In-process metrics for software testing

by S. H. Kan
J. Parrish
D. Manlove

In-process tracking and measurements play a critical role in software development, particularly for software testing. Although there are many discussions and publications on this subject and numerous proposed metrics, few in-process metrics are presented with sufficient experiences of industry implementation to demonstrate their usefulness. This paper describes several in-process metrics whose usefulness has been proven with ample implementation experiences at the IBM Rochester AS/400® software development laboratory. For each metric, we discuss its purpose, data, interpretation, and use and present a graphic example with real-life data. We contend that most of these metrics, with appropriate tailoring as needed, are applicable to most software projects and should be an integral part of software testing.

Measurement plays a critical role in effective software development. It provides the scientific basis for software engineering to become a true engineering discipline. As the discipline has been progressing toward maturity, the importance of measurement has been gaining acceptance and recognition. For example, in the highly regarded software development process assessment and improvement framework known as the Capability Maturity Model, developed by the Software Engineering Institute at Carnegie Mellon University, process measurement and analysis and utilizing quantitative methods for quality management are the two key process activities at the Level 4 maturity.[1,2]

In applying measurements to software engineering, several types of metrics are available, for example, process and project metrics versus product metrics, or metrics pertaining to the final product versus metrics used during the development of the product. From the standpoint of project management in software development, it is the latter type of metrics that is the most useful—the in-process metrics. Effective use of good in-process metrics can significantly enhance the success of the project, i.e., on-time delivery with desirable quality.

Although there are numerous discussions and publications in the software industry on measurements and metrics, few in-process metrics are described with sufficient experiences of industry implementation to demonstrate their usefulness. In this paper, we intend to describe several in-process metrics pertaining to the test phases in the software development cycle for release and quality management. These metrics have gone through ample implementation experiences in the IBM Rochester AS/400* (Application System/400*) software development laboratory for a number of years, and some of them likely are used in other IBM development organizations as well. For those readers who may not be familiar with the AS/400, it is a midmarket server for e-business. To help meet the demands of enterprise e-commerce applications, the AS/400 features native support for key Web-enabling technologies. The AS/400 system software includes microcode supporting the hardware, the Operating System/400* (OS/400*), and many licensed program products supporting the latest tech-

0018-8670/01/$5.00 © 2001 IBM

nologies. The size of the AS/400 system software is currently about 45 million lines of code. For each new release, the development effort involves about two to three million lines of new and changed code.

It should be noted that the objective of this paper is not to research and propose new software metrics, although it may not be that all the metrics discussed are familiar to everyone. Rather, its purpose is to discuss the usage of implementation-proven metrics and address practical issues in the management of software testing. We confine our discussion to metrics that are relevant to software testing after the code is integrated into the system library. We do not include metrics pertaining to the front end of the development process such as design review, code inspection, or code integration and driver builds. For each metric, we discuss its purpose, data, interpretation and use, and where applicable, pros and cons. We also provide a graphic presentation where possible, based on real-life data. In a later section, we discuss in-process quality management vis-a-vis these metrics and a metrics framework that we call the *effort/outcome paradigm*. Before the conclusion of the paper, we also discuss the pertinent question: How do you know your product is good enough to ship?

Since the examples in this paper are based on experiences with the AS/400, it would be useful to outline the software development and test process for the AS/400 as the overall context. The software development process for the AS/400 is a front-end focused model with emphases on key intermediate deliverables such as architecture, design and design verification, code integration quality, and driver builds. For example, the completion of a high-level design review is always a key event in the system schedule and managed as a key intermediate deliverable. At the same time, testing (development and independent testing) and customer validation are the key phases with an equally strong focus. As Figure 1 shows, the common industry model of testing includes functional test, system test, and customer beta trials before the product is shipped. Integration and solution test can occur before or after a product is first shipped and is often conducted by customers because a customer's integrated solution may consist of products from different vendors. For AS/400, the first formal test after unit test and code integration into the system library consists of component test (CT) and component regression test (CRT), which is equivalent to functional test. Along the way, a stress test is also conducted in a large network environment with performance workload running in the background to stress the system. When significant progress is made in component test, the product level test (PLT), which focuses on the products and subsystems of the AS/400 system (e.g., Transmission Control Protocol/Internet Protocol, database, client access, clustering), starts. Network test is a specific product level test focusing on communications and related error recovery processes. The above tests are conducted by the development teams (CT, CRT) or by joint effort between the development teams and the independent test team (PLT, stress test). The tests done by the independent test group include the install test and the software system integration test (called Software RAISE—for reliability, availability, install, service, environment) which is preceded by its acceptance test—the system test acceptance test (STAT). Because AS/400 is a highly integrated system, these different tests each play an important role in achieving a quality deliverable for each release of the software system. As Figure 1 shows, several early customer programs also start in the back end of our development process, and some of them normally run until 30 days after general availability (GA) of the product.

In-process metrics for software testing

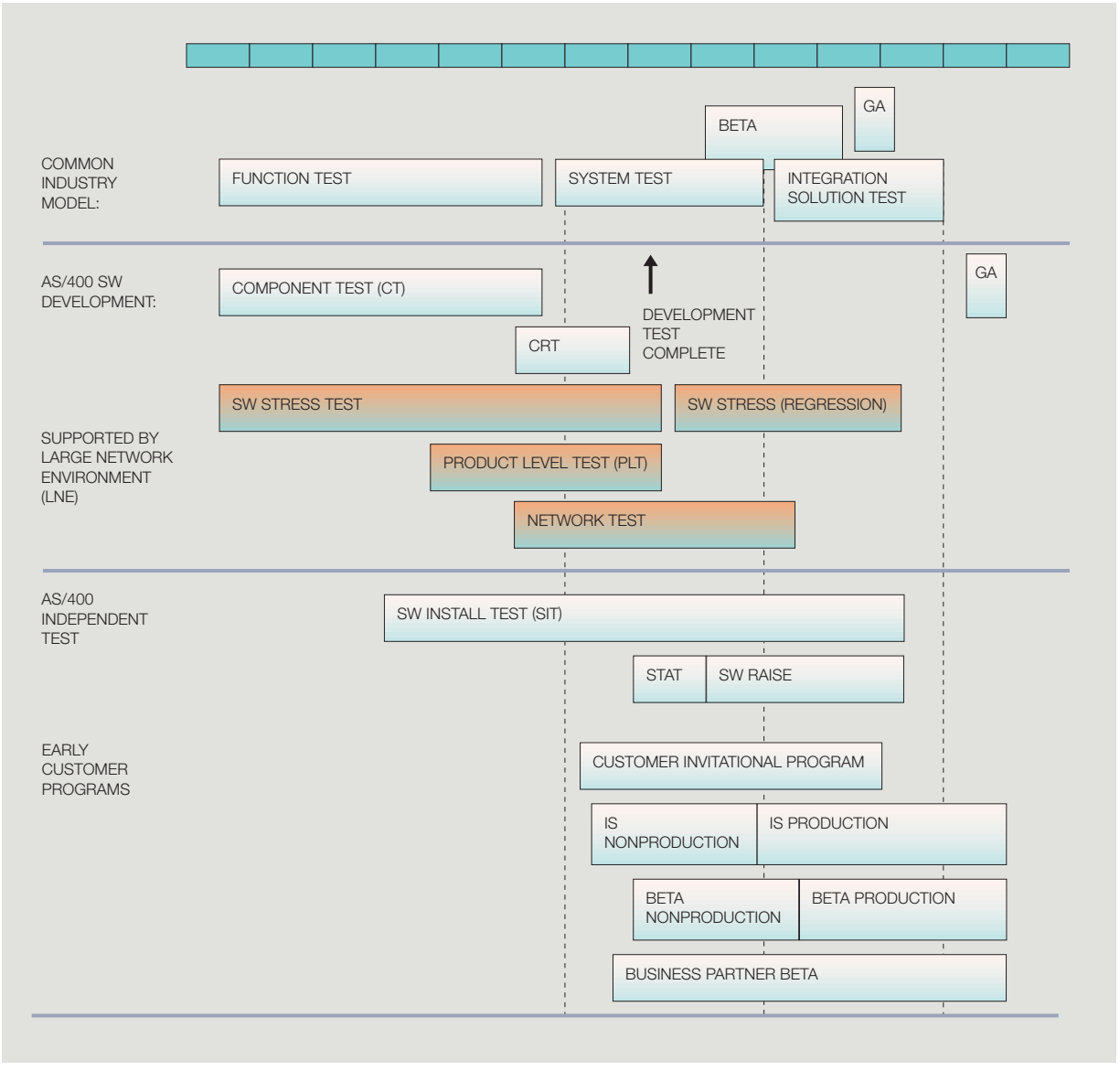
In this section, we discuss what in-process metrics are used in software testing.

