Globalization by Chunking: A Quantitative Approach

Audris Mockus, *Bell Labs*
David M. Weiss, *Avaya Laboratories*

Distributing development over many sites, often in different countries, can cause productivity-reducing coordination difficulties. This article introduces methods for assessing and minimizing coordination problems by identifying tightly coupled work items, or chunks, as candidates for independent development.

Because of economic, political, and practical needs, businesses regularly distribute their software production globally.[1] Participants at the different development sites often suffer inhibited communication and coordination because they are remote from each other. One result of the affected communication and coordination might be reduced productivity and an increased production interval.

In this article, we look for technical solutions to accommodate the business needs for distributed software development. In doing so, we investigate quantitative approaches to distributing work across geographic locations to minimize communication and synchronization needs. Our main premise is inspired by Melvin Conway's work, which suggests that a software product's structure reflects the organizational structure of the company that produced it,[2] and David Parnas's work suggesting software modularity should reflect the division of labor.[3] Here, we introduce ways to quantify the three-way interactions between an organization's reporting structure, its geographic distribution, and the structure of its source code. We based our analysis on records of *work items* (for this analysis, a work item is the assignment of developers to a task, usually to make changes to the software).

Work items

For software development to be most efficient, the organization's geographic distribution and reporting structure should match the division of work in software development. Tightly coupled work items that require frequent coordination and synchronization should be performed within one site and one organizational subdivision.

We try to identify such work items empirically by analyzing the changes made to software. When a set of work items all change the same set of code, we refer to the set of code as a *chunk*. We use the term *module* to mean a set of code contained in a directory of files, which follows the usage of the development projects whose software we analyzed.

Our main contribution is to define an analysis process for identifying candidate chunks for distributed development across several locations based on quantitative evidence. As part of this process, we have defined

0740-7459/01/$10.00 © 2001 IEEE

- a method to quantify the impact on development time and effort of work items spanning development sites;
- a method to identify work item-induced chunks in software systems;
- a process to identify chunks that could be developed independently in different organizations or in different development sites, including a way to define quantitative measures that describe chunks; and
- an algorithm to find chunks (in terms of independent changeability).

Work items range in size from very large changes, such as *releases*, to very small changes, such as individual *deltas* (modifications) to a file. A release, also called a *customer delivery*, is a set of *features* and problem fixes. A feature is a group of *modification requests* associated with new software functionality. And, an MR is an individual request for changes. Put another way, each release can be characterized as a base system that a set of MRs modifies and extends. Figure 1 shows a hierarchy of work items with associated attributes.

The source code of large software products is typically organized into subsystems according to major functionality (database, user interface, and so on). Each subsystem contains source code files and documentation. A version control system maintains the source code and documentation versions. Common VCSs are the Concurrent Versioning System,[4] which is commonly used for open source software projects, and commercial systems, such as ClearCase, Continuus Change Management Suite, or Visual SourceSafe. We frequently deal with Source Code Control System and its descendants.[5]

VCSs operate over a set of source code files. An atomic change or delta to the program text comprises the deleted lines and the lines added to make the change. Deltas are usually computed by a file-differencing algorithm (such as Unix diff), invoked by the VCS, which compares an older version of a file with the current version. Included with every delta is information such as the time it was made, the person making it, and a short comment describing it.

In addition to a VCS, most projects employ a change request management system that tracks MRs. Whereas deltas track changed lines of code, MRs are collections of deltas made for a single purpose—for example, to fix a simple defect. Some commonly used problem-tracking systems include ClearDDTS from Rational and the Extended Change Management System (ECMS).[6] Most commercial VCSs also support problem tracking. Usually, such systems associate a list of deltas with each MR.

There are several reasons for MRs, including fixing previous changes that caused a failure during testing or in the field and introducing new features to the existing system. Some MRs restructure the code to make it easier to understand and maintain. The latter activity is more common in heavily modified code, such as in legacy systems.

Based on informal interviews in several software development organizations within Lucent, we obtained the following guidelines for dividing work into MRs:

- The software-development team splits work items that affect several subsystems (the largest building blocks of functionality) into distinct MRs so that each MR affects one subsystem.
- The team further organizes a work item in a subsystem that is too much for one person into several MRs, each suitable for one person.

