not work if long and int have same size, which is the default in 32-bit platform. On the source dialect, the test works incorrectly, but still chooses the correct little Endian branch and on the target it correctly chooses the Big Endian branch, so the most significant byte of the union data is correctly identified and the code fragment is Endian safe (for this pair of dialects). Migration of this code can be carried out iteratively, one variable at a time, or more or less (e.g. change to 64 bit types can be broken further into separate long size shift, pointer size shift, and (optional) long double size shift). Sequencing among the iterations may be decided by the desire for more robustly tested code, such as shifting to Big Endian before changing from 32-bits to catch and fix the faulty Endian predicate on line 9.

2. RELATED WORK

Prior work on program analysis and transformation ([6, 17, 20], technical white-paper, www.reasoning.com), does not dwell upon process and process planning for software maintenance activities like porting. The support tooling described in our preliminary process and planning paper [21] shares [6]'s approach in utilizing a real-world front-end for multi-dialect source-code analysis (EDG in our case, www.edg.com).

Formal approaches pertinent to porting include the Strategy language [22] for specifying program transformation by traversals over an abstract syntax tree (AST). A strategy specification includes an AST definition as well as a traversal *strategy*, the com-

bination of which are then automatically converted into a source-to-source program transformer. The approach is suited to fully automatic program transformation only and not interactive remediation as needed in the real-world porting context of C/C++. [1] similarly argues for a separate specification of real-world code transformation for a multitude of languages and does not offer any software engineering process or planning support. By contrast, we focus on and delineate the migration domain for C/C++ programs and provide a tool-based, semi-automatic, eXtreme code transformation process delineating and leveraging the domain structure.

GNU Autotools (autoconf, automake, libtool), www.gnu.org, simplify adaptation of software package sources to different platforms, but do not support source-to-source transformation of package code. Autoconf generates shell scripts to match package needs with (discovered) platform capabilities. Automake encourages manual build abstraction into into a higher-level specification (Makefile.am) from which platform-specific makefiles can be automatically generated. Libtool simplifies building shared objects (dynamically linked libraries).

Our work addresses the following XP practices the most: planning game, testing, refactoring, daily build, small releases, simple design, and pair programming. XP is recommended for small teams, with scalability limitations as in [4]. We believe that our tool-based porting process can surmount these limitations as discussed in [21]. Large refactorings, identified as a "Bad Smell" in [7] can be carefully monitored based on the tool generated issues, metrics and issue kinds in our porting process. We expect that our tools will reify successful practices in our XP porting process, thereby deriving stability [13]. Compared to Extreme Modeling [3] which adds executable, testable models as design/documentation to the XP process, we add tool-generated porting issues and remediation strategies to the same. With reification benefit [13], reduction of bad smell potential, and scalability potential, we expect our process to provide significant economic, throughput, & latency gain for enterprise-scale porting.

3. A TOOLS-BASED ITERATIVE PORTING PROCESS

Given the C context (similarly C++) [10], porting can be formally defined as:

> *Porting is a meaning-preserving source-to-source transformation that rewrites a program written in a* (*conforming*) *C dialect into its strictly conforming subset.*

Although academic, the above definition motivates the basic step of iterative methodology: Starting from an initial configuration comprising a C/C++ dialect (including the OS, hardware and libraries-related flags), choose the next dialect to migrate to based on work size and rebuild considerations. As discussed in the section on the planning tool (called *orchestrator*) later, the space of C/C++ programs is organized into a Cartesian space of language dialects among which the orchestrator does planning by projecting work requirements for <from, to> dialect choices. A porting project comprises iterations starting from the source platform dialect settings, moves through intermediate dialect settings and finally ends in the target platform settings. Each iteration begins with the work plan and tests generation, and goes on to code remediation/refactoring, unit testing,

and rebuild and (optional) integration testing. Overall planning creates a rough roadmap of the entire migration (an overall "best", partially ordered set of choices), and iteration planning fine tunes it by making concrete, local choices (a total order), one at a time. Besides identifying the (say n) choices, the orchestrator helps incrementally choose one out of the (worst-case n!) orderings and assists each individual step of the port. Fig. 2 shows our process. In Fig. 2, the same compiler settings can run through several consecutive iterations. Unit compile may be followed by unit tests, before the next iteration. Any errors found in compile/test are fixed within the testing phase, prior to the next iteration.

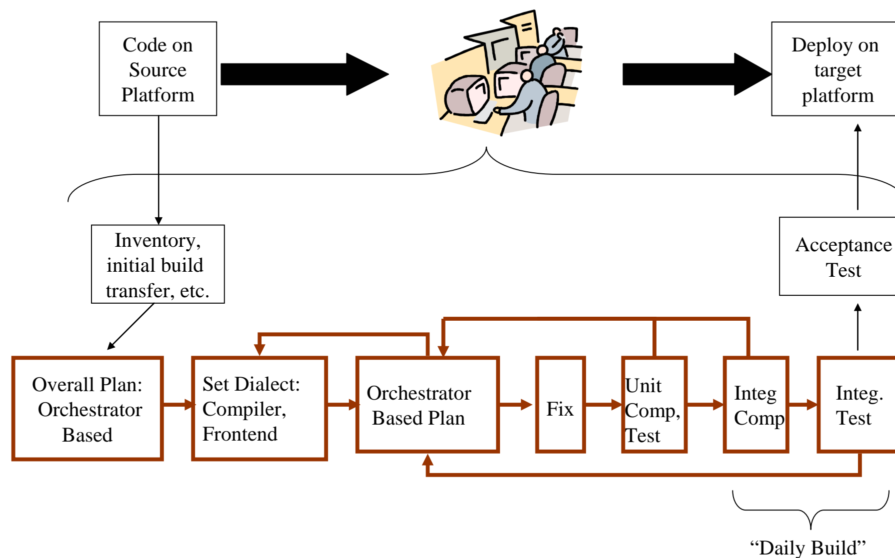

Fig. 2. NextGen eXtreme porting.