Test progress S curve (plan, attempted, actual). Test progress tracking is perhaps the most important and basic tracking for managing software testing. The metric we recommend is a test progress S curve over time with the x-axis representing the time unit (preferably in weeks) and the y-axis representing the number of test cases or test points. By S curve we mean that the data are cumulative over time and resemble an "s" shape as a result of the period of intense test activity, causing a steep planned test ramp-up. For the metric to be useful, it should contain the following information on the same graph:

- Planned progress over time in terms of number of test cases or number of test points to be completed successfully by week
- Number of test cases attempted by week
- Number of test cases completed successfully by week

The purpose of this metric is obvious—to track actual testing progress against plan and therefore to be able to be proactive upon early indications that testing activity is falling behind. It is well-known that

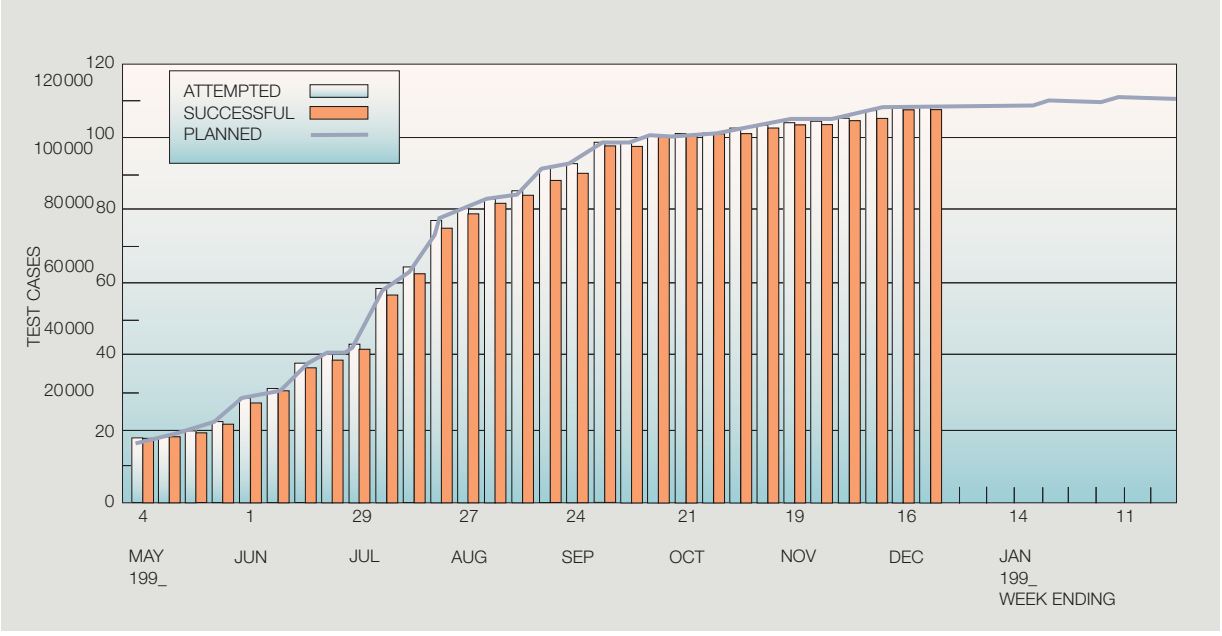
Figure 1 AS/400 software testing cycle



when the schedule is under pressure in software development, it is normally testing (especially development testing, i.e., unit test and component test or functional verification test) that is impacted (cut or reduced). Schedule slippage occurs day by day and week by week. With a formal metric in place, it is much more difficult for the team to ignore the problem, and they will be more likely to take actions. From the project planning perspective, the request for an S curve forces better planning (see further discussion in the following paragraphs).

The example shown in Figure 2 shows the component test (functional verification test) metric at the end of the test for a major release of the AS/400 software system.

Figure 2 Testing progress S curve example

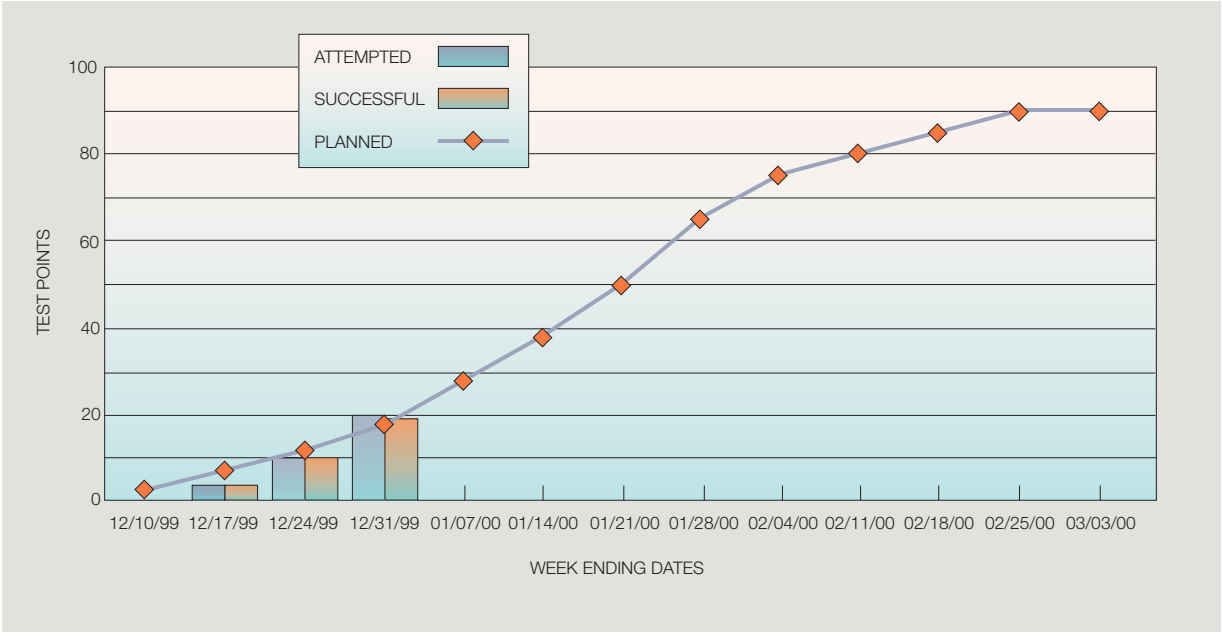


As can be seen from the figure, the testing progress plan is expressed in terms of a line curve, which is put in place in advance. The lightly shaded bars are the cumulative number of test cases attempted, and the red bars represent the number of successful test cases. With the plan curve in place, each week when the test is in progress, two more bars (one for attempted and one for successful completion) are added to the graph. This example shows that during the rapid test ramp-up period (the steep slope of the curve), for some weeks the test cases attempted were slightly ahead of plan (which is possible), and the successes were slightly behind plan.

Because some test cases may be more important than others, it is not a rare practice in software testing to assign test scores to the test cases. Using test scores is a normalized approach that provides more accurate tracking of test progress. The assignment of scores or points is normally based on experience, and for AS/400, teams usually use a 10-point scale (10 for the most important test cases and 1 for the least). As mentioned before, the unit of tracking for this S curve metric can be any unit that is appropriate. To make the transition from basic test-case S-curve tracking to test-point tracking, the teams simply enter the test points into the project management tool (for the overall plan initially and for actuals every week). The example in Figure 3 shows test point tracking for a product level test, which was underway for a particular function of the AS/400 system. It should be noted that there is always an element of subjectivity in the assignment of weights. The weights and the resulting test scores should be determined in the testing planning stage and remain unchanged during the testing process. Otherwise, the purpose of this metric will be lost in the reality of schedule pressures. In software engineering, weighting and test score assignment remains an interesting area where more research is needed. Possible guidelines from such research will surely benefit the planning and management of software testing.

For tracking purposes, testing progress can also be weighted by some measurement of coverage. For example, test cases can be weighted by the lines of code tested. Coverage weighting and test score assignment consistency become increasingly important in proportion to the number of development groups involved in a project. Lack of attention to tracking consistency across functional areas can result in a misleading view of true system progress.

Figure 3 Testing progress S curve—test points tracking



When a plan curve is in place, an in-process target can be set up by the team to reduce the risk of schedule slippage. For instance, a disparity target of 15 percent or 20 percent between attempted (or successful) and planned test cases can be used to trigger additional actions versus a business-as-usual approach. Although the testing progress S curves, as shown in Figures 2 and 3, give a quick visual status of the progress against the total plan and plan-to-date (the eye can quickly determine if testing is ahead or behind on planned attempts and successes), it may be difficult to discern the exact amount of slippage. This is particularly true for large testing efforts, where the number of test cases is in the hundreds of thousands, as in our first example. For that reason, it is useful to also display testing status in tabular form, as in Table 1. The table also shows underlying data broken out by department and product or component, which helps to identify problem areas. In some cases, the overall test curve may appear to be on schedule, but because some areas are ahead of schedule, they may mask areas that are behind when progress is only viewed at the system level. Of course, testing progress S curves are also used for functional areas and for specific products.

When an initial plan curve is put in place, the curve should be subject to brainstorming and being challenged. For example, if the curve shows a very steep ramp-up in a short period of time, the project manager may challenge the team with respect to how doable the plan curve is or what the team's specific actions to execute the plan are. As a result, better planning will be achieved. It should be cautioned that before the team settles on a plan curve and uses it as a criterion to track progress, a critical evaluation of what the plan curve represents should be made. For example, is the total test suite considered effective? Does the plan curve represent high test coverage? What are the rationales behind the sequences of test cases in the plan? This type of evaluation is important because once the plan curve is in place, the visibility of this metric tends to draw the whole team's attention to the disparity between attempts, successes, and the plan.