For practical reasons, organizations avoid strictly enforcing these guidelines so that some MRs cross subsystem boundaries and some have several people working on them.

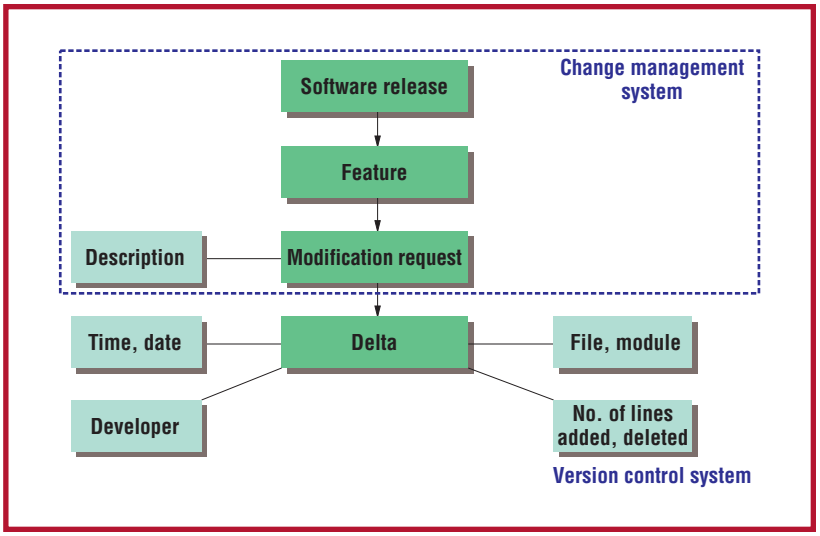


Figure 1. Hierarchy of work items and associated data sources. Dashed lines define data sources, thick lines define changes, and thin lines define work-item properties. The arrows define "contains" relationships among changes; for example, each modification request is a part of a feature.

Work-item-based measures of coordination needs

Changes performed both within and outside of work items require coordination. For

The tight coordination needed within MRs suggests that they're the smallest work items that can be done independently of each other.

a software release, all coordination happens within the release, whereas for an individual delta on a file, coordination is between the file's other deltas.

Changes made as part of an MR require tight internal coordination and are preferably done by a single developer. For example, a change to a function's parameters would require a change in function declaration, function definition, and all the places where the function is called. Conversely, coordination between MRs, although needed, typically does not represent as much coordination as do changes within one MR. The tight coordination needed within MRs suggests that they're the smallest work items that can be done independently of each other. In particular, MRs can be assigned to distinct development sites or distinct organizations. This hypothesis is supported by the evidence that MRs involving developers distributed across geographic locations take much longer to complete.[7]

Based on the guidelines for dividing work into MRs described previously, the work items encompassing several MRs might reflect only a weak coupling among parts of the code that they modify. Consequently, such work items might be divided among several developers.

The tight coupling of work in an MR suggests the following measure of work-item-based coupling between entities in a software project. For two entities A and B, the number of MRs that result in changes to or activity by both A and B define the measure of absolute coupling. For example, if A and B represent two subsystems of the source code, the absolute measure of work-item coupling would be the number of MRs such that each MR changes the code in both subsystems. The coupling for two group of developers would be represented by the number of MRs such that each MR has at least one developer from each group assigned to it. A coupling between the code and a group of developers is defined in a similar fashion. To adjust for A and B's size, dividing the absolute measure by the total number of MRs that relate to A or B can provide measures of relative coupling.

Coordination needed to accomplish MRs is also embodied in other activities and in ways that aren't reflected in the preceding coupling measures. Examples are coordination among MRs in a feature or during system integration and testing.

Empirical evaluation of work-coupling measures

Let's now examine the work-coupling measures of a major telecommunications software project to investigate their relationship to the development interval (also known as *project lead time* or *project development time*). The project involved work in a complex area of telephony, where market requirements and standards are changing rapidly. Such conditions make coordinating the development work extremely difficult and subject to continuous change. Additionally, the product competes in an aggressive market—a situation that brings extreme time pressures to development work.