After dialect settings choice, the orchestrator organizes XP stories (and tasks) in terms of compilation units/files, each backed up with a detailed work plan comprising of (largely) automatically-detected potential/exact porting issues per unit. Typically a compilation unit is not broken across stories, unless the unit is very large. Individual team members can pick stories based on manifest porting issues, and indeed generate white-box- and (optionally) black-box-based test cases and a fine tuned time estimate based on the same.

Although team member mobility among files to a degree is fine, continuity with a compilation unit reduces work time due to increased familiarity. Since testing is not exhaustive, increased mobility offered by pair programming maintains continuity while increasing quality. Depending upon the extent of automated support available for the kind of porting issues in a particular story, the extent of pairing can be decided. As noted in [14], pairs are most valuable in harder problems with larger learning and design considerations such as Endian problems, whose automatic, general detection is an undecidable problem. Experts with a high skill set are best left mobile, for assisting difficult

cases on need. Thus the economic and manpower-availability considerations [16], and knowledge transfer can all be realized using discontinuous pair programming [11] in our context.

Test design is best carried out using whitebox and structural techniques [15, 24] to exercise the code around each porting issue. Integration/functionality testing may not be feasible in all iterations due to the lack of (any) customer-provided tests, and the unavailability of library support for a platform setting. As determined in [8] in the context of eXtreme practices (pair programming) and at a large scale in [15], test-based design and development leads to better software quality. Lack of customer tests can be ameliorated by the use of team-generated test cases. It is important to use test cases which reflect real-life software use patterns [23]. This is easier for migration than new code development since usage information for the pre-existing sources can be collected.

Testing and debugging support developed by us leverages the Eclipse environment's C/C++ Development Tools (CDT), www.eclipse.org, to integrate to standard compilers like GCC to compile the specific dialects needed at the <from, to> platform settings as discussed in [21]. Daily build is best done as daily as possible [12, 15]. Thus dedicating a (make) expert with the role of build engineer is preferred. Build upgradation (makefile remediation) may be assisted by the orchestrator's choice and enumeration of <from-to> settings per compilation unit. Similarly, remediation automation assists in successful integration by reducing inconsistencies and mismatches [12].

4. OVERALL AND ITERATION PLANNING SUPPORT

As noticed in [9, 10, 19], each choice of compiler explicit/implicit flag settings can be considered as a distinct dialect of the C/C++ language. The hundreds of settings thus can be classified as either pertaining to extension features (variables $E1$ to En), or parameters/qualifiers ($Q1$ to Qm) in the following. C (similarly C++) can be viewed as a parameterized family of languages, with each parameter choice (e.g. size of char, int, short) choosing specific members of the family. An extension of the language on the other hand adds (i.e. unions) a new set of programs, manifesting the extension feature over a common base. Indeed, the set of programs represented by a dialect comprising extensions $E1, \ldots, En$ and parameters $Q1, \ldots, Qm$ in a C base language (similarly C++) can be denoted as:

$$\mathcal{D} « \text{Dialect}(E1, \ldots, En, \boldsymbol{C}, Q1, \ldots, Qm) » = \\ (\cup_{E \in \text{Pow}\{E1, \ldots, En\}} \mathcal{E} «C, E») \cap \mathcal{P} «Q1» \ldots \cap \mathcal{P} «Qm» \quad (1)$$

where $\mathcal{D}$ « … » is the denotation map supported by $\mathcal{E}$ and $\mathcal{P}$ functions. $\mathcal{P}$ maps a parameter to the set of all possible C dialects with the given parameter fixed. Extensions are enumerated in all combinations by generating their powerset (Pow), followed by mapping a combination and the C base to its programs set. Union over all the extension combination sets followed by intersection with the parameters' denotation yields the dialect's meaning.

Migrating codebases is a multi-file problem (file × dialect × dialect)*, of which for convenience, we only describe the single file (with includes) problem in the present sec-

tion. In general, it is best to identify the minimum effort steps per file by which migration can take place. Once such data is available at the finest resolution, coarsening the same into "day-long" team efforts can be done either intra file or inter file. With this context, we explore now the general single-file (with includes) problem.

4.1 The Full Problem Graph

A given *E* or *Q* is actually a *K*-valued variable, with *K* usually two (Boolean). The presence of an *E*/*Q* in a dialect as described above formula (1) reflects a non-default active value for the variable (e.g. extensions on, as opposed to off). Consider each dialect in an extended form, enumerating the space of all possible *E* and *Q* variable combinations with corresponding (mostly default/off) values. Consider the space of all feasible dialects as a directed graph whose vertices are individual dialects, and edges migration steps from one dialect to another. Migration from one dialect (vertex) to another comprises choosing one of many paths in the graph from a source dialect to the target, based on attributes such as minimum effort, small even-sized effort steps, daily build feasibility at individual steps etc. Size attributes are migration project specific – the effort is sized vis-à-vis porting issues contained in the code-to-be-migrated along an edge. Build and test attributes are informed by the presence or absence of compiler, hardware, run-time library support at a dialect vertex. Fig. 3 shows an example of a full graph illustrating the migration space for dialects created using 4 Boolean variables.

Our source-to-source transformation toolkit is frontend driven (and supports a broad set of dialects) and is not full C/C++ compiler based. Test support on the other hand requires full compiler support for a given dialect, the availability of which along with supporting hardware cannot be assumed in general. Run-time libraries, if available, may require remapping library calls to different functions, which adds to the work effort estimate for a given step. Availability of run-time libraries in portable source form (e.g. GCC libraries), provides the easiest, zero-remap-effort answer, if available. When libraries simply aren't available, stubs, comprising of cache-functions frontend and a (distributed) library-process backend can (often) provide reasonable test support, as described in [21].

4.2 The Abstracted Problem

The combinatorial explosion in enumerating all *E* and *Q* tuples for the migration space makes a direct use of the space intractable. We seek to collapse the graph into a much smaller abstraction that covers all the desirable details. For vertices of the graph, the abstraction comprises a (linear) separate/un-combined enumeration of the *E* and *Q* variable value pairs along with the constraints on combining them to form valid dialects. The constraints we support are implications (e.g. $E_1 \Rightarrow E_2$). By choosing one value per variable from the abstraction we can resolve individual vertices of the full graph, with implications checking validity of the same.