Once the plan line is set, any changes to the plan should be reviewed. Plan slips should be evaluated against the project schedule. In general, the baseline plan curve should be maintained as a reference. Ongoing changes to the planned testing schedule can mask schedule slips by indicating that attempts are

Figure 4 Testing plan curve—release-to-release comparison

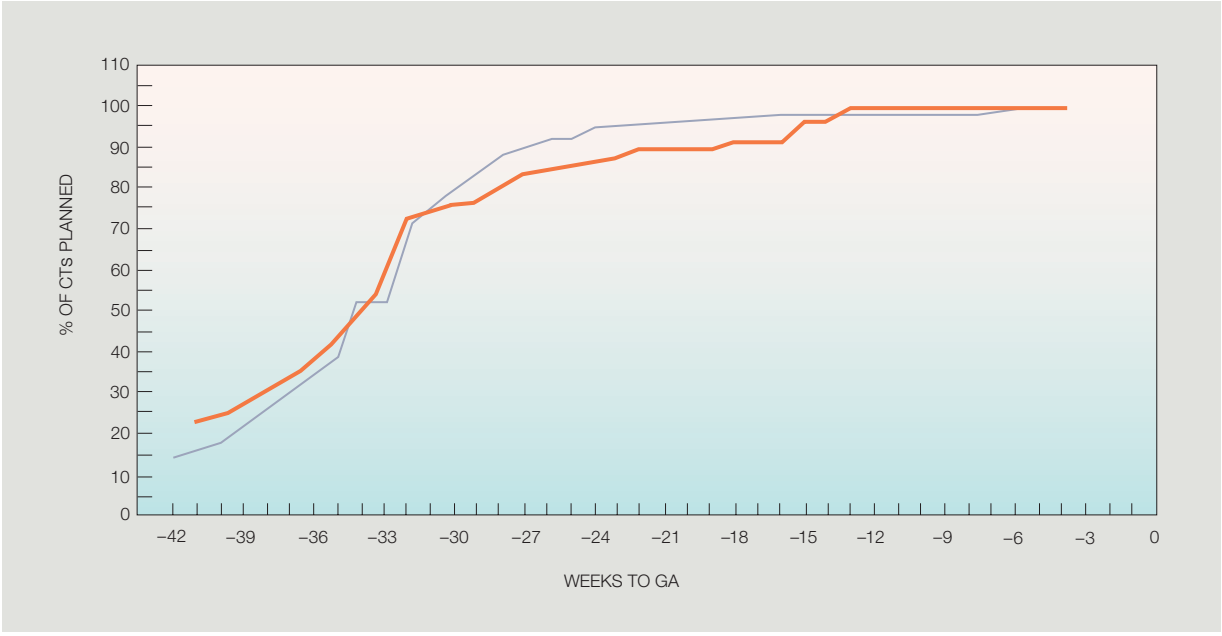


Table 1 Test progress tracking—plan, attempted, successful

	Plan to Date (# of Test Cases)	Percent Attempted of Plan	Percent Successful of Plan	Plan Not Attempted (# of Test Cases)	Percent Attempted of Total	Percent Successful of Total
System	60577	90.19	87.72	5940	68.27	66.1
Dept. A	1043	66.83	28.19	346	38.83	15.6
Dept. B	708	87.29	84.46	90	33.68	32.59
Dept. C	33521	87.72	85.59	4118	70.6	68.88
Dept. D	11275	96.25	95.25	423	80.32	78.53
Dept. E	1780	98.03	94.49	35	52.48	50.04
Dept. F	4902	100	99.41	0	96.95	95.93
Product A	13000	70.45	65.1	3841	53.88	49.7
Product B	3976	89.51	89.19	417	66.82	66.5
Product C	1175	66.98	65.62	388	32.12	31.4
Product D	277	0	0	277	0	0
Product E	232	6.47	6.47	217	3.78	3.7

on track, whereas the plan is actually moving forward.

In addition to what is described above, this metric can be used for release-to-release or project-to-project comparisons, as the example in Figure 4 shows.

For release-to-release comparisons, it is important to use weeks before product general availability (GA) as the time unit for the x-axis. By referencing the GA dates, the comparison provides a true in-process status of the current release. In Figure 4, we can see that the release represented by the red, thicker curve is more back-end loaded than the release represented by the blue, thinner curve. In this context, the metric is both a quality and a schedule statement for the release, as late testing will affect late cycle defect arrivals and hence the quality of the final product. With

this type of comparison, the project team can plan ahead (even before the actual start of testing) to mitigate the risks.

To implement this metric, the testing execution plan needs to be laid out in terms of the weekly target, and actual data need to be tracked on a weekly basis. For small to medium projects, such planning and tracking activities can use common tools such as Lotus 1-2-3** or other project management tools. For large, complex projects, a stronger tools support facility normally associated with the development environment may be needed. For AS/400, the data are tracked via the DCR (Design Change Request—a tool for project management and technical information for all development items) tool, which is a part of the IDSS/DEV2000 (Integrated Development Support System) environment. The DCR is an on-line tool to track important characteristics and the status of each development item. For each line item in the release plan, one or more DCRs are created. Included in the DCRs are descriptions of the function to be developed, components that are involved, major design considerations, interface considerations (e.g., application programming interface and system interface), performance considerations, Double Byte Character Set (DBCS) requirements, component work summary sections, lines of code estimates, machine readable information (MRI), design review dates, code integration schedule, testing plans (number of test cases and dates), and actual progress made (by date). For the system design control group reviews, DCRs are the key documents with the information needed to ensure design consistency across the system. For high-level design reviews, DCRs are often included in the review package together with the design document itself. The DCR process also functions as the change-control process at the front end of the development process. Specifically, for each integration of code into the system library by developers, a reason code (a DCR number) is required. For test progress tracking, development or test teams enter data into DCRs, databases are updated daily, and a set of reporting and analysis programs (VMAS, or Virtual Machine Application System, SAS**, Freelance*) are used to produce the reports and graphs.

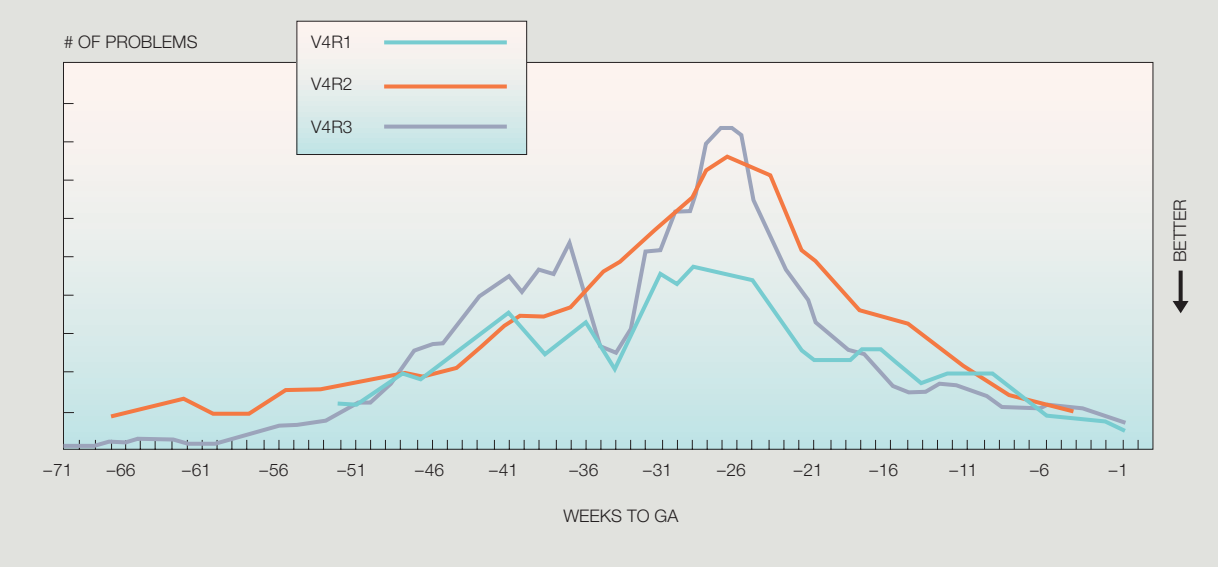
PTR (test defects) arrivals over time. Defect tracking during the testing phase is highly recommended as a standard practice for any software testing. At IBM Rochester, defect tracking is done via the Problem Tracking Report (PTR) tool. With regard to metrics, what is recommended here is tracking the test defect arrival pattern over time, in addition to tracking by test phase. Overall defect density during test, or for a particular test, is a summary indicator, but not really an in-process indicator. The pattern of defect arrivals over time gives more information. Even with the same overall defect rate during testing, different patterns of defect arrivals imply different quality levels in the field. We recommend the following for this metric:

- Always include data for a comparable release or project in the tracking chart if possible. If there is no baseline data or model data, at the minimum, some expected level of PTR arrivals at key points of the project schedule should be set when tracking starts (e.g., system test entry, cut-off date for code integration for GA driver).
- The unit for the x-axis is weeks before GA.
- The unit for the y-axis is the number of PTR arrivals for the week, or related measures.

Figure 5 is an example of this metric for several AS/400 releases. For this example, release-to-release comparison at the system level is the main goal. The metric is also often used for defect arrivals for specific tests, i.e., functional test, product level test, or system integration test.

The figure has been simplified for presentation. The actual graph has much more information, including vertical lines depicting the key dates of the development cycle and system schedules such as last new function integration, development test completion, start of system test, etc. There are also variations of the metric: total PTR arrivals, high-severity PTRs, PTRs normalized to the size of the release (new and changed code plus a partial weight for ported code), raw PTR arrivals versus valid PTRs, and number of PTRs closed each week. The main chart, and the most useful one, is the total number of PTR arrivals. However, we also include a high-severity PTR chart and a normalized view as mainstays of our tracking. The normalized PTR chart can help to eliminate some of the visual guesswork of comparing current progress to historical data. In conjunction with the high-severity PTR chart, a chart that displays the relative percentage of high-severity PTRs per week can be useful. Our experience indicated that the percentage of high-severity problems normally increases as the release moves toward GA, while the total number decreases to a very low level. Unusual swings in the percentage of high-severity problems could signal serious problems and should be investigated.

Figure 5 Testing defect arrivals metric



When do the PTR arrivals peak relative to time to GA and previous releases? How high do they peak? Do they decline to a low and stable level before GA? Questions such as these are crucial to the PTR arrivals metric, which has significant quality implications for the product in the field. The ideal pattern of PTR arrivals is one with higher arrivals earlier, an earlier peak (relative to previous releases or the baseline), and a decline to a lower level earlier before GA. The latter part of the curve is especially important since it is strongly correlated with quality in the field, as long as the decline and the low PTR levels before GA are not attributable to an artificial reduction or even to the halt of testing activities. High PTR activity before GA is more often than not a sign of quality problems. Myers[3] has a famous counterintuitive principle in software testing, which states that the more defects found during testing, the more that remain to be found later. The reason underlying the principle is that finding more defects is an indication of high error injection during the development process, if testing effectiveness has not improved drastically. Therefore, this metric can be interpreted in the following ways:

- If the defect rate is the same or lower than that of the previous release (or a model), then ask: Is the testing for the current release less effective?
  - If the answer is no, the quality perspective is positive.
  - If the answer is yes, extra testing is needed. And beyond immediate actions for the current project, process improvement in development and testing should be sought.
- If the defect rate is substantially higher than that of the previous release (or a model), then ask: Did we plan for and actually improve testing effectiveness significantly?
  - If the answer is no, the quality is negative. Ironically, at this stage of the development cycle, when the "designing-in quality" phases are past (i.e., at test phase and under the PTR change control process), any remedial actions (additional inspections, re-reviews, early customer programs, and most likely more testing) will yield higher PTR rates.
  - If the answer is yes, then the quality perspective is the same or positive.