You can find a detailed study of the project elsewhere.[7] Here, we focus on coupling measures between different project sites (located in Germany, England, and India) and their relationship with development interval. We define *work interval* as the difference between the date of the last delta and the first delta for an MR. Such a measure is a good approximation of the period of time, or interval, that implementing the change requires.

For each MR, we determined whether the individuals associated with it were co-located or resided at more than one site. MRs that have individuals from more than one site are classified as multisite; the rest are classified as single-site. The ratio of multisite MRs to total MRs is a relative measure of MR coupling between sites; the ratio provides an approximation of the coordination needs between the sites.

The work-interval comparison in Table 1 considers MRs done over two years—July 1997 to July 1999—that were nontrivial (required more than one delta). Because multisite MRs involve at least two people, to avoid bias we excluded single-person MRs from the first comparison. Table 1 shows that approximately 18 percent of multiperson MRs are multisite and incur an average penalty of 7.6 days with 95-percent confidence interval of [3,13] days. We can use the MR-coupling measure between sites (216 MRs) together with average interval penalty (7.6 days) and average number of participants in multiperson MRs (2.6) to obtain the total delay of

Table 1

Comparison of Work Interval Measured in Calendar Time

Modification requests	Sites	Average interval (days)	Number of modification requests
Multiperson	Single	8.2	979
	Multiple	15.8	216
All	Single	6.8	1408

11.7 (7.6*216*2.6/365) person years (PY).

These data suggest that multisite MRs carry a significant penalty of increased work interval and that reducing the number of multisite MRs could reduce work interval by eliminating these delays. Work done by James Herbsleb and others indicates that, primarily, communication inefficiencies caused the longer development intervals.[7]

Globalization: A problem of distributing software development

Our main goal is to help project management make better-informed decisions through quantitative evaluation of possible consequences. We start by asking, "What work could be transferred from a primary site with resource shortages to a secondary site that has underutilized development resources?" To answer, we evaluate a particular transfer approach's costs and benefits and use an algorithm to find the best possible transfer. In studying such transfers in Lucent Technologies, we observed the following approaches being considered or used:

- *Transfer by functionality*, in which the ownership of a subsystem or set of subsystems is transferred. This was the most commonly applied approach in the software organizations we studied. Distributing development among different sites by functional area ensures that each site will have its own domain expertise and therefore require only a small- to medium-sized development group that could be trained relatively quickly. The main disadvantage is that adding new functionality might require using experts from several sites, thereby increasing the need to coordinate feature work between sites.
- *Transfer by localization*, in which developers modify the software product locally for a local market. An example of such a modification is translating the documentation and user interface into a local language. An advantage of such an approach is that the local development team is highly aware of its customers' needs and knows the local language and the nature of locality-specific features. A disadvantage is the requirement to maintain experts in all the domains that might require change when adapting the system to the local market. Often, such an adaptation requires expertise in virtually all of the system's domains.
- *Transfer by development stage*, in which developers perform different activities at different locations (for example, developers might perform design and coding at a different site than system testing). The advantages include having development-stage experts at a single site, but the disadvantages include a need to communicate and coordinate between sites to proceed to the next development stage.
- *Transfer by maintenance stage*, in which developers transfer older releases primarily for the maintenance phase when they no longer expect to add new features to the release. This makes more resources available for developing new functionality at the site uninvolved in the maintenance phase. The disadvantages include a potential decrease in quality and increase in problem resolution intervals because the site maintaining the product hasn't participated in the design and implementation of the functionality they maintain. Communication needs between the original site and the maintaining site might increase when difficult maintenance problems require the original site's expertise.

The *globalization* problem—the difficulty of distributing development among several sites—is multifaceted, involving trade-offs in training needs, utilization of available expertise, and risk assessment, as well as a number of social and organizational factors.[8]

