SOFTWARE QUALITY ASSURANCE FOR MATHEMATICAL MODELING SYSTEMS

Michael R. Bussieck, Steven P. Dirkse, Alexander Meeraus, Armin Pruessner
GAMS Development Corporation, 1217 Potomac Street NW, Washington, DC
{MBussieck, SDirkse, AMeeraus, APruessner}@gams.com

Abstract With increasing importance placed on standard quality assurance methodologies by large companies and government organizations, many software companies have implemented rigorous quality assurance (QA) processes to ensure that these standards are met. The use of standard QA methodologies cuts maintenance costs, increases reliability, and reduces cycle time for new distributions. Modeling systems differ from most software systems in that a model may fail to solve to optimality without the modeling system being defective. This additional level of complexity requires specific QA activities. To make software quality assurance (SQA) more cost-effective, the focus is on reproducible and automated techniques. In this paper we describe some of the main SQA methodologies as applied to modeling systems. In particular, we focus on configuration management, quality control, and testing as they are handled in the GAMS build framework, emphasizing reproducibility, automation, and an open-source public-domain framework.

Keywords: Quality assurance; software; modeling systems; mathematical programming, automation

1. Introduction

Quality Assurance (QA) has become an essential component in most industrial and commercial undertakings and has increasingly become important in most software engineering sectors. See for example, the emergence of organizations focusing specifically on quality ([10] and [2]). Unfortunately, software quality assurance (SQA) has received much less attention in the Mathematical Programming (MP) community. Historically, many innovative solver technologies have emerged from the academic sector, where the emphasis has generally been on *performance* and not on QA as in the commercial sector. On the other hand, the commercial sector has always emphasized *reliability* as the primary goal. Given this focus on performance in the (academic) MP community, it is not surprising, that the few papers addressing SQA methodologies in

MP focus mostly on the areas of performance testing and benchmarking. See for example [17], [5], [4], and [11]. Bussieck et. al. [3] have examined the QA steps necessary for reducing the risks of introducing new solver technologies into the community, although many of these procedures and processes focused on full system integration performance-type testing as well.

Traditional, commercial SQA techniques have always emphasized *full life-cycle testing* (as opposed to system integration testing only) and usually rely heavily on *auditing* and *peer-review* inspections. While these techniques are no doubt effective, the latter is quite expensive, and given the small MP industry, economically prohibitive. Note that many commercial MP companies fall into the small business category or into specialized research groups within larger organizations. Therefore, the focus of SQA activities for MP must rely on *reproducible and automated tools and testing*. We will focus on such activities, emphasizing how such tools and procedures improve overall product quality, reduce cycle time for new distributions, and reduce turnaround time for bug fixes. Furthermore, we place many of these tools in the *public domain* (most in the form of models) for use by customers and researchers alike.

The public availability of such models and testing tools has several advantages. In the absence of formal peer review and auditing activities, the use of public domain models and testing tools is important in that it allows customers and the MP community to become directly involved in the QA validation process. It also speeds up the dissemination of knowledge in the areas of algorithms and solver technologies from academics by allowing them to have direct access to public domain quality assurance tools. Finally, testing activities without the possibility of reproducibility are essentially meaningless since there is no verification ability of specific quality tests by customers at a future date.

While initial formal SQA processes were pushed for by commercial demand, the use of such methods and tools should be of interest to academics, MP software vendors, individual commercial users, as well as commercial companies. Academic users may be interested in performance testing and benchmarking, whereas a commercial client may need to verify that the third party MP software they receive is of quality[1]. The latter may have its own QA department for their products and domain-specific services, but needs assurance of the modeling system to satisfy their customers. Solver vendors want their solver technology to perform well in the market place and want assurance that the modeling system functions appropriately with their respective product. Fi-

[1]There is a certain imbalance between publication in MP and jobs. Consider the unscientific quick-and-dirty study of finding jobs at Careerbuilder.com by keywords (May 12, 2004): “Quality Assurance” 3,556 results, and “Mathematical Programming” 8, a ratio of 444 to 1. A Google search by keyword on the same date results in 4,510,000 for “Quality Assurance” and 111,000 for “Mathematical Programming, a ratio of 40 to 1.

nally the individual user is interested in solving models accurately and to be able to focus on modeling instead of the solve process.

Modeling systems (and numerical software in general) differ from most software systems in that a model may fail to solve to optimality without the modeling system being defective. This *algorithm failure* differs from the traditional *software implementation defect* since the modeling system must be able handle the failure mode and provide the user with sufficient return information to determine the return state. Note that many solvers are available to developers of modeling systems only in library form (i.e. no source), essentially limiting interaction between the modeling system and the solver to black box input-output communication. This additional level of complexity requires *MP-specific QA activities* to test for such returns.