The above approach to interpretation of the PTR arrival metric is related to the framework of our effort/outcome paradigm for in-process metrics quality management. We discuss the framework later in this paper.

The data underlying the PTR arrival curve can be further displayed and analyzed in numerous ways to gain

additional understanding of the testing progress and product quality. Some of the views that we have found useful include the pattern of build and integration of PTR arrivals and the pattern of activity source over time. The later charts can be used in conjunction with testing progress to help assess quality as indicated by the effort/outcome paradigm.

Most software development organizations have test defect tracking mechanisms in place. For organizations with more mature development processes, the problem tracking system also serves as a tool for change control, i.e., code changes are tracked and controlled by PTRs during the formal test phases. Such change control is very important in achieving stability for large and complex software projects. For AS/400, data for the above PTR metric are from the PTR process that is a part of the IDSS/DEV2000 development environment. The PTR process is also the change control process after code integration into the system library. Integration of any code changes to the system library to fix defects from testing requires a PTR number.

PTR (test defects) backlog over time. We define the number of PTRs remaining at any given time as the "PTR backlog." Simply put, the PTR backlog is the accumulated difference between PTR arrivals and PTRs that were closed. PTR backlog tracking and management are important from the perspective of both testing progress and customer rediscoveries. A large number of outstanding defects during the development cycle will likely impede testing progress, and when the product is about to go to GA, will directly affect field quality performance of the product. For software organizations that have separate teams conducting development testing and fixing defects, defects in the backlog should be kept at the lowest possible level at all times. For those software organizations in which the same teams are responsible for design, code, development testing, and fixing defects, however, there are appropriate timing windows in the development cycle for which the highest priority on what to focus may vary.

Although the PTR backlog should be managed at a reasonable level at all times, it should not be the highest priority during a period when making headway in functional testing is the most important development activity. During the prime time for development testing, the focus should be on test effectiveness and test execution, and defect discovery should be encouraged to the maximum possible extent. Focusing too early on overall PTR reduction may run into a conflict with these objectives and may lead to more latent defects not being discovered or being discovered later at subsequent testing phases (e.g., during the independent testing time frame), or PTRs not opened to record those defects that are discovered. The focus should be on the fix turnaround of the critical defects that impede testing progress instead of the entire backlog. Of course, when testing is approaching completion, there should be a strong focus on drastic PTR backlog reductions.

For software development projects that build on existing systems, a large backlog of "aged" problems can develop over time. These aged PTRs often represent fixes or enhancements that developers believe would legitimately improve the product, but which are passed over during development because of resource or design constraints. They may also represent problems that have already been fixed or are obsolete because of other changes. Without a concerted effort, this aged backlog can build over time. This area of the PTR backlog is one which can and should be focused on earlier in the release cycle, even prior to the start of development testing.

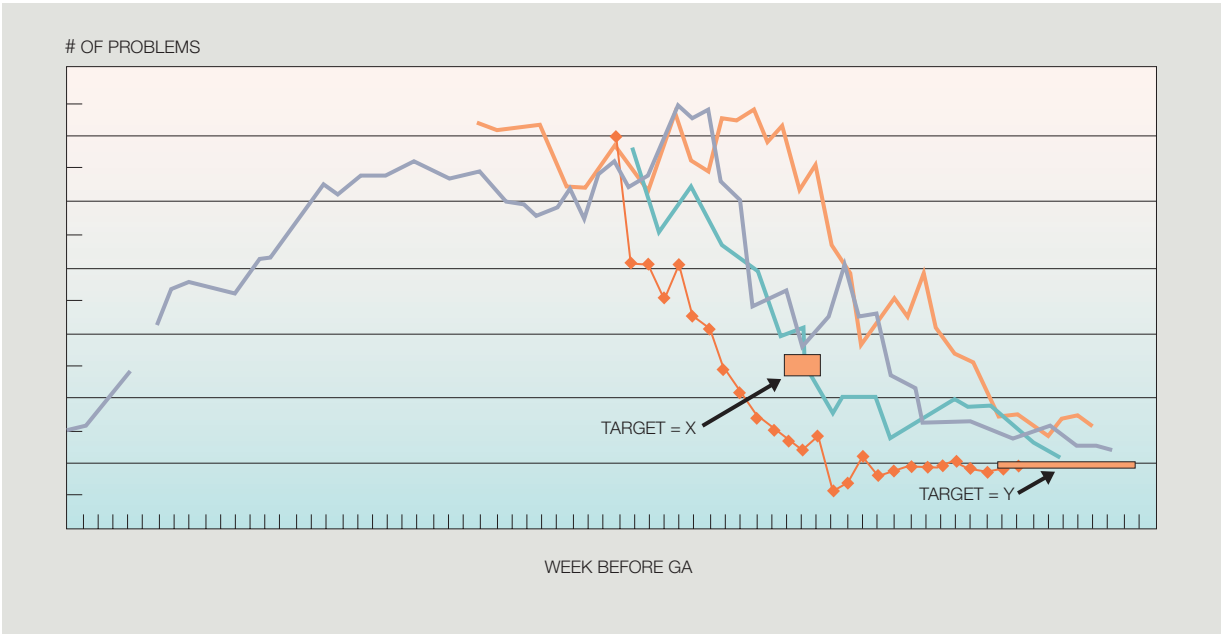
Figure 6 shows an example of the PTR backlog metric for AS/400. Again, release-to-release comparisons and actuals versus targets are the main objectives. The specific targets in the graph are associated with key dates in the development schedule, for example, cut-off dates for fix integration control and for the GA driver.

It should be noted that for this metric, solely focusing on the numbers is not sufficient. In addition to the overall reduction, what types and which specific defects should be fixed first play a very important role in terms of achieving system stability early. In this regard, the expertise and ownership of the development and test teams are crucial.

Unlike PTR arrivals that should not be controlled artificially, the PTR backlog is completely under the control of the development organization. For the three metrics we have discussed so far, the overall project management approach should be as follows:

- When a test plan is in place and its effectiveness evaluated and accepted, manage test progress to achieve early and fast ramp-up.
- Monitor PTR arrivals and analyze the problems (e.g., defect cause analysis and Pareto analysis of problem areas of the product) to gain a better understanding of the test and the quality of the prod-

Figure 6 Testing defect backlog tracking



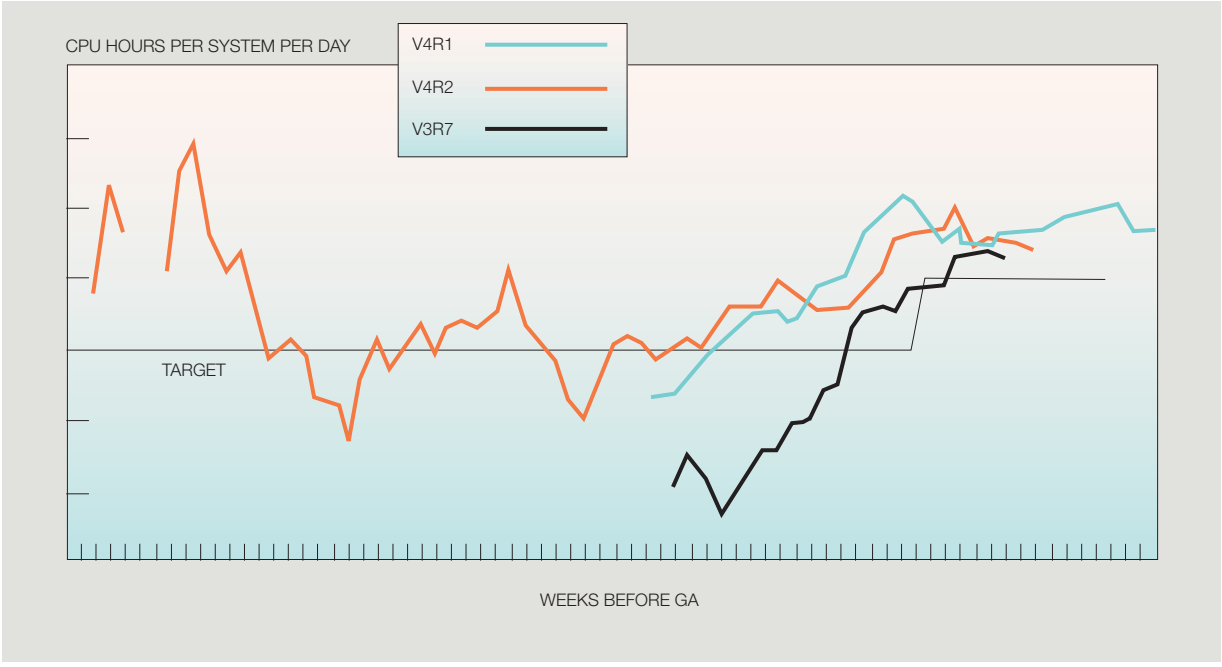
uct, but *do not* artificially manage or control PTR arrivals, which is a function of test effectiveness, test progress, and the intrinsic quality of the code (the amount of defects latent in the code). (Pareto analysis is based on the 80–20 principle that states that normally 20 percent of the causes accounts for 80 percent of the problems. Its purpose is to identify the few vital causes or areas that can be targeted for improvement. For examples of Pareto analysis of software problems, see Chapter 5 in Kan.[4])

- Strongly manage PTR backlog reduction, especially with regard to the types of problems being fixed early and achieving predetermined targets that are normally associated with the fix integration dates of the most important drivers.