Abstract (directed) edges in our representation enumerate only intra-variable value changes. Thus they are straightforward to represent in terms of the vertex abstraction mentioned above. An edge can optionally specify a synchrony constraint with it identifying a set of other edges that it is synchronous with. The constraint is a requirement that

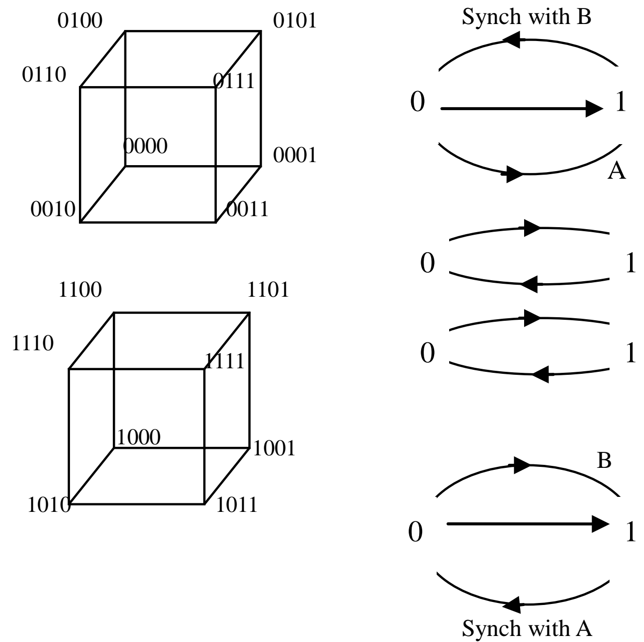

Fig. 3. A full graph and an abstraction for a 4-Boolean variables domain. Variable values are shown as binary bit patterns in full graph (cubes). In abstraction (right), most-significant bit is laid at top, and lsb at bottom. Full graph shows merged directed edges as bi-directional edges for brevity and shows only an edge subset. The synchronized abstract edges ensure that two concerned variables never get turned off together, in a migration process, although (using other edges) each can be turned on independently of the other.

a change of the variable along the edge be accompanied by a change along one of the edges in its synchrony set. Motivation for synchronized edges comes from the need to migrate some language features directly into others without an intermediate step of shifting the program into a featureless base language (e.g. from one concurrency/communication extension/library to another without sequentializing in-between). A synchrony constraint therefore allows specification of alternative packages from which one can be chosen to be switched ON, in conjunction with the one being switched OFF. Fig. 3 shows our abstract graph representation for an illustrative 4-Boolean-variables full graph.

An edge can have more than one synchrony constraints associated with it. Edges belonging to a synchrony set are reserved solely for the synchronous action specified by the synchrony constraint. Such edges are therefore *subordinate* or dedicated to the synchrony specifying edge and do not get executed/traversed except in synchrony with the specifying edge. This allows precise specification of information pertinent to pairs of synchronized edges, e.g. by annotation of cost weights to the synchrony set members as opposed to one cost for all members annotated to the synchrony-specifying edge. A subordinate edge can itself specify a synchrony constraint, thereby building up a tree of subordinates. All intra-variable changes are enumerated by explicit edges. There can be multiple edges for the same source and target abstract vertices, so long as each edge specifies different synchrony constraints, or is subordinate to a different synchrony constraint. A

synchronized edge can represent any edge in the full graph, by capturing all the variables changed by the full-graph edge in synchrony constraints.

Tool support, defined as vertex attributes in the full graph shifts to a Boolean expression in the abstract specification. Literal terms of the expression are of the form (var == value), where var is a dialect variable and the literals are combined using standard $\wedge$, $\vee$, $\neg$ (and, or, not) operators to make up a logical formula describing which dialects have tool support or not. The literal terms are the same as the dialect variable vertices contained in the abstract graph. A specific language dialect, a vertex in the full graph, specifies values for all dialect variables and hence the truth value of individual literal terms. Using this, the Boolean expression can be reduced to the truth value of tool support for the dialect or lack thereof. As an example of tool support formula, the following states that tool support is available for all dialects on Big Endian hardware platforms: $\neg$(Endian == LittleEndian).

Effort attributes annotate abstract edges, just as they annotate full-graph edges. This assumes that all the concrete, full-graph edges represented by an abstract edge have the same effort attribute as the abstract edge itself. This assumption is reasonable as a first-order approximation for the migration domain wherein: the bulk of the transformed code remains unchanged, analysis cost is tied to the bulk code, and direct interference among changes is a minority. Since test/debug support at individual vertices is not a given, effort estimates exclude these activities. All effort weights are positive quantities. A dialect shift along a synchronized edge accrues an effort equal to the sum of the weights of the chosen individual edges from its (recursive) synchrony sets (one per set). Note that computing effort attribute is a project-specific exercise, involving tool-based source code analysis. Containing abstract edges to one degree of freedom (i.e. one variable – its value changes) contains the number of analyzers required to linear in the number of variables in the system (the from-to value combinations for the variable can be analyzer arguments).

Only overall orchestration (Fig. 2) relies on the effort attributes of abstract edges in global planning. In iteration planning, the abstract edge estimate is further refined to its specific concrete context (code has already migrated along earlier iterations/edges). This includes detailed fine-tuned analyses and test-related costs. Overall orchestration need not obsess about estimation accuracy so long as the relative size of all estimates vis-à-vis each other is stable. Hence standard, weighted, metrics- and metric-functions-based analyses can be used in it.

The objectives of planning are multifaceted as follows:

1. Minimize overall effort.
2. Maximize integrated tests in the path and ensure that they evenly space out analyze & fix efforts.
3. Maximize integrated builds in the path and ensure that they evenly space out analyze & fix efforts.
4. Maximize unit tests in the path and ensure that they evenly space out analyze & fix efforts.
5. Maximize unit builds in the path and ensure that they evenly space out analyze & fix efforts.

Goals 2-5 are about regular compile-time and run-time testing and the need to maximize their frequency. Testing at irregular intervals is unproductive since the longer (untested) intervals may scale testing costs super-linearly, while the short intervals may only add the testing overhead without much benefit from the activity.

4.3 An Optimal Solution to Overall Planning

We solve the overall planning problem by translating it into a restricted Integer Linear Programming (ILP) problem, for which we provide an optimal solution.