Although we focus on the ways to mini-

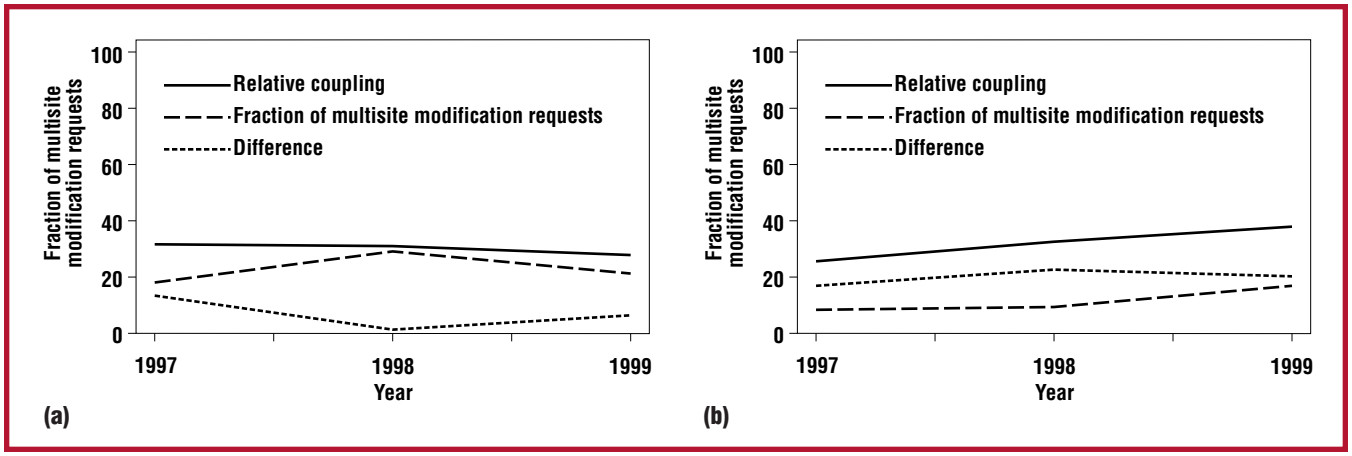


Figure 2. Two globalization candidates. For both candidates, the solid line shows the yearly trend of relative measure of work-item-based coupling between the candidate and the complement, the dashed line shows the trend of the fraction of multisite maintenance requests within a candidate, and the dotted line shows the difference between the two trends: (a) candidate 1 appears to be significantly better for distribution than candidate 2 (b).

mize the need for coordination and communication among sites, in practice it is equally important to use documentation, practices, and tools to enable better communication and coordination. This includes maintaining systems documentation, user manuals, and design and requirements documents; providing good email, telephone, and video-conference facilities, and using presence-awareness tools such as instant messengers and electronic message boards. (A reference of communication and coordination needs for globally distributed software development and a list of promising tools are available elsewhere.)[1,7]

Qualitative factors

Globalization might lead to transfer of work that is in some way undesirable to the primary site. The last three globalization approaches noted in the preceding section reflect different types of undesirable work, such as localization, maintenance (often referred to as *current engineering*), testing, and tools support. We observed several instances of functionality transfer (the first approach), where the areas undesirable to the primary site are transferred. (Of course, they might have been transferred for other reasons as well.)

The decision to transfer work might involve informal risk-management strategies, especially if the transfer is to a secondary site that hasn't worked with the primary site before or had problems working with the primary site in the past. The risk-management strategies consist of identifying work that isn't critical to the overall project in general and to the primary site in particular so that the completion of the project (especially the work in the primary location) would not be catastrophically affected by potential delays or quality problems at the secondary site. Examples of such "noncritical" work include simulation environments, development-tool enhancements, current engineering work, and parts of regression testing. To some extent, the risk management can be done by transferring a functional area, such as a part of operations, administration, and management.

For the work transfer to be successful, the receiving location needs appropriate training. If the work involves knowing the fine points of legacy systems, the primary site must expect to offer significant training. Such a situation is likely to arise if the maintenance or testing stages are transferred. The amount of training might be especially high if the secondary location has high programmer turnover and therefore must continuously retrain personnel. The training needs vary depending on how specialized the work is.

Quantitative evaluation criteria

Although there are a number of dimensions to costs and benefits, we focus on quantifying several aspects of the globalization problem. We propose two globalization scenarios:

- when an organization evaluates the competing globalization factors, and
- when an organization generates globalization solutions.

The most common globalization approach that we have seen is to divide functionality among the locations. We quantify a number of factors for that approach.