The three metrics discussed so far are obviously interrelated, and therefore special insights can be gained by viewing them together. We have already discussed the use of the effort/outcome paradigm for analyzing PTR arrivals and testing progress. Other examples include analyzing testing progress and the PTR backlog together to determine areas of concern and examining PTR arrivals versus backlog reduction to project future progress. Components or development groups that are significantly behind in their testing and have a large PTR backlog to overcome warrant special analysis and corrective actions. A large backlog can mire progress and, when testing progress is already lagging, prevent rudimentary recovery actions from being effective. Combined analysis of testing progress and the backlog is discussed in more detail in a later section. Progress against the backlog, when viewed in conjunction with the ongoing PTR arrival rate, gives an indication of the total capacity for fixing problems in a given period of time. This maximum capacity can then be used to help project when target backlog levels will be achieved.

Product/release size over time. Lines of code or another indicator of product size or function (e.g., function points) can also be tracked as a gauge of the "effort" side of the development equation. During development there is a tendency toward product growth as requirements and designs are fleshed out. Functions may also continue to be added because of late requirements or a developer's own desire to make improvements. A project size indicator, tracked over time, can serve as an explanatory factor for testing progress, PTR arrivals, and PTR backlog. It can also relate the measurement of total defect volume to per unit improvement or deterioration.

Figure 7 CPU utilization metric



CPU utilization during test. For computer systems or software products for which a high level of stability is required in order to meet customers' needs (for example, systems or products for mission-critical applications), it is important that the product performs well under stress. In testing software during the development process, the level of CPU utilization is an indicator of the extent that the system is being stressed.

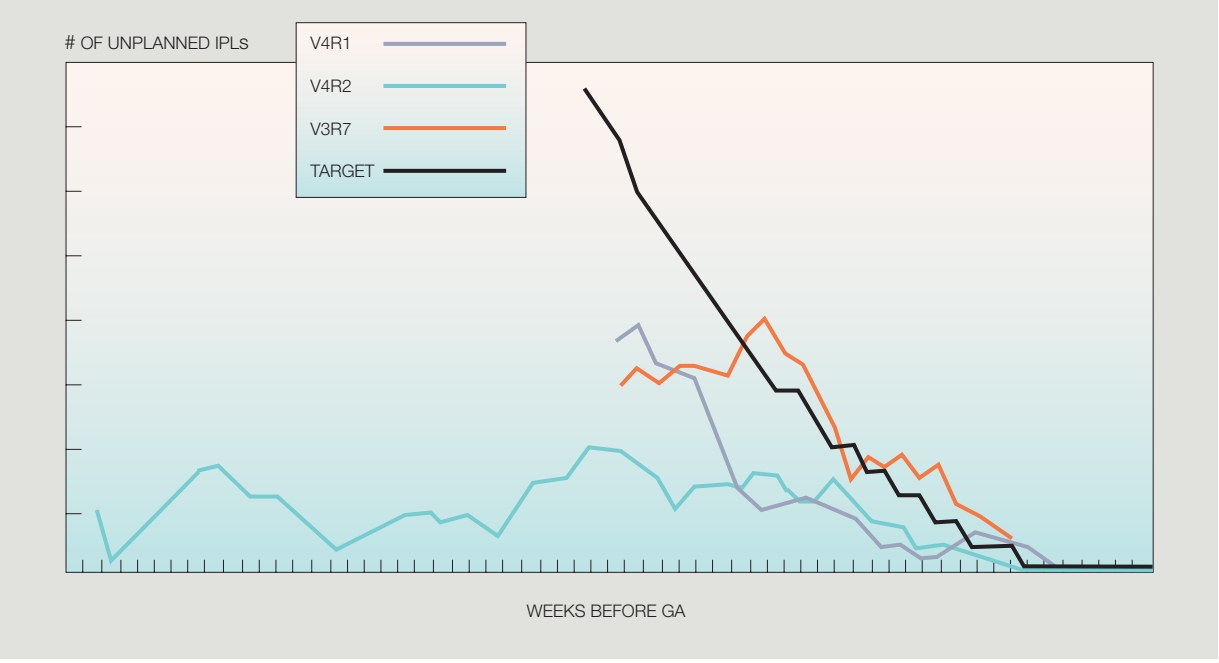
To ensure that our software tests are effective, the AS/400 development laboratory sets CPU utilization targets for the software stress test, the system test acceptance test (STAT), and the system integration test (the RAISE test). The stress test starts at the middle of component test and may run into the RAISE time frame with the purpose of stressing the system in order to uncover latent defects that cause system crashes and hangs that are not easily discovered in normal testing environments. It is conducted with a network of systems. RAISE test is the final system integration test, with a customer-like environment. It is conducted in an environment that simulates an enterprise environment with a network of systems including CPU-intensive applications and interactive computing. During test execution, the environment is run as though it were a 24-hour-a-day, seven-day-a-week operation. STAT is used as a preparatory test prior to the formal RAISE test entry.

The example in Figure 7 demonstrates the tracking of CPU utilization over time for the software stress test. We have a two-phase target as represented by the step line in the chart. The original target was set at 16 CPU hours per system per day on the average, with the following rationale:

- The stress test runs 20 hours per day, with four hours of system maintenance.
- The CPU utilization target is 80%+.

For the last couple of releases, the team felt that the bar should be raised when the development cycle is approaching GA. Therefore, the target from three months to GA is now 18 CPU hours per system per day. As the chart shows, a key element of this metric, in addition to actual versus target, is release-to-release comparison. One can observe that the V4R2 curve had more data points in the early development cycle, which were at higher CPU utilization levels. It results from conducting pretest runs prior to when the new release content became available. As men-

Figure 8 System crashes/hangs metric



tioned before, the formal stress test starts at the middle of the component test cycle, as the V3R7 and V4R1 curves indicate. The CPU utilization metric is used together with the system crashes and unplanned IPL (initial program load, or "reboot") metric. We will discuss this relationship in the next subsection.

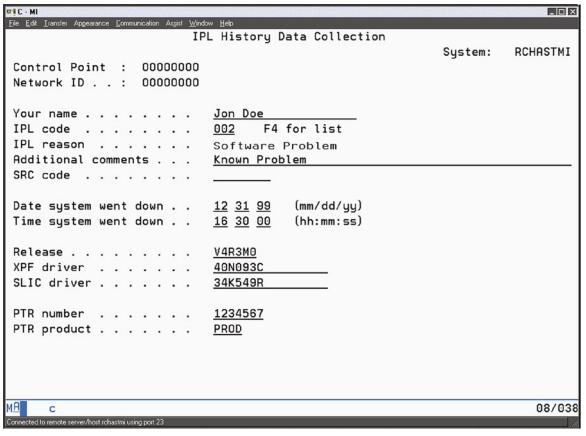
To collect actual CPU utilization data, we rely on a performance monitor tool that runs continuously (a 24-hour-a-day, seven-day-a-week operation) on each test system. Through the communication network, the data from the test systems are sent to an AS/400 nontest system on a real-time basis, and by means of a Lotus Notes** database application, the final data can be easily tallied, displayed, and monitored.

System crashes and unplanned IPLs. Going hand in hand with the CPU utilization metric is the metric for system crashes and unplanned IPLs. For software tests whose purpose is to improve the stability of the system, we need to ensure that the system is stressed and testing is conducted effectively to uncover the latent defects leading to system crashes and hangs or, in general, any IPLs that are unplanned. When such defects are discovered and fixed, stability of the system improves over time. Therefore, the metrics of CPU utilization (stress level) and unplanned IPLs describe the effort aspect and the outcome (findings) aspect of the effectiveness of the test, respectively.

Figure 8 shows an example of the system crashes and hangs metric for the stress test for several AS/400 releases. The target curve, which was derived after V3R7, was developed by applying statistical techniques (for example, a simple exponential model or a specific software reliability model) to the data points from several previous releases including V3R7.

In terms of data collection, when a system crash or hang occurs and the tester re-IPLs the system, the performance monitor and IPL tracking tool will produce a screen prompt and request information about the last IPL. The tester can ignore the prompt temporarily, but it will reappear regularly after a certain time until the questions are answered. One of the screen images of this prompt is shown in Figure 9. Information elicited via this tool includes: test system, network identifier, tester name, IPL code and reason (and additional comments), system reference code (SRC) (if available), data and time system went down, release, driver, PTR number (the defect that causes the system crash or hang), and the name of

Figure 9 Screen image of an IPL tracking tool



the product. The IPL reason code consists of the following categories:

- 001 hardware problem (unplanned)
- 002 software problem (unplanned)
- 003 other problem (unplanned)
- 004 load fix (planned)

Because the volume and trend of system crashes and hangs are germane to the stability performance of the product in the field, we highly recommend this in-process metric for any software for which stability is considered an important attribute. These data should also be used to make release-to-release comparisons in similar time frames prior to GA, and can eventually be used as leading indicators to GA readiness. Although CPU utilization tracking definitely requires a tool, tracking of system crashes and hangs can start with pencil and paper if a proper process is in place.

Mean time to IPL (MTI). Mean time to failure (MTTF) or mean time between failures (MTBF) is the standard measurement in the reliability literature.[5] In the software reliability literature, this metric and various models associated with it have also been discussed extensively. The discussions and use of this metric are predominantly related to academic research or specific-purpose software systems. To the authors' knowledge, implementation of this metric is rare in organizations that develop commercial systems. This may be due to several reasons, including issues related to single-system versus multiple-systems testing, the definition of a failure, the feasibility and cost in tracking all failures and detailed time-related data (note: failures are different from defects or faults because a single defect can cause failures multiple times and in different machines) in large commercial projects during testing, and the value and return on investment of such tracking.