4.3.1 Reducible constraint equations

Goal 1 above (minimize overall effort) translates into an objective function whose value is sought to be minimized, subject to (a) *path constraints* that the migration process define a valid path from the source to the target dialect, and (b) that synchrony constraints among the edges be satisfied in the migration process.

We define Boolean (i.e. 0/1 valued) variables for the directed edges available in the abstract graph. Edge variables for the ith variable are enumerated as e_{ij}. Each value vertex for the variable is represented by a constant v_{im}, where m ranges from 1 to K, for the K values of the given variable. Path constraints yield equations of the following form:

$$v_{is} + c_{i1}e_{i1} + c_{i2}e_{i2} + \ldots + c_{in}e_{in} = v_{if} \quad (2)$$

where the edge variables associated with the ith variable total n, v_{is} is a value representing the starting vertex of the migration process for the ith variable, v_{if} represents the final vertex for the ith variable, and $c_{ij} = v_{im} - v_{il}$, where edge e_{ij} is directed from the lth value to the mth value.

Let an *edge assignment* be the set of edges whose variables are assigned a value 1 (one) in a path constraint's equation. An edge assignment that forms a continuous path from the source vertex to the target vertex comprises a solution to the path constraint equation. Additionally, the solution may have cycles that do not touch the continuous path at any vertex, as the example in Fig. 4 shows. These additional cycles are taken care of in the ranking of solutions according to the objective function later. Ensuring the converse for a path constraint, that every solution necessarily comprises a *continuous* path (plus disconnected cycles) from the source dialect to the target requires careful choice of the vertex constants v_{ij}. Letting $v_{ij} = (n + 2)^j$, where n is the total number of edges as given in the above equation, and j ranges from 1 to K (for the K vertices) suffices to ensure this. Briefly, the insight for this choice is that each vertex is being assigned a distinct bit in a large $(n + 2)$-ary radix number system. Since the above equation comprises at most $n + 1$ vertex value additions and n vertex value deletions, dedicating distinct bits to the values ensures that additions and deletions of values distinct from v_{ij} do not affect the bit space reserved for v_{ij}. Formally, a proof of this is as follows:

Theorem 1 An edge assignment comprises a path constraint equation's solution if and only if it specifies a continuous path and disconnected cycles from the source vertex to the target vertex.

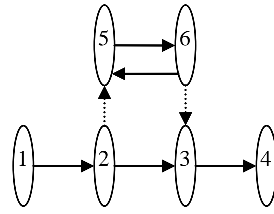

Fig. 4. A set of vertices and edges for one variable in an abstract graph, with source vertex being 1 and target vertex being 4. The whole line edges (and not dashed line edges) belong to the edge assignment for a solution of the path constraints equation. The edge assignment contains one continuous path from vertex 1 to 4, and one disconnected cycle between vertices 5 and 6. The disconnected cycle is not screened by the path constraint equation because the contributions of its member edges cancel each other out.

Proof: (*if case*): Let v_{is}, $v_{i\alpha}$, $v_{i\beta}$, $v_{i\gamma}$, …, $v_{i\phi}$, v_{if} be the ordered sequence of vertices visited on a continuous path from the source to target. Substituting the 0/1 values for edge variables from the edge assignment, and reordering the left hand side (LHS) of the path constraint Eq. (2) above so that edge variables for the edges in the continuous path are listed in the order in which they are traversed, the LHS reduces to: $v_{is} + (v_{i\alpha} - v_{is}) + (v_{i\beta} - v_{i\alpha}) + (v_{i\gamma} - v_{i\beta}) + \ldots + (v_{if} - v_{i\phi}) + cycle_1 + cycle_2 + \ldots + cycle_n = (v_{is} - v_{is}) + (v_{i\alpha} - v_{i\alpha}) + (v_{i\beta} - v_{i\beta}) + \ldots + (v_{i\phi} - v_{i\phi}) + v_{if} = v_{if} =$ RHS, the right hand side. The edges for each cycle sum to zero, as the vertices cancel each other out within the cycle itself.

(*only if case*): For any edge assignment, LHS $= v_{is} + (v_{i\alpha} - v_{i\alpha'}) + (v_{i\beta} - v_{i\beta'}) + (v_{i\gamma} - v_{i\gamma'}) + \ldots + (v_{i\phi} - v_{i\phi'})$ = RHS $= v_{if}$, for some vertices $v_{i\alpha'}$, $v_{i\beta'}$, etc. Suppose in this summation of edge weights, each negative term can be cancelled by precisely one positive term. Also suppose that $v_{is} \neq v_{if}$. First, the summation can be reduced by removing the zero-contrib- uting cycles from it. Then, since LHS = RHS = v_{if}, one of the remaining positive edge terms has to be v_{if} and one of the remaining negative edge terms has to cancel out the initial v_{is} term in LHS. The summation can be rearranged to place the v_{is} canceling edge after v_{is}. This is succeeded by an edge whose negative term cancels out the positive term of the present edge and so on. The summation thus reduces to a cycle-free version of the form given in the previous proof above, viz. $v_{is} + (v_{i\alpha} - v_{is}) + (v_{i\beta} - v_{i\alpha}) + (v_{i\gamma} - v_{i\beta}) + \ldots + (v_{if} - v_{i\phi})$. This identifies a cycle-free continuous path from the source to the target vertex. If the cycles removed previously are added to this path, the path relaxes to continuous path and disconnected cycles. If we remove the supposition that $v_{is} \neq v_{if}$, then all the edge terms comprise cycles, which may or may not be connected to each other. This comprises a possibly trivial (null) continuous path from the source to target plus disconnected cycles. In summary, the supposition that each negative term can be cancelled by precisely one positive term implies that a continuous path plus disconnected cycles exists from source to target and hence is a solution of the path constraints equation (from the if-case proof above).