This paper is organized as follows: in §2 we describe general SQA principles, focusing on configuration management, testing, and quality control. In §3 we describe QA activities specific to modeling systems, emphasizing how components interact and where errors can occur. In §4 we show how these principles are applied in the GAMS build framework and how the use of client models can further improve quality. In §5 we give examples of client model testing activities and finally in §6, we draw conclusions.

2. Software Quality Assurance Principles

Various software quality assurance principles (or models) have been developed by different organizations to ensure that specific standards are met and to give guidelines on achieving these standards. Although these address the full software lifecycle, we will focus on configuration management and automated testing.

Standard Models

Many *QA standards and models* exist with a large number of choices. According to [12] "there are more than 300 standards developed and maintained by more than 50 different organizations." Popular models are the ISO 9001, which specifies requirements for a quality management system within an organization and the Software Engineering Institute (SEI) Capability Maturity Model (CMM), which provides a framework for continuous software process improvement [16], although many others are used, depending on user goals. The key notion is that they provide guidelines for conducting audits, testing activities, and for process improvement.

The CMM approach classifies the maturity of the software organization and practices into five levels describing an evolutionary process from chaos to discipline [16]:

- ***Level 1: Initial.*** The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.

- ***Level 2: Repeatable.*** Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

- ***Level 3: Defined.*** The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.

- ***Level 4: Managed.*** Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.

- ***Level 5: Optimizing.*** Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

The challenge for many MP vendors is to move from Level 1, the chaotic, creative and exciting phase to Level 5 without losing creativity and, most importantly, to stay in business.

Configuration Management

Software configuration management (SCM) refers to all activities used to (1) identify change, (2) control change, (3) ensure that change is being properly implemented, and (4) to report changes in the software to others who may need to know of them [14]. This includes all activities related to version control and change control.

Change identification is often in the form of some audit string information, which details the product primary version number, minor version number as well as possible incremental bug fix releases or patches. Change control is necessary to ensure that existing source used for previous releases is not overwritten and only authorized personnel can add new source. *Accurate audit information* becomes increasingly important for software consisting of a high number of modules (GAMS currently has over 45), each of which may depend and build on various other modules. Furthermore, configuration management ensures, that authorized changes are actually implemented in the formal build product. Some configuration management elements that are important include

Audit Strings. Audit strings are necessary to determine the exact versions of a particular product, particularly those that rely on several other modules. The audit strings become particularly important when tracking bug reports so that the exact configuration source (of both the actual product and the supporting modules) can be determined and appropriate fixes can be made.

Product versions are consistent. Use of version checking tools to determine if the version used in the previous official distribution, the latest version in the product repository, the version used for the build in the Makefile and the version used in the audit string are identical. The accuracy and unison of versions is necessary for bug tracking as well (See audit strings).

Product versions are frozen automatically for builds. This ensures that developers do not accidentally overwrite existing code and informs them if versions need to be bumped. Because of the large number of products that exist in the GAMS portfolio, each with numerous versions, automation tools (scripts) for tasks such as making current product directories read-only or read-write become increasingly important.

Quality Control and Testing

All testing activities should include processes to uncover defects during the complete life cycle of the software. The focus on *full life cycle testing* (as opposed to system integration testing only) is important, because the cost of defects rises exponentially the later the phase of the cycle [14]. The testing activities include, but are not limited to:

Unit testing. Testing of the individual component (solver and solver link module) using both black-box (input-output only) and white-box (known internal code structure) type tests.

Regression testing. Testing to determine if changes to the software or fixing a defect cause any problems to other components in the system.

System Integration Testing. Testing done to ensure that the entire product (base module plus solver modules) functions as intended and to specifications.

The emphasis of all testing activities is again on automation and reproducibility. Many of the tools are available publicly in the form of model libraries. The models in these libraries can be used by just running them within GAMS. In [3], some of these tools and models that are publicly available are already described.

Metrics

Most engineering processes involve measurements to make accurate assessments of the attributes of the product. The use of metrics is important in that it quantifies attributes of a given process or of the product. In particular, metrics such as number of defects of a particular type (critical, serious, cosmetic, etc.) are used within GAMS to determine if the product is ready to move from beta phase to a shippable product. Since development takes place continuously, metrics should be collected continuously as well to determine quality of the current product.