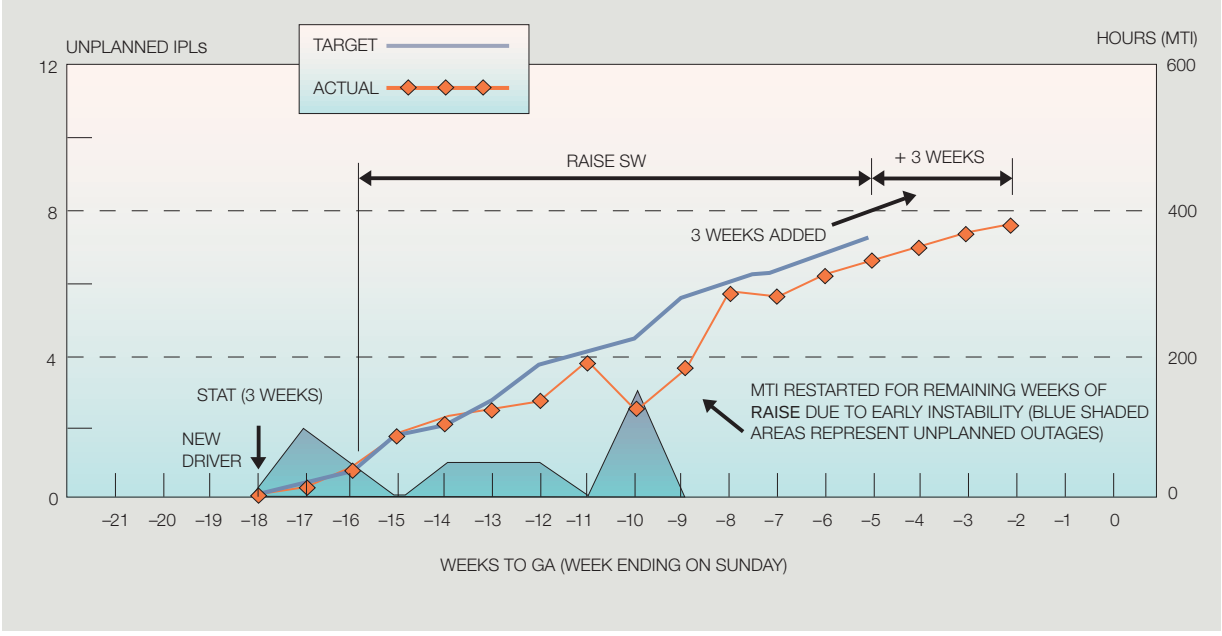
System crashes and hangs and therefore unplanned IPLs (or rebooting) are the more severe forms of failures. Such failures are more clear-cut and easier to track, and metrics based on such data are more meaningful. Therefore, for the AS/400, we use mean time to unplanned IPL (MTI) as the key software reliability metric. This metric is used only during the STAT/RAISE tests time period, which, as mentioned before, is a customer-like system integration test prior to GA. Using this metric for other tests earlier in the development cycle is possible but will not be as meaningful since all the components of the system cannot be addressed collectively until the final system integration test. To calculate MTI, one can simply take the total number of CPU run hours for each week (H_i) and divide it by the number of unplanned IPLs plus 1 ($I_i + 1$). For example, if the total CPU run hours from all systems for a specific week was 320 CPU hours and there was one unplanned IPL caused by a system crash, the MTI for that week would be $320/(1 + 1) = 160$ CPU hours. In the AS/400 implementation, we also apply a set of weighting factors that were derived based on results from prior baseline releases. The purpose of using the weighting factors is to take the outcome from the prior weeks into account so that at the end of the RAISE test (with a duration of 10 weeks), the MTI represents an entire RAISE statement. It is up to the practitioner whether or not to use weighting or how to distribute the weights heuristically. Deciding factors may include: type of products and systems under test, test cycle duration, or even how the testing period is planned and managed.

$$\text{Weekly MTI}_n = \sum_{i=1}^{n} W_i * \left(\frac{H_i}{I_i + 1}\right)$$

where n = the number of weeks that testing has been performed on the current driver (i.e., the current week of test), H = total of weekly CPU run hours, W = weighting factor, and I = number of weekly (unique) unplanned IPLs (due to software failures).

Figure 10 is a real-life example of the MTI metric for the STAT and RAISE tests for a recent AS/400 release. The x-axis represents the number of weeks to GA.

Figure 10 MTI (mean time to unplanned IPL) metric



The y-axis on the right side is MTI and on the left side is the number of unplanned IPLs. Inside the chart, the shaded areas represent the actual number of unique unplanned IPLs (crashes and hangs) encountered. From the start of the STAT test, the MTI metric is shown tracking to plan until week 10 (before GA) when three system crashes occurred during a one-week period. From the significant drop of the MTI, it was evident that with the original test plan, there would not be enough burn-in time for the system to reach the MTI target. Since this lack of burn-in time typically results in undetected critical problems at GA, additional testing was added, and the RAISE test was lengthened by three weeks with the product GA date unchanged. (Note that in the original plan, the duration for STAT and RAISE was three weeks and ten weeks, respectively. The actual test cycle was 17 weeks.) As in the case above, discrepancies between actual and targeted MTI should be used to trigger early, proactive decisions to adjust testing plans and schedules to make sure that GA criteria for burn-in can be achieved, or at a minimum, that the risks are well understood and a mitigation plan is in place.

Additionally, MTI can also be used to provide a "heads-up" approach for assessing weekly in-process testing effectiveness. If weekly MTI results are significantly below planned targets (i.e., 10–15 percent or more), but unplanned IPLs are running at or below expected levels, this situation can be caused by other problems affecting testing progress (just *not* those creating system failures). In this case, low MTI tends to highlight how the overall effect of these individual products or functional problems (possibly being uncovered by different test teams using the same system) are affecting overall burn-in. This is especially true if it occurs near the end of the testing cycle. Without an MTI measurement and planned target, these problems and their individual action plans might unintentionally be assessed as acceptable to exit testing. But to reflect the lack of burn-in time seen as lower CPU utilization numbers, MTI targets are likely to not yet be met. Final action plans might include:

- Extending testing duration or adding resources, or both
- Providing for a more exhaustive regression testing period if one were planned
- Adding a regression test if one were not planned
- Taking additional actions to intensify problem resolution and fix turnaround time (assuming that

there is still enough elapsed time available until the testing cycle is planned to end)

Note that MTI targets and results, because they represent stability, are only as reliable as the condition under which the system(s) was tested. MTI does not substitute for a poorly planned or executed test, but instead tends to highlight the need for improvements in such areas as stress testing capabilities (i.e., numbers of real or simulated users, new automation technologies, network bandwidth, understanding system, and testing limitations, etc.), and providing for enough burn-in time—with all testing running at "full throttle"—for test exit criteria to be satisfied.

Another positive side effect from the visibility of reporting weekly MTI results is on the core system components that are used to support the performance monitoring and CPU data collection. These tools are also performing an indirect usage test of many base system functions, since MTI data are also collected or retrieved by a central system on fixed hourly intervals, seven days a week. This requires those key interfaces of the system(s) and network to not only stabilize early, but to remain so for the duration of the testing cycle. Problems in those core areas are quickly noticed, identified, and resolved because of the timing and possible effect on the validity of MTI data.

Critical problems—show stoppers. The show stopper parameter is very important because the severity and impact of software defects varies. Regardless of the volume of total defect arrivals, it only takes a few show stoppers to render the product dysfunctional. Compared to the metrics discussed previously, this metric is more qualitative. There are two aspects to this metric. The first is the number of critical problems over time, with release-to-release comparison. This dimension is as quantitative as all other metrics. More importantly, the second dimension is concerned with the types of critical problems and the analysis and resolution of each of them.

The AS/400 implementation of this tracking and focus is based on the general criterion that any problem that will impede the overall progress of the project or that will have significant impact on a customer's business (if not fixed) belongs to such a list. The tracking normally starts at the middle of component test when a critical problem meeting is held weekly by the project management team (with representatives from all functional areas). When it comes close to system test and GA time, the focus intensifies and daily meetings take place. The objective is to facilitate cross-functional teamwork to resolve the problems swiftly. Although there is no formal set of criteria, problems on the critical problem list tend to be related to install, system stability, security, data corruption, etc. Before GA, all problems on the list must be resolved—either fixed or a fix underway with a clear target date for the fix to be available.

In-process metrics and quality management

On the basis of the previous discussions of specific metrics, we have the following recommendations for implementing in-process metrics in general:

- Whenever possible, it is preferable to use calendar time as the measurement unit for in-process metrics, versus using phases of the development process. There are some phase-based defect models or defect cause or type analysis methods available, which were also extensively used in AS/400. However, in-process metrics and models based on calendar time provide a direct statement on the status of the project with regard to whether it can achieve on-time delivery with desirable quality. As appropriate, a combination of time-based metrics and phase-based metrics is desirable.
- For time-based metrics, use the date of product shipment as the reference point for the x-axis and use weeks as the unit of measurement. By referencing the GA date, the metric portrays the true in-process status and conveys a "marching toward completion" message. In terms of time units, we found that data at the daily level proved to have too much fluctuation, and data at the monthly level lost their timeliness; neither can provide a trend that can be spotted easily. Weekly data proved to be optimal in terms of both measurement trends and cycles for actions. Of course, when the project is approaching the back-end critical time period, some metrics need to be monitored and actions taken at the daily level. Examples are the status of critical problems near GA and the PTR backlog a couple of weeks prior to the cut-off date for the GA driver.
- Metrics should be able to tell what is "good" or "bad" in terms of quality or schedule in order to be useful. To achieve these objectives, historical comparisons or comparisons to a model are often needed.
- Some metrics are subject to strong management actions; for others there should be no intervention. Both types should be able to provide meaningful

indications of the health of the project. For example, defect arrivals, which are driven by testing progress and other factors, should not be artificially controlled, whereas the defect backlog is completely subject to management and control.
- Finally, the metrics should be able to drive improvements. The ultimate question for metric work is what kind and how much improvement will be made and to what extent the final product quality will be influenced.