Suppose next that an edge assignment that is not a continuous path with disconnected cycles is a solution of the path constraints equation. For such a case, we have from above that each negative term cannot be cancelled by precisely one positive term in the LHS summation. The LHS - RHS has the general form: $(v_{is} - v_{if}) + (v_{i\alpha} - v_{i\alpha'}) + (v_{i\beta} -$

$v_{i\beta'}) + (v_{i\gamma} - v_{i\gamma'}) + \ldots + (v_{i\phi} - v_{i\phi'})$. From above, we have that not all negative terms due to edges can be cancelled one-to-one by positive terms from elsewhere. Of the n edge terms, suppose m terms can be cancelled thus ($n > m$). Then LHS – RHS reduces to the form $av_{i\alpha} + bv_{i\beta} + cv_{i\gamma} + \ldots - a'v_{i\alpha'} - b'v_{i\beta'} - c'v_{i\gamma'} - \ldots$, where the positive constants a, b, c … sum to $n + 1 - m > 0$ and so do the negative constants a', b', c', …. The 1 in $n + 1 - m$ comes from the $(v_{is} - v_{if})$ term, which is not edge related. Since $n > m$, at least one positive term and one negative term exist in LHS – RHS. In this form, each of the vertices represented by $v_{i\alpha}$, $v_{i\beta}$, $v_{i\alpha'}$, $v_{i\beta'}$ etc. is distinct. The constants a, b, c, a', b', c', etc. are all positive integers less than or equal to $n + 1 - m$. Since m can be zero, each of the constants is bounded by $n + 1$. Substituting the values for the vertex constants $v_{i\alpha}$, $v_{i\beta}$, etc., we get LHS – RHS = $a(n + 2)^{\alpha} + b(n + 2)^{\beta} + c(n + 2)^{\gamma} + \ldots - a'(n + 2)^{\alpha'} - b'(n + 2)^{\beta'} - c'(n + 2)^{\gamma'} - \ldots$. In an $n + 2$ radix number system, the positive terms add to a number comprising of digits a, b, c … and the negative terms add to a number comprising digits a', b', c', …. Each of the digits is in a distinct position, different from the others. Thus the number represented by the positive terms is distinct from the number represented by the negative terms and hence LHS – RHS ≠ 0. This contradicts the supposition that the edge assignment is a solution of the path constraints equation. ❑

A synchrony constraint that e_{ij} be used if and only if any one of e_{ab}, e_{cd}, e_{fg}, etc. is used is modeled by an equation of the form:

$$e_{ij} = e_{ab} + e_{cd} + e_{fg} \ldots \quad (3)$$

A more relaxed model for synchrony, which mandates the use of one or more among e_{ab}, e_{cd}, e_{fg}, …, in conjunction with e_{ij} or otherwise is as given below. We eschew this as unnecessary, especially since it makes the ILP optimization a harder problem.

$$e_{ij} \leq e_{ab} + e_{cd} + e_{fg} \ldots \quad (4)$$

A synchrony constraint, as modeled above captures an OR relationship among edges. An AND relationship is specified by specifying multiple synchrony constraints for an edge simultaneously, which results in a set of simultaneous equations, one for each of the constraints.

4.3.2 Objective function

Synchrony and path constraints lead to a set of simultaneous linear equations that need to be solved while minimizing an overall effort objective. This takes the form:

$$\text{Minimize} \sum w_{ij}\, e_{ij} \quad (5)$$

where i, j range over all edges and w_{ij} is the traversal cost of a given edge. Note that while all edge weights are non-negative, some of them can also be zero, reflecting the edges' synchrony-specifying status. The weight clubbed with such edges accrues from the weights associated with their recursive subordinates.

The simultaneous equations reduce the number of free edge variables in the above objective function. In a system of n variables and m equations, ($n > m$), the number of free variables can be reduced to $n – m$ if the equations are independent of each other (not a combination of each other). The equations can also be infeasible (contradictory), in which case the problem is straightforward to report to the user. If $n = m$, the free variables may all be eliminated identifying one solution exactly. Such a system may be written in matrix form as $W_{n\times n} E_{n\times 1} = C_{n\times 1}$, with the solution being computed as $E_{n\times 1} = W_{n\times n}^{-1} C_{n\times 1}$, if the inverse matrix W^{-1} exists. Otherwise, the equations result in m variables being expressed in terms of the $n – m$ free variables, each of which can take two values 0/1. Substitution of the re-expressed variables in the objective function retains the objective function's linear form. The optimal solution is then obtained by minimizing each term in the revised objective function independently, thereby obtaining the value for each of the free variables in time proportional to their number. Next the re-expressed variables can be computed in terms of the free variables.

Note that since path constraints allow extraneous (disconnected) cycles to pass through, it is in the step of optimizing the objective function that such cycles get eliminated first. This occurs since each cycle adds its positive edge weights to the objective, which the minimizing process would seek to avoid. Such screening is not complete however, and complete elimination occurs in a post-processing step discussed later.

It is straightforward to rank the space of potential solutions generated by free variables in terms of the objective function value, which may be needed as higher-ranked solutions may get eliminated for reasons such as lingering disconnected cycles. This lets the final solution be the highest ranked, feasible one, which makes it an optimal solution. A method to generate solutions in rank order comprises generating ranked solutions when (a) one variable changes from the highest rank solution, (b) two variables change from the highest rank solution, (c) three variables change from the highest rank solution, and so on. These solution sequences are generated on demand, and interleaved to yield one totally ranked solution sequence. For a given number of changed variables, generating the ranked sequence is straightforward, given the linear form of the objective function.