3. SQA For Modeling Languages and Systems

In this section we describe some of the *special* problems of maintaining software quality in the context of modeling systems which differs from traditional QA principles. Furthermore, we give some background on modeling systems in general to motivate the QA principles.

Basic Technical Principles

In the early days of mathematical programming systems, the existing techniques to construct, manipulate, and solve models required several manual, time-consuming, and error-prone translations into the different, problem-specific representations required by each solution method. Furthermore, the solution methods were usually tied to a specific architecture and platform and portability of models was virtually nonexistent.

Learning from these early techniques, most modeling languages today, including LINDO [15], GAMS [1], and AMPL [7], appearing in that chronological order, adhere to the following basic technical principles: (1) separation of model and solution methods, (2) computing platform independence, and (3) multiple solvers, platforms, and model types. The adherence to these principles has many advantages which are described, for example, in [3].

Description of Components

In order to understand where possible defects can occur, an overview of the system architecture of modeling systems is necessary. The *base module* (or execution system) is the core of the system and is designed to translate the human-level algebraic model statements into the scalar formulation passed on to the *solver modules*. The base module includes the language compiler, which performs syntax and other checks to ensure the (grammatical) integrity of the model formulation. If the compilation is successful, the base module expands the model formulation and generates a (sparse) matrix to be passed on to the solver module to solve the problem. Additional user-specified solver options are passed on to the solver modules as well. The solver modules are essentially black box modules which return solve and model status as well as solution information (if it exists). Finally, there exist other *external modules*, which are not solvers, but are linked to the base module in a similar manner, to perform specific tasks. These include, for example, links to Matlab, Excel or data base interfaces, or conversion modules such as CONVERT [3] to translate the model into other modeling language formats or standards.

Chance of Failure

The reliability of a system is sometimes referred to as "mean time to failure." Although in traditional SQA terminology failure implies defect, in the MP world this refers to algorithmic failure to find a solution. While phenomenal progress has been made and we are now able to solve many difficult and large-scale problems, the steady state of mathematical programming software should *conservatively* still be assumed to be the failure state. Unlike other software systems, such as database applications, control systems, or other types of systems, where ever more detailed specifications and detailed testing can continually reduce the chance of defect (hypothetically to zero), no such paradigm for failure-free solves exists for optimization (or other numerical) software. In particular, it is likely that there will always exist models, which cannot be solved reliably. Indeed, many of us have likely experienced unpredictable behavior in models, where a change of a single equation or data item suddenly proves to be the culprit in failing to solve. The use of SQA techniques can help minimize this chance of failure and provide graceful return information if no solution is found. It is because of this chance of failure, that commercial modeling systems must focus on reliability, in particular by providing informative return information in case of failure. This level of QA complexity is in *addition* to QA activities associated with traditional defect prevention.

Defects (as opposed to failures) can occur at any phase of the process flow and can result in catastrophic malfunction or undesirable return information if a solution is not found. The defects can occur either in the compilation phase, the data manipulation phase, the model generation (matrix generation) phase, the solve phase by the solver modules, in the solve phase by the sub-solver modules (for example, an MINLP solver may make use of an NLP solver), or during the processing of the solution returned by the solver to the base module. A solve may initiate the execution of other modeling system components. At a minimum, any rigorous quality control testing procedures must address the possibility of failure at each of these phases nested several levels deep.

Uniform Return Status

Because of the inherent chance of failure, modeling systems must be adept at dealing with various return states. Unfortunately, solvers are not uniform in the amount or type of information returned to the base module. At a bare minimum this information includes the solution point, if it has been found, but could also include dual information or infeasibilities if the model was found to be infeasible. In order to ensure uniform return information, GAMS requires return codes for both the model and the solver from which one can accurately deduce the overall return state.

The *solver return status* refers to the status of the solver, for example normal, resource (time) interrupt, iteration interrupt, no solution, and error. For simplicity, numerical codes from 1-13 are assigned to each of these states. The *model return status* refers to the state of the entire model or of a particular point: for example, optimal solution, locally optimal, integer solution, infeasible, locally infeasible, unbounded, or error. These are mapped from 1-19. The status of error (13) should ideally never be returned. Rather, any return status should be mapped more specifically to well-defined return status. As described later in this paper, the error return code is to be treated as a serious bug in the software. It should be clear that without uniform return codes obtained from each solver call, regardless of the solver module, it may become increasingly difficult to do consistent error checking.