With regard to the last item in the above list, it is noted that to drive specific improvement actions, sometimes the metrics have to be analyzed at a more granular level. As a real-life example, in the case of the testing progress and PTR backlog metrics, the following was done (analysis and guideline for action) for an AS/400 release near the end of development testing (Component Test, or CT, equivalent to Functional Verification Test [FVT] in other IBM laboratories).

Components that were behind in CT were identified using the following methods:

1. Sorting all components by "percentage attempted of total test cases" and selecting those that are less than 65 percent. In other words, with less than three weeks to completion of development testing, these components have more than one-third of CT left.
2. Sorting all components by "number of planned cases not attempted" and selecting those that are 100 or larger, then adding these components to those from 1. In other words, these several additional components may be on track or not seriously behind percentage-wise, but because of the large number of test cases they have, there is a large amount of work left to be done.

Because the unit (test case, or test variation) is not of the same weight across components, 1 was used as the major criterion, supplemented by 2.

Components with double-digit PTR backlogs were identified. Guidelines for actions are:

1. If CT is way behind and PTR backlog is not too bad, the first priority is to really ramp-up CT.
2. If CT is on track and PTR backlog is high, the key focus is on PTR backlog reduction.
3. If CT is way behind and PTR backlog is high, these components are really in trouble. *Get help* (e.g., extra resources, temporary help from other components, or teams, or areas).
4. For the rest of the components, continue to keep a strong focus on both finishing up CT and PTR backlog reduction.

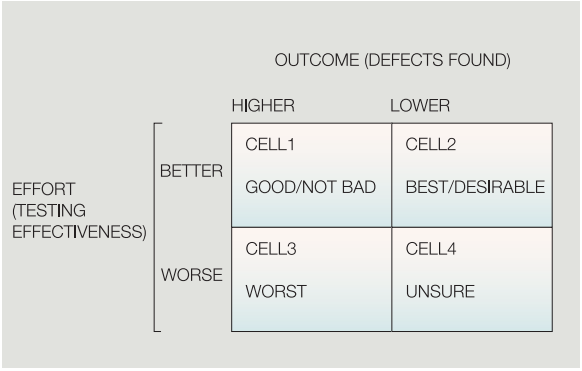
Furthermore, analysis on defect cause, symptoms, defect origin (in terms of development phase) and where found can provide more information for possible improvement actions. For examples of such analyses and the discussion on phase defect removal effectiveness (design reviews, code inspections, and test), see Kan.[4]

Metrics are a tool for project and quality management. In software development as well as many types of projects, commitment by the teams is very important. Experienced project managers know, however, that subjective commitment is not enough. Is there a commitment to the system schedules and quality goals? Will delivery be on time with desirable quality? Even with eyes-to-eyes commitment, these objectives are often not met because of a whole host of reasons, right or wrong. In-process metrics provide the added value of objective indication. It is the combination of subjective commitments and objective measurement that will make the project successful.

To successfully manage in-process quality (and therefore the quality of the final deliverables), in-process metrics must be used effectively. We recommend an integrated approach to project and quality management vis-a-vis these metrics, in which quality is managed as importantly as other factors such as schedule, cost, and content. Quality should always be an integral part of the project status report and checkpoint reviews. Indeed, many of the examples we give in this paper are metrics for both quality and schedule (those weeks to GA measurements) as the two parameters are often intertwined.

One common observation with regard to metrics in software development is that the project teams often explain away the negative signs as indicated by the metrics. There are two key reasons for this. First, in practice there are many poor metrics, which are further complicated by abundant abuses and misuses. Second, the project managers and teams do not take ownership of quality and are not action-oriented. Therefore, the effectiveness, reliability, and validity of metrics are far more important than the quantity of metrics. And once a negative trend is observed, an early urgency approach should be taken in order

Figure 11 Example of an effort-outcome matrix



to prevent schedule slips and quality deterioration. Such an approach can be supported via in-process quality targets, i.e., once the measurements fall below a predetermined target, special actions will be triggered.

Effort/outcome paradigm. From the discussions thus far, it is clear that some metrics are often used together in order to provide adequate interpretation of the in-process quality status. For example, testing progress and PTR arrivals, and CPU utilization and the number of system crashes and hangs are two obvious pairs. If we take a closer look at the metrics, we can classify them into two groups: those that measure the testing effectiveness or testing effort, and those that indicate the outcome of the test expressed in terms of quality. We call the two groups the effort indicators (e.g., test effectiveness assessment, test progress S curve, CPU utilization during test) and the outcome indicators (PTR arrivals—total number and arrivals pattern, number of system crashes and hangs, mean time to IPL), respectively.

To achieve good test management, useful metrics, and effective in-process quality management, the effort/outcome paradigm should be utilized. The effort/outcome paradigm for in-process metrics was developed based on our experience with AS/400 software development and was first introduced by Kan.[4] Here we provide further details. We recommend the 2×2 matrix approach such as the example shown in Figure 11.

For testing effectiveness and the number of defects found, cell 2 is the best-case scenario because it is an indication of the good intrinsic quality of the design and code of the software—low error injection during the development process. Cell 1 is also a good scenario; it represents the situation that latent defects were found via effective testing. Cell 3 is the worst case because it indicates buggy code and probably problematic designs—high error injection during the development process. Cell 4 is the unsure scenario because one cannot ascertain whether the lower defect rate is a result of good code quality or ineffective testing. In general, if the test effectiveness does not deteriorate substantially, lower defects are a good sign.

It should be noted that in the matrix, the better/worse and higher/lower designation should be carefully determined based on release-to-release or actual versus model comparisons. This effort/outcome approach also provides an explanation of Myers' counterintuitive principle of software testing as discussed earlier.

The above framework can be applied to pairs of specific metrics. For example, if we use testing progress as the effort indicator and the PTR arrivals pattern as the outcome indicator, we obtain the following scenarios:

Positive scenarios:

- Testing progress is on or ahead of schedule, and PTR arrivals are lower throughout the testing cycle (compared with the previous release).
- Testing progress is on or ahead of schedule, and PTR arrivals are higher in the early part of the curve—chances are the PTR arrivals will peak earlier and decline to a lower level (compared with the previous releases).

Negative scenarios:

- Testing progress is significantly behind schedule, and PTR arrivals are higher—chances are the PTR arrivals will peak later and the problem of late cycle defect arrivals will emerge.
- Testing progress is behind schedule, and PTR arrivals are lower in the early part of the curve—this is an unsure scenario, but from the testing effort point of view it is not acceptable.

The interpretation of the pair of metrics for system stability (CPU utilization and system crashes and hangs) is similar.

Generally speaking, outcome indicators are more common, whereas effort indicators are more difficult to establish. Furthermore, different types of software and tests may need different effort indicators. Nonetheless, the effort/outcome paradigm forces one to establish appropriate effort measurements, which, in turn, drive the improvements in testing. For example, the metric of CPU utilization is a good effort indicator for operating systems. In order to achieve a certain level of CPU utilization, a stress environment needs to be established. Such effort increases the effectiveness of the test.

For integration type software where a set of vendor software is integrated together with new products to form an offering, effort indicators other than CPU stress level may be more meaningful. One could look into a test coverage-based metric, including the major dimensions of testing such as:

- Setup
- Install
- Minimum/maximum configuration
- Concurrence
- Error recovery
- Cross-product interoperability
- Cross-release compatibility
- Usability
- DBCS

A five-point score (one being the least effective and five being the most rigorous testing) can be assigned for each dimension, and the sum total can represent an overall coverage score. Alternatively, the scoring approach can include the "should be" level of testing for each dimension and the "actual" level of testing per the current testing plan based on independent assessment by experts. Then a "gap score" can be used to drive release-to-release or project-to-project improvement in testing. For example, assuming the testing strategy for an offering calls for the following dimensions to be tested, each with a certain sufficiency level: setup (5), install (5), cross-product interoperability (4), cross-release compatibility (5), usability (4), and DBCS (3). Based on expert assessment of the current testing plan, the sufficiency levels of testing are: setup (4), install (3), cross-product interoperability (2), cross-release compatibility (5), usability (3), and DBCS (3). Therefore, the "should be" level of testing would be 26 and the "actual" level of testing would be 20, with a gap score of 6. This approach may be somewhat subjective, but it also involves the experts in the assessment process—those who can make the difference. Although it would not be easy in a real-life implementation, the point here is that the effort/outcome paradigm and the focus on effort metrics have direct linkage to improvements in testing. In AS/400, there was only limited experience in this approach while we were improving our product level test. Further research in this area or implementation experience will be useful.

How you know your product is good enough to ship

Determining when a product is good enough to ship is a complex issue. It involves the types of products (for example, shrink-wrap application versus an operating system), the business strategy related to the product, market opportunities and timing, customers' requirements, and many more factors. The discussion here pertains to the scenario that quality is an important consideration and that on-time delivery with desirable quality is the major project goal.

A simplistic view is that a target is established for one or several in-process metrics, and when the target is not met, the product is not shipped per schedule. We all know that this rarely happens in real life, and for legitimate reasons. Quality measurements, regardless of their maturity levels, are never as black and white as meeting or not meeting a GA date. Furthermore, there are situations where some metrics are meeting targets and others are not. There is also the question: If targets are not being met, how bad is the situation? Nonetheless, these challenges do not diminish the value of in-process measurements; they are also the reason for improving the maturity level of software quality metrics.

When various metrics are indicating a consistent negative message, it is then evident that the product may not be good enough to ship. In our experience, indicators from all of the following dimensions should be considered together to get an adequate picture of the quality of the product to be designated GA:

- System stability/reliability/availability
- Defect volume
- Outstanding critical problems
- Feedback from early customer programs
- Other important parameters specific to a particular product (ease of use, performance, install, etc.)