4.3.3 Post-processing a candidate solution

4.3.3.1 Generating intra-variable path candidates

Once the solution for edge variables is obtained, intra-dialect variable migration paths have to be identified. Each variable's path has to cover all the edges contained in the edge assignment for the variable's solution. The algorithm for identifying all valid paths for a dialect variable is as follows: Keep around as global variables, an under-construction results tree, three sets (visited edges, unvisited edges, and tree nodes requiring further exploration), and a current tree node. In the following, each tree node contains a copy of an abstract graph vertex, which is referred to simply as the vertex, instead of the vertex copy. Working with copies allows the tree to be built with a multiset of graph vertices, which is needed since a path through the graph may visit a vertex more than once. Initially, let the results tree be simply a root node comprising (a copy of) the start vertex, the current node be this root node, unvisited edges be the intra-variable solu-

tion's edge assignment, and the remaining sets be null. Next, carry out the following steps:

1. For the current node's vertex, remove one outgoing edge present in the unvisited edges, shift it to visited edges and jump to the target vertex of the edge. Create a tree node containing the jumped to vertex as the new current node. Store also the edge used to make the jump in the tree node, for interpreting the tree for results after this algorithm is over.
2. Enhance the result tree by making the new current node a child of the earlier current node. Annotate the earlier current node in the tree with the visited and unvisited edge sets prior to executing the edge shift.
3. Continue with this by going back to step 1 till either the unvisited edges set become empty or no unvisited outgoing edges remain for the current node.
4. Next, from the current node, go backwards up the path to the result tree's root looking for nodes from which alternative traversal paths exist. An alternative traversal path exists anytime a node is annotated with an unvisited edge set from which an alternative unvisited edge can be chosen as an outgoing edge for the node's graph vertex. Union these nodes to the set of tree nodes requiring further exploration.
5. Unless the set of tree nodes requiring further exploration is null, remove a node from the set to explore as the current node as follows. From the unvisited edge set annotating the removed node, remove any outgoing edge (that can be used to make a jump from the node's vertex) and shift it to the visited edge set annotating the node. Next, make these annotating visited/unvisited edge sets the global visited/unvisited edge sets. Next, jump to the target vertex of the shifted edge, making a new tree node containing the target (and the edge used to jump to it) as the new current tree node. Go to step 2.

Once the result tree has been constructed as above, each path from the root of the tree to a leaf is a valid intra-variable path, so long as it fully covers the variable's edge assignment. If no path covers all the edges, then the path constraint's solution contains disconnected cycles in it and must be discarded. Fig. 5 shows an example of computing valid intra-variable paths. Of the five alternative paths generated for the example, only two are valid and cover the edge assignment for the solution.

4.3.3.2 Generating global path candidates

Synchrony constraints disallow a valid intra-variable path to be traversed freely, asynchronously. For a synchrony-specifying edge *e*, any (recursively) subordinate synchrony edge executes simultaneously with the edge, if at all. As the example in Fig. 6 shows, these synchronization requirements can cause deadlocks and make otherwise feasible paths, infeasible. In short, further screening for synchronized schedule-ability of intra-variable paths has to be carried out prior to declaring a solution of the constraints equations as a globally feasible one.

Note that the recursive synchrony sets are all available directly from the constraints in the abstract graph and all edge members of these sets can be assigned a shared, global time when they will execute, if at all. Consider a traversal along any intra-variable

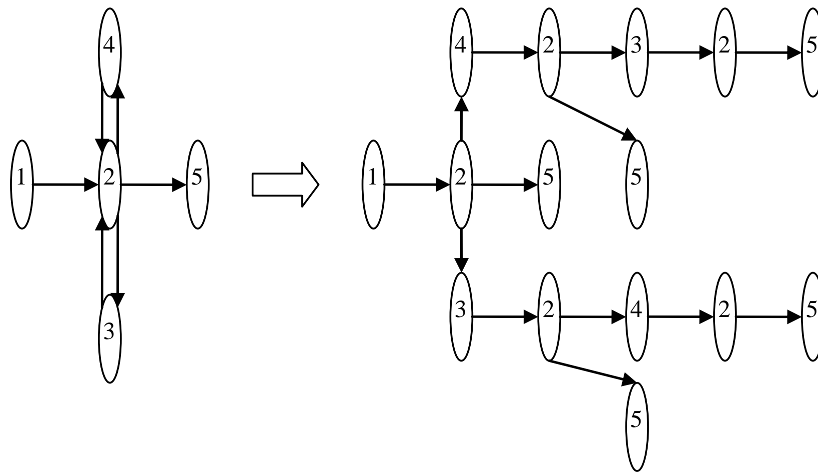

Fig. 5. The graph on the left shows a subset of an abstract graph for one dialect variable, containing only edges assigned a value 1 in the path constraints solution. The solution allows for multiple paths from the start vertex 1 to the target vertex 5. The tree on the right unfolds the path possibilities, with the two longest paths (from tree root to leaf nodes) being the only valid intra-variable paths for this example.

migration path. For the synchronized edges encountered in the path, a *temporal* constraint must be satisfied that the later edges have a later shared global timestamp than the earlier encountered edges. This constraint can be encoded as a set of inequations $t_a < t_b < t_c < t_d \ldots$, for the shared timestamps encountered in an intra-variable path. Consider one path obtained for each dialect variable. If the set of temporal relations obtained for this set of paths are consistent with each other, then the set of paths form a globally schedule-able solution for the total migration problem.

The method for establishing schedule-ability is to enumerate all path combinations for dialect variables and to test for temporal consistency in order to establish the presence of at least one schedule-able combination in the overall solution. For the simplest and common case of paths involving no more than one visit to a dialect vertex in any dialect-variable path, the total number of path combinations to be tested is only one, which keeps the checking requirements low in usual cases.

The test for temporal consistency checks whether a set of timestamps can exist which satisfy the temporal constraints defined for them. This is carried out using symbolic timestamps as follows: Each set of totally-ordered symbolic timestamps for individual dialect variable paths (i.e. $t_a < t_b < t_c < t_d \ldots$, where $<$ is the total temporal order) is represented as a directed graph (linear graph), where the directed edges represent the temporal order. Starting from one such graph, the others are combined with it one-by-one as follows. If a timestamp common to the combined graph (under construction) and the graph being merged is found, then the vertex is shared among the two graphs while the intra-graph edges are preserved. Similarly, all other such vertices are merged. If as a result of such merging, the combined graph becomes cyclic, then the temporal orders are inconsistent otherwise not. Fig. 7 illustrates this by showing how a cycle develops in the combined graph for the example of Fig. 6.