Work coupling. Work items spanning multiple locations tend to introduce coordination overheads and associated delays. Consequently, having as few of such work items as possible is desirable. This criterion can be measured by the number of MRs that modify both the candidate and the complementary parts of the software. That number is the measure of absolute coupling between the candidate and the rest of the system. Chunks are the candidates that minimize this measure, because they have the minimal amount of coupling to the rest of the code base.

In addition to predicting future coordination needs, assessing the candidate part of software's current coordination overhead is important. Organizations can make that assessment by counting the MRs which involve participants from multiple locations.

Figure 2 compares two globalization candidates. Each candidate is represented by a list of files that people involved in the globalization decision review. Both candidates start with approximately the same degree of relative coupling, but Candidate 1's relative coupling tends to decrease in time whereas Candidate 2's tends to increase. Additionally, Candidate 1 requires considerably more multisite MRs than candidate 2. Consequently, Candidate 1 appears to be significantly better for distribution than Candidate 2.

Amount of effort. When assigning a part of the code to a remote location, it is important to ensure that the effort needed on that part of the code matches the candidate location's development-resource capacity. It is also important that the candidate embodies some minimal amount of work; transferring a candidate that requires only a trivial amount of effort might not be worthwhile.

The organization can estimate the amount of work that a candidate needs by assessing that candidate's historical effort trends. Assuming that a developer spends roughly equal amounts of effort for each delta, adding the proportions of deltas each developer completed on the candidate during that year can give an approximation of the total effort spent during a year. For example, a developer who completed 100 deltas in a year, 50 of which apply to a particular candidate, would contribute .5 technical head-count years to the candidate. (The scale of effort is thus in terms of PY.) In our experience, resources of between 10 and 20 PY were available in the remote locations, roughly corresponding to a group reporting to a technical manager. The assumption that each delta (done by the same programmer) carries equal amount of effort is only a rough approximation. In fact, in a number of software projects, a delta that fixes a bug requires more effort than a delta that adds new functionality.[9] However, in our problem the approximation of equal effort per delta is reasonable because there is fairly large prediction noise (because the effort spent on a candidate might vary over time). Furthermore, each programmer is likely to have a mixture of different deltas in the candidate, averaging out the distinctions in effort among the different types of deltas. In cases when managers need more precise estimates, models are available[9] that could help find a more precise effort for each delta.

Learning curves. When a chunk of code is transferred to developers who are unfamiliar with the product, the developers might need to substantially adjust their effort. In one of the projects we studied, a typical rule of thumb was that the remote new team would reach full productivity was 12 months. Figure 3 presents an empirical estimate of such a curve. The productivity is measured by the number of deltas a developer completes in a month. We shifted the time for each developer to show their first delta occurring in month one, which let us calculate productivity based on the developer's experience with the transferred code. The figure shows that the time to reach full productivity (flat learning curve) is approximately 15 months. Because developers in this project train for three months before starting work, the total time to reach full productivity is 18 months.

In one of the projects we studied, a typical rule of thumb was that the remote new team would reach full productivity in 12 months.

Algorithm to find the best candidates

We also investigated ways to generate candidates that optimize a desired criterion. Organizations can compare such automatically generated alternatives to existing candidates using qualitative and quantitative evaluations.

Based on the previous analysis, we have the following criteria for evaluating candidates:

- The number of MRs that modify both

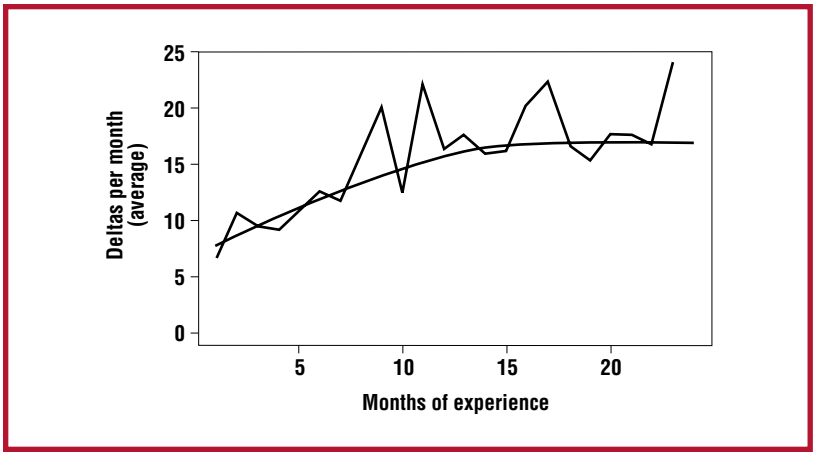


Figure 3. Learning curve. The horizontal axis shows a developer's experience on the project in months and the vertical axis shows the average number of deltas for 50 developers who started working on the project between 1995 and 1998. The jagged curve represents monthly averages, and the smooth curve illustrates the trend by smoothing the monthly data.

the candidate and the rest of the system should be minimized.
- The number of MRs within the candidate that involve participants from several sites should be maximized.
- The effort needed to work on the candidate should approximately match the spare development resources at the proposed remote site.

Because the first two criteria both measure the number of undesirable MRs, we can minimize the difference between them. In other words, let A be the number of multisite MR's at present, and let B be the number of multisite MRs after the organization transfers the candidate to a remote site. The increase in multisite MRs because of such a transfer can be expressed by the difference: $B - A$. The number B can be approximated by the number of MR's that cross the candidate's boundary (the first criterion). The number A represents multisite MRs that are entirely within a candidate, and, presumably, they will become single-site MRs once the organization transfers the candidate to a new location (the second criterion).

The algorithm generates possible candidates and selects the best according to the desired criterion. We use a variation of simulated annealing,[10,11] whereby new candidates are generated iteratively from a current candidate. The algorithm accepts the generated candidate as the current candidate with a probability that depends on whether the evaluation criteria for the generated candidate are better than for the current candidate.

As input to the algorithm, we provide a set of files or modules; each file is associated with an effort in PY for the prior year (calculated as described previously). Another input consists of a set of MRs, in which each MR is associated with the list of files it modifies and with an indicator of whether it is a multisite MR. Finally, we provide a range of effort in PY for the candidate. Initially, the algorithm generates a candidate by randomly selecting modules until it gets within the bounds of the specified effort.

The new candidate is generated iteratively, where the iteration involves randomly choosing one of three steps:

- Add a module to the candidate set by randomly selecting modules from the system's complement until one emerges that does not violate the effort boundary conditions.
- Delete a module from the candidate set by randomly selecting modules to delete from the candidate until one emerges that does not violate the effort boundary conditions.
- Exchange modules by randomly selecting one module from the candidate and one from the complement until the exchange does not violate the effort boundary conditions.

Once the new candidate is generated, the algorithm evaluates the criterion of interest (coupling to the rest of the system) and compares it to the current candidate's value. If the criterion is improved, the algorithm accepts the new candidate as the current candidate; if not, the algorithm accepts the new candidate as the current candidate with a probability $p < 1$. This probability p is related to the annealing temperature, which can be decreased with the number of iterations to speed the convergence. Because the computation speed wasn't a challenge for us, we chose to keep the p always above 1/3 to make sure that the algorithm explores the entire solution space without getting stuck in a local minimum. If the current criterion improves upon the criterion value obtained in any step before the iteration, the current candidate and the criterion are recorded as the best solution. This ends the iteration.

In practice, we use a slight modification of the algorithm, in which we record optimal candidates for different effort bounds during a single run of the algorithm. Of the two candidates in Figure 2, the first is optimal among candidates consuming approximately 10 PY per year, and the second is optimal among candidates consuming approximately 20 PY per year.

Software-development teams could use similar techniques to compare the chunks and the modularity of the code—that is, to check if the work items match the source code directory structure, which typically represents the software's modularity. In large software systems, the alignment between work items and organizational and software structures answers several important practical questions:

- What is the current work structure, and does it match the initial architecture?
- Do the current work and software structures match the organizational structure?
- Does the current work structure match the organization's geographic distribution?
- How do we define a piece of software so that it is and remains an independent chunk that developers could develop or change independently: Is it a file, directory, or some other entity?

Our approach applies to any project in which change data have been accumulated. Even in so-called greenfield projects, the development proceeds by incremental change so that once the project has produced a substantial amount of code, the algorithm could be applied to the change data. The same technique applies to other areas, including distributing work to contractors in the same country or assessing an existing distribution.