In Figure 12, we present a real-life example of an assessment of in-process quality of a recent AS/400 release when it was near GA. The summary tables

Figure 12 Quality assessment summary—an example

INDICATOR	OBSERVATION	VS. V3R7	VS. V4R2	ASSESSMENT
CT/CTR PROGRESS	BASE COMPLETE. PRODUCT X TO COMPLETE 7/31.	↔	↔	OK
PTR ARRIVALS	PEAK EARLIER THAN V4R2 AND V3R7, AND LOWER BACK END—FOR BOTH ABSOLUTE NUMBERS AND NORMALIZED (TO SIZE) RATES.	↑	↑	GOOD
PTR SEVERITY DISTRIBUTION	LOWER THAN V3R7 AND V4R2 AT BACK END.	↑	↑	GOOD
PTR BACKLOG	EXCELLENT BACKLOG MANAGEMENT, LOWER THAN V4R2 AND V3R7 AND ACHIEVED TARGETS AT PTF CONTROL AND HIDING. *NEEDS FOCUS FOR FINAL CUM.*	↔	↔	OK
# PENDING PTFs	ABOVE V4R2 AT SAME TIME. *NEED FOCUS TO MINIMIZE CUSTOMER REDISCOVERY.*	↔	↓	OK
CRITICAL PROBLEM LIST	STRONG PROBLEM MANAGEMENT. NUMBER OF PROBLEMS ON THE CRITICAL LIST SIMILAR TO V4R2.	↑	↑	GOOD
SYSTEM STABILITY - UNPLANNED IPLS - CPU RUN TIME	STABILITY SIMILAR TO, MAYBE SLIGHTLY BETTER THAN, V4R2.	↑	↑	OK
PLAN CHANGE	V4R3 PLAN CHANGES NOT AS PERVASIVE AS V4R2.	N/A	↑	OK
TIMELINESS OF NLS/MRI	EARLY AND PROACTIVE BVT DAILY MEETINGS. NLV BVT BEHIND, BUT SCHEDULES ACHIEVABLE.	↔	↔	OK

outline the parameters or indicators used (column 1), a brief description of the status (column 2), release-to-release comparisons (columns 3 and 4), and an assessment (column 5). Although some of the indicators and assessments are based on subjective information, for many parameters we have formal metrics and data in place to substantiate the assessment. The assessment was done about two months before GA, and as usual, a series of actions was taken throughout GA. The release now has been in the field for over two years and has demonstrated excellent field quality.

Conclusion

In this paper we discuss a set of in-process metrics for the testing phases of the software development process. We provide real-life examples based on implementation experiences of the IBM Rochester AS/400 development laboratory. We also describe the effort/outcome paradigm as a framework for establishing and using in-process metrics for quality management.

There are certainly many more in-process metrics for software testing that we did not include, nor is

Figure 12 Continued

INDICATOR	OBSERVATION	VS. V3R7	VS. V4R2	ASSESSMENT
HARDWARE RAISE AND NORTH STAR	TARGET COMPLETE: 7/31/98. FOCUSING ON BACKLOG REDUCTION.	VS. V4R1 - MAKO/MIL	INVADER	GOOD
HARDWARE RELIABILITY	HARDWARE RELIABILITY PROJECTED TO MEET CI/105 (BE BETTER THAN PRIOR RELEASES) FOR ALL MODELS.	VS. V4R1 - MAKO/MIL	INVADER	GOOD
PLT/LNE	TESTING CONTINUES FOR DATABASE (SAP), INTERNET (HTTP SERVER), AND LDAP, BUT NO MAJOR PROBLEMS.			OK
INSTALL	PHASE II TESTING AHEAD OF PLAN. ONE OF THE CLEANEST RELEASES IN SIT.			OVERALL: GOOD
SERVICEABILITY/UPGRADE TESTING	*CONCERN W/CONFIGURATOR READINESS AND SOFTWARE ORDER STRUCTURE IN MANUFACTURING.*			*CONCERN*
SOFTWARE STAT/RAISE	RELEASE LOOKS GOOD OVERALL.			OVERALL: GOOD
SERVICE READINESS	WW SERVICE COMMUNITY IS ON TRACK TO BE READY TO SUPPORT V4R3.			OVERALL: GOOD
EARLY PROGRAMS	GOOD EARLY FEEDBACK ON THE RELEASE.			OVERALL: GOOD
MANUFACTURING BUILD AND TEST	STILL EARLY, BUT NO MAJOR PROBLEMS.			OK

our intent to provide comprehensive coverage. Even for those that we discussed here, not each and every one is applicable universally. However, we contend that the several metrics that are basic to software testing (such as the testing progress curve, defect arrivals density, and critical problems before GA) should be an integral part of any software testing.

It can never be overstated that it is the effectiveness of the metrics that matters, not the number of metrics available. It is a strong temptation for quality or metrics practitioners to establish more and more metrics. However, in the practice of metrics and measurements in the software engineering industry, abuses and misuses abound. It is ironic that software engineering is relying on the measurement approach to elevate its maturity level as an engineering discipline, yet measurement practices when improperly used pose a big obstacle and provide confusion and controversy to such an expected advance. Ill-founded metrics are not only useless, they are actually counterproductive, adding extra costs to the organization and doing a disservice to the software engineering and quality disciplines. Therefore, we must take a serious approach to metrics. Each metric to be used should be subjected to the examination of basic prin-

ciples of measurement theory. For example, the concept, the operational definition, the measurement scale, and validity and reliability issues should be well thought out. At a macro level, an overall framework should be used to avoid an *ad hoc* approach. We discuss the effort/outcome framework in this paper, which is particularly relevant for in-process metrics. We also recommend the Goal/Question/Metric (GQM) approach in general for any metrics.[6–8]

At the same time, to enhance success, one should take a dynamic and flexible approach, for example, tailor-make the metrics to meet the needs of a specific team, product, and organization. There must be "buy-in" by the team (development and testing) in order for the metrics to be effective. Metrics are a means to an end—the success of the project—not an end itself. The project team should have intellectual control and a thorough understanding of the metrics and data that they use, and should therefore make the right decisions. Although good metrics can serve as a useful tool for software development and project management, they do not automatically lead to improvement in testing and in quality. They do foster data-based decision-making and provide objective criteria for actions. Proper use and continued refinement by those involved (for example, the project team, the testing community, and the development teams) are therefore crucial.

Acknowledgment

This paper is based on a white paper written for the IBM Software Test Council. We are grateful to Carl Chamberlin, Gary Davidson, Brian McCulloch, Dave Nelson, Brock Peterson, Mike Santivenere, Mike Tappon, and Bill Woodworth for their reviews and helpful comments on an earlier draft of the white paper. We wish to thank the three anonymous reviewers for their helpful comments on the revised version for the *IBM Systems Journal*. We wish to thank the entire AS/400 software team, especially the release managers, the extended development team members, and the various test teams who made the metrics described in this paper a state of practice instead of just a theoretical discussion. A special thanks goes to Al Hopkins who implemented the performance monitor tool on AS/400 to collect the CPU utilization and unplanned IPL data.

Cited references

1. M. C. Paulk, C. V. Weber, B. Curtis, and M. B. Chrissis, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley Longman, Inc., Reading, MA (1994).
2. W. S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley Longman, Inc., Reading, MA (1995).
3. G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, Inc., New York (1979).
4. S. H. Kan, *Metrics and Models in Software Quality Engineering*, Addison-Wesley Longman, Inc., Reading, MA (1995).
5. P. A. Tobias and D. C. Trindade, *Applied Reliability*, Van Nostrand Reinhold Company, New York (1986).
6. V. R. Basili, "Software Development: A Paradigm for the Future," *Proceedings of the 13th International Computer Software and Applications Conference (COMPSAC)*, keynote address, Orlando, FL (September 1989).
7. Materials in the Software Measurement Workshop conducted by Professor V. R. Basili, University of Maryland, College Park, MD (1995).
8. N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, *2nd edition*, PWS Publishing Company, Boston (1997).

Accepted for publication September 15, 2000.

Stephen H. Kan *IBM AS/400 Division, 3605 Highway 52 N, Rochester, Minnesota 55901 (electronic mail: skan@us.ibm.com).* Dr. Kan is a Senior Technical Staff Member and a technical manager in programming at IBM in Rochester, Minnesota. He is responsible for the Quality Management Process in AS/400 software development. His responsibility covers all aspects of quality ranging from quality goal setting, supplier quality requirements, quality plans, in-process metrics, quality assessments, and CI105 compliance, to reliability projections, field quality tracking, and customer satisfaction. Dr. Kan has been the software quality focal point for the software system of the AS/400 since its initial release in 1988. He is the author of the book *Metrics and Models in Software Quality Engineering*, numerous technical reports, and articles and chapters in the *IBM Systems Journal*, *Encyclopedia of Computer Science and Technology*, *Encyclopedia of Microcomputers*, and other professional journals. Dr. Kan is also a faculty member of the University of Minnesota Master of Science in Software Engineering (MSSE) program.

Jerry Parrish *IBM AS/400 Division, 3605 Highway 52 N, Rochester, Minnesota 55901 (electronic mail: parrishj@us.ibm.com).* Mr. Parrish is an advisory software engineer and has been a member of the IBM Rochester AS/400 System Test team since the late 1980s. He is currently the test lead for the AS/400 Software Platform Integration Test (also known as RAISE). As part of the overall focus of the AS/400 on system quality, he has applied his expertise since the early 1990s to focus on the areas of test process methodology and applied models and metrics. Over this period, he has helped transform the System Test group in Rochester into a significant partner in IBM Rochester's total quality commitment. Mr. Parrish holds an undergraduate degree in computer science. He has been employed by IBM since 1980 with experience in software development, software system testing, and product assurance.

Diane Manlove *IBM AS/400 Division, 3605 Highway 52 N, Rochester, Minnesota 55901 (electronic mail: dmanlove@us.ibm.com).*

Ms. Manlove is an advisory software quality engineer for IBM in Rochester. Her responsibilities include management of release quality during product development, system quality improvement, and product quality trend analysis and projections. Ms. Manlove is certified by the American Society for Quality as a Software Quality Engineer, a Quality Manager, and a Reliability Engineer. She holds a master's degree in reliability and an undergraduate degree in engineering. She has been employed by IBM since 1984 and has experience in test, manufacturing quality, and product assurance.