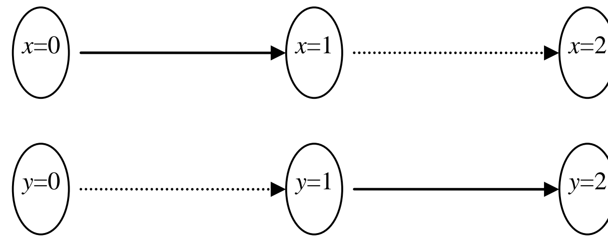

Fig. 6. The intra-variable paths for variables x and y are synchronized. The dashed edge between vertices $x = 1$ and $x = 2$ is synchronized with the dashed edge between vertices $y = 0$ and $y = 1$. The remaining two edges are synchronized with each other. The migration from $x = 0$, $y = 0$ to $x = 2$, $y = 2$ is deadlocked by the contradicting temporal requirements of these synchronization constraints.

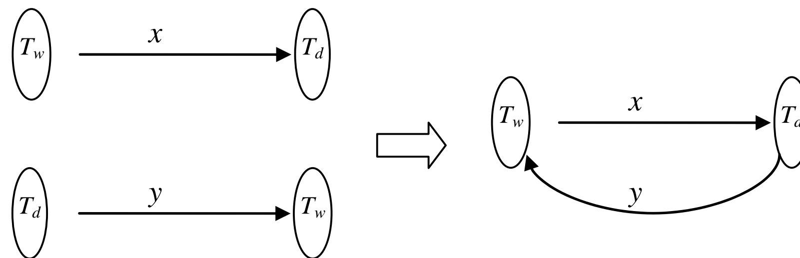

Fig. 7. The graphs on the left show the ordered symbolic edge timestamps for the example in Fig. 6. T_w is the timestamp for the whole line edges and T_d is the timestamp for the dashed line edges. The combined graph is shown on the right, wherein timestamp vertices are shared. The backward edge due to edge-preservation in the graph of variable y creates a cycle in the combined graph, which identifies that the temporal constraints for these paths cannot be satisfied.

If a given overall solution is found to not be schedule-able, an alternative solution of lesser objective function value has to be tried till a feasible solution is found or the space of solutions runs out. In the latter case, the user is informed that no solution exists for the given migration problem.

In a manner similar to the above, a timestamps graph can be created to test for satisfiability of implication constraints in a given solution. The timestamps identify when a given dialect variable vertex is populated in a migration path. Concurrent occupation of vertices from different variables defines the total dialect of the given program/file. Since implication constraints enforce or disallow variable combinations, the timestamp graph can establish whether the intermediate dialects held by a program on its migration path are valid or not. This is a straightforward extension of the method described above for synchrony constraints.

4.4 Testability Objectives

In the above framework, the ability to specify that the migration path traverse primarily through more testable space comes freely. As mentioned in an earlier example

(section 4.2), a testing environment (tool/libraries etc.) may be available only on Big Endian dialects, given by the tool support formula: ¬(Endian == LittleEndian). In such a scenario, it may be valuable to do the iterative porting primarily through the Big Endian space, even if both the source and target dialects of the migration exercise are on Little Endian platforms. A detour through the Big Endian space can be specified by adding a *testability constraint* that the edge from Endian = LittleEndian to Endian = BigEndian be in the edge assignment for the Endian variable's path constraint. The presence of this edge in the final solution enables the Endian variable to be set at the BigEndian vertex, which if carried out early enough ensures maximum tool support. Final scheduling of individual migration edges is discussed in the section on iteration planning. Regardless, for simple, or complex tool support logical formulae, it has to be determined which set of literal terms have to be made true and then edges capable of causing the literal terms to be true added as testability constraints to the edge assignment. Finally, the edge execution has to be scheduled carefully in order to obtain the desired tool coverage along the migration path.

The value of the objective function with and without testability constraints can be compared to determine the tradeoff in terms of effort increase due to forced traversal through more testable space. Naïve addition of testability constraints can result in the migration exercise becoming infeasible (contradictory). In such a case, some of the testability constraints have to be removed.

4.5 Iteration Planning and Regular Testing

The overall plan is presented to the user as a global edge assignment for all the migration variables. It is left to the user to determine which edge to execute when in the iterative porting process shown in Fig. 2. Every time the user makes a choice of edge(s) to traverse in an iteration, the decision can be automatically verified (using the post-processing steps of overall planning) as containing at least one feasible path for the succeeding migration. In short, while overall planning at best provides a partially ordered set of migration edges, iteration planning converts the same into a total order. A decision, whether to execute one edge in a sequence of iterations in Fig. 2, or to execute more than one edges in one iteration, is also taken during iteration planning. This decision is driven in part by the regular building and testing objectives (goals 2-5, section 4.2), wherein enough number of edge(s) are executed in a set of iteration(s) so as to regularly space out the building and testing efforts.

The (human) iteration planner may take into account more detailed information not available to the (automatic) overall planner such as the availability/schedule of the relevant human skills for a given step. Alignment of migration steps among different files for simpler makefile migration may be considered in this step. Local, detailed effort estimation analyses, based on actual issues, manual code inspection, test case development etc. feed this exercise. Finally, the need to iterate primarily through more testable space mandates that edge scheduling be carried out in an order which makes the relevant tool support available at the earliest and remain so as long as possible. For the Endian example discussed earlier, this means the shift to BigEndian platform occur preferably as the first edge, and the shift back to LittleEndian occur preferably as the last step. In between, depending on the tool support formula, the testability-related edges have to be juggled/

scheduled in a manner that retains the overall truth of the formula, despite changing support from sub-expressions of the formula.

4.6 Discussion: Expressiveness of the Constraints Framework

A synchrony constraint models an OR-like relationship between subordinate edges. Since a synchrony-specifying edge can specify multiple synchrony constraints, a conjunction of these disjunctive clauses can be specified. Since each subordinate can itself be synchrony specifying, each disjunctive clause term can itself be a conjunction of disjunctive clauses and so on. A synchrony-specifying edge can be a virtual edge, meant only as a placeholder for a conjunction of disjunctive clauses. A virtual edge is straightforwardly written as an edge from a vertex to itself. Term negation can be brought in simply using the power of the equational framework, wherein negation of a virtual/other edge y is simply $1 - y$, as in the equation $x = 1 - y$. Thus complex synchrony expressions can be specified by the synchrony language and its equational framework.