Because of our strong emphasis on independent changeability, we think about what we have done as exposing the empirical information hiding a software system's structure. As a system evolves, decisions that are embodied in the code's structure become intertwined such that they are dependent on each other; a change to one usually means a change to the others. Evolution of the system drives the formation of chunks. The challenge for the software architect is to construct a modular design where the modules and the chunks closely correspond to each other throughout the system's lifetime.

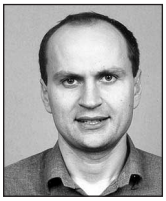
Acknowledgments

We thank the many software developers at Lucent who have assiduously used the configuration control systems that provided us with the data we needed to perform this and other studies. We also thank development managers Mary Zajac, Iris Dowden, and Daniel Owens for sharing their globalization experiences and opinions.

References

1. E. Carmel, *Global Software Teams*, Prentice-Hall, Upper Saddle River, N.J., 1999.
2. M.E. Conway, "How Do Committees Invent?," *Datamation*, vol. 14, no. 4, Apr. 1968, pp. 28–31.
3. D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM*, vol. 15, no. 12, 1972, pp. 1053–1058.
4. *CVS—Concurrent Versions System*, www.cvshome.org/docs/manual/index.html (current 9 Feb. 2001).
5. M.J. Rochkind, "The Source Code Control System," *IEEE Trans. Software Eng.*, vol. 1, no. 4, 1975, pp. 364–370.
6. A.K. Midha, "Software Configuration Management for the 21st Century," *Bell Labs Technical J.*, vol. 2, no. 1, Winter 1997, pp. 154–165.
7. J.D. Herbsleb et al., "Distance, Dependencies, and Delay in a Global Collaboration," *Proc. ACM 2000 Conf. Computer-Supported Cooperative Work*, ACM Press, New York, 2000, pp. 319–328.
8. R.E. Grinter, J.D. Herbsleb, and D.E. Perry. "The Geography of Coordination: Dealing with Distance in R&D Work." *Proc. GROUP '99*, ACM Press, New York, 1999, pp. 306–315.
9. T.L. Graves and A. Mockus, "Inferring Change Effort from Configuration Management Data," *Proc. Metrics 98: Fifth Int'l Symp. Software Metrics*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 267–273.
10. N. Metropolis et al., "Equation of State Calculations by Fast-Computing Machines," *J. Chemical Physics*, vol. 21, 1953, pp. 1087–1092.
11. S. Kirkpatrick, C.D. Gellat Jr., and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, May 1983, pp. 671–680.

About the Authors



Audris Mockus is a member of the technical staff in the Software Production Research department of Bell Labs, where he designs data-mining methods to summarize and augment the system-evolution data; Web-based interactive visualization techniques to inspect, present, and control the systems; and statistical models and optimization techniques to understand the systems. He is investigating properties of software changes of large software systems. He received a BS and an MS in applied mathematics from the Moscow Institute of Physics and Technology, and an MS and a PhD in statistics from Carnegie Mellon University. He is a member of the IEEE and American Statistical Association. Contact him at Bell Labs, 263 Shuman Blvd., Rm. 2F-319, Naperville, IL 60566; audris@research.bell-labs.com; www.bell-labs.com/~audris.

David M. Weiss is the director of software technology research at Avaya Laboratories, where he performs and guides research into ways of improving the effectiveness of software development. Formerly, he was the director of software production research at Bell Labs. He has also served as CTO of PaceLine Technologies and as the director of reuse and measurement at the Software Productivity Consortium. At the Congressional Office of Technology Assessment, he was coauthor of an assessment of the Strategic Defense Initiative, and he was a visiting scholar at the Wang Institute. He originated the GQM approach to software measurement, was a member of the A-7 project at the Naval Research Laboratory, and devised the FAST process for product-line engineering. He has also worked as a programmer and a mathematician. He received a PhD in computer science from the University of Maryland. He is a member of the IEEE and ACM. Contact him at Avaya Communication, 2C-555, 600-700 Mountain Ave., PO Box 636, Murray Hill, NJ 07974; weiss@avaya.com, www.avayalabs.com/~weiss.