5. CONCLUSIONS

We have presented a general process and planning framework for software maintenance activities like porting. The framework allows a natural specification of porting requirements and dependencies and provides an optimal plan for migration of software iteratively from its source settings to the target settings. The plan and its use are carried out in an overall step and a sequence of iteration-specific steps, whereby the porting team can fine-tune the application of the overall plan depending upon the local context. Infeasible requirements or migration deadends (misguided fine-tuning) are caught automatically. Plan generation supports specification of testable migration spaces, the guiding of migration through such spaces and the computing the tradeoffs involved. The iterative process and planning support thus ensures more reliable and higher quality porting than the present state of the art.

REFERENCES

1. I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS: program transformations for practical scalable software evolution," in *Proceedings of the IEEE International Conference on Software Engineering* (*ICSE '04*), 2004, pp. 625-634.
2. K. Beck, *Extreme Programming Explained: Embrace Change*, Pearson Education, Reading, Singapore, 2000.
3. M. Boger, T. Baier, F. Wienberg, and W. Lamersdorf, "Extreme modeling," in *Proceedings of the International Conference on eXtreme Programming and Flexible Processes in Software Engineering* (*XP '00*), 2000, pp. 175-189.
4. R. Crocker, "The 5 reasons XP can't scale and what to do about them," in *Proceedings of the International Conference on eXtreme Programming and Flexible Processes in Software Engineering* (*XP '01*), 2001, pp. 62-65.
5. A. van Deursen, T. Kuipers, and L. Moonen, "Legacy to the extreme," in *Pro-*

ceedings of the International Conference on eXtreme Programming and Flexible Processes in Software Engineering (*XP '00*), 2000, pp. 501-514.
6. P. T. Devanbu, "GENOA – a customizable, front-end-retargetable source code analysis framework," *ACM Transactions on Software Engineering and Methodology*, Vol. 8, 1999, pp. 177-212.
7. A. Elssamadisy and G. Schalliol, "Recognizing and responding to 'Bad Smells' in extreme programming," in *Proceedings of the IEEE International Conference on Software Engineering* (*ICSE '02*), 2002, pp. 617-622.
8. B. George and L. Williams, "An initial investigation of test driven development in industry," in *Proceedings of the ACM Symposium on Applied Computing* (*SAC '03*), 2003, pp. 1135-1139.
9. ISO/IEC 14882:1998 C++ standard, 1998, http://www.iso.org.
10. ISO/IEC 9899:1999 C standard, 1999, Also, ISO/IEC 9899: 1999 C Technical Corrigendum, 2001, http://www.iso.org.
11. A. Janes, B. Russo, P. Zuliani, and G. Succi, "An empirical analysis on the discontinuous use of pair programming," in *Proceedings of the International Conference on eXtreme Programming and Agile Processes in Software Engineering* (*XP '03*), 2003, pp. 205-214.
12. M. Kajko-Mattsson, M. Jonson, S. Koroorian, and F. Westin, "Lesson learned from attempts to implement daily build," in *Proceedings of the European Conference on Software Maintenance and Reengineering* (*CSMR '04*), 2004, pp. 137-146.
13. M. Lippert, S. Roock, R. Tunkel, and H. Wolf, "Stabilizing the XP process using specialized tools," in *Proceedings of the International Conference on eXtreme Programming and Flexible Processes in Software Engineering* (*XP '01*), 2001, pp. 38-41.
14. K. M. Lui and K. C. C. Chan, "When does a pair outperform two individuals?" in *Proceedings of the International Conference on eXtreme Programming and Agile Processes in Software Engineering* (*XP '03*), 2003, pp. 225-233.
15. E. M. Maximilien and L. Williams, "Assessing test-driven development at IBM," in *Proceedings of the IEEE International Conference on Software Engineering* (*ICSE '03*), 2003, pp. 564-569.
16. M. M. Muller and F. Padberg, "On the economic evaluation of XP projects," in *Proceedings of the ACM International Symposium on Foundations of Software Engineering* (*FSE '03*), 2003, pp. 168-177.
17. G. C. Murphy, "Lightweight lexical source model extraction," *ACM Transactions on Software Engineering and Methodology*, Vol. 5, 1996, pp. 262-292.
18. C. J. Poole, T. Murphy, J. W. Huisman, and A. Higgins, "Extreme maintenance," in *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM '01*), 2001, pp. 301-311.
19. Rationale for C99 standardization, Version 5.1, 2003, http://wwwold.dkuug.dk/JTC1 /SC22/WG14/www/C99RationaleV5.10.pdf.
20. P. Varma, "Compile-time analyses and run-time support for a higher order, distributed data-structures based, parallel language," Ph.D. dissertation, Department of Computer Science, Yale University, U.S.A., 1995, University Microfilms International, Ann Arbor, Michigan, 1995.
21. P. Varma, A. Anand, D. P. Pazel, and B. R. Tibbitts, "NextGen EXtreme porting:

structured by automation," in *Proceedings of the ACM Symposium on Applied Computing* (*SAC*), 2005, pp. 1511-1517.
22. E. Visser, "Stratego: a language for program transformation based on rewriting strategies," in *Proceedings of Rewriting Techniques and Applications* (*RTA '01*), in A. Middledrop, editor, LNCS 2051, Springer-Verlag, 2001, pp. 357-361.
23. E. J. Weyuker, "Using operational distributions to judge testing progress," in *Proceedings of the ACM Symposium on Applied Computing* (*SAC '03*), 2003, pp. 1118-1122.
24. J. Yuan, M. Holcombe, and M. Gheorghe, "Where do unit tests come from?" in *Proceedings of the International Conference on eXtreme Programming and Agile Processes in Software Engineering* (*XP '03*), 2003, pp. 161-169.

Pradeep Varma received the B.Tech. degree in Electrical Engineering from Indian Institute of Technology, Delhi, India, in 1987, the M.S. and M.Phil. degrees in Computer Science from Yale University, New Haven, CT, U.S.A. in 1990, and the Ph.D. degree in Computer Science from Yale University, CT, in 1995. He is a research staff member at the India Research Laboratory of IBM Research in New Delhi, where he has been since the laboratory was founded in 1998. His areas of research include software lifecycle, circuits, and parallel and distributed computing. See http://www.research.ibm.com/people/v/varma/.