Status code analysis is handled in GAMS by use of *matrix filtering*. Through careful analysis, acceptable model/solve return status code combinations have been pre-determined and are ranked (a higher value indicates a better return state than a lower value). Return codes can be used in system integration testing by using matrix filtering: flagging all model/solver combinations with model/solver status code combinations having a ranking that is less than the specified threshold. Roughly, the higher the threshold, the stricter the testing pass criteria.

In Table 1, we show all possible acceptable return code combinations with their respective ranking. Solve status codes are in the columns and model status codes on the rows. For example a Model Status / Solve Status code combination of optimal solution / normal completion (1,1) has a high ranking of 9, whereas a combination of Error no Solution / Error Solver Failure (13,13) is never acceptable (denoted by a dot .).

4. Software Quality Assurance in GAMS

GAMS is available on 7 different platforms (Windows, Linux and 5 other UNIX environments) and consists of over 45 different modules and products. Thus the build process is quite complex and requires a single standard repository from which source and libraries are extracted. Software builds for the dif-

Table 1. Matrix Filtering. Ranking of model and solver status return codes

	Model Status	NormalCompletion	IterationInterrupt	ResourceInterrupt	SolverInterrupt	EvalErrorInterrupt	CapabilityProblem	LicensingProblem	UserInterrupt	SetupFailure	SolverFailure	SolverError	SolverSkipped	SystemFailure
1	Optimal	9	.	.	.	.	.	.	.	.	.	.	.	.
2	LocallyOptimal	9	9	9	9	9	.	.	9	.	.	.	.	.
3	Unbounded	9	.	.	.	.	.	.	.	.	.	.	.	.
4	Infeasible	9	.	.	.	.	.	.	.	.	.	.	.	.
5	LocallyInfeasible	9	.	.	.	.	.	.	.	.	.	.	.	.
6	intermediateInfeasible	.	5	5	5	5	.	.	5	.	.	.	.	.
7	IntermediateNonoptimal	.	5	5	5	5	.	.	5	.	.	.	.	.
8	IntegerSolution	9	9	9	9	9	.	.	9	.	.	.	.	.
9	Intermediate_NonInteger	.	5	5	5	5	.	.	5	.	.	.	.	.
10.	IntegerInfeasible	9	.	.	.	.	.	.	.	.	.	.	.	.
11	Licensing_NoSolution	.	.	.	.	.	.	5	.	.	.	.	.	.
12	ErrorUnknown	.	.	.	.	.	.	.	.	.	.	3	.	3
13	Error_NoSolution	.	.	.	.	.	.	.	.	3	3	.	3	.
14	NoSolution	4	4	4	4	4	6	.	4	.	.	.	3	.
15	SolvedUnique	9	.	.	.	.	.	.	.	.	.	.	.	.
16	Solved	9	.	.	.	.	.	.	.	.	.	.	.	.
17	SolvedSingular	9	.	.	.	.	.	.	.	.	.	.	.	.
18	Unbounded_NoSolution	9	.	.	.	.	.	.	.	.	.	.	.	.
19	Infeasible_NoSolution	9	.	.	.	.	.	.	.	.	.	.	.	.

ferent platforms take place on different machines, which may exist in-house, or elsewhere.

Because of the complexity of porting to a new machine, which requires multiple compilers and utilities, initialization of a new machine requires installation of various software modules (including specialized GAMS-specific build scripts). This step is unfortunately not automated, although the build environment is relatively stable and new porting machines are only infrequently introduced. The build process can be described as follows for each platform: (1) Copy general build instructions (in the form of Makefiles) from the master repository to local environment, (2) for each product: extract product-specific build instructions and source in the form of Makefiles, build the product, redeposit built product to master repository, (3) do post-build processes to create installation files.

The description of this process is of course simplified herein, although it should be clear that without the appropriate *build automation tools*, the necessary steps to do a single build would be extremely time consuming and the probability for failure high. Furthermore, because of dependencies of some products on other products, it becomes increasingly difficult to sort through dependencies manually.

The porting environment allows near-automated builds of entire distributions. Indeed, as an added SQA measure, we have fully *automated the build task* for Windows and Linux, so that a full build is completed automatically once a week. Such automated full compilation activities ensures that our porting environment is continually in a *buildable state*. This uncovers potential bugs early in the life cycle and in turn reduces cycle time for full release of new distributions. The automatic build also includes installation and testing activities so a continual analysis of porting source is possible.

Configuration Management

Within the GAMS porting environment, we use configuration management tools to ensure that product versions are consistent and version source integrity through automatic code freezes is maintained. Furthermore, we have a consistent audit string assignment for each module.

All GAMS modules can be easily identified in terms of their version numbers, build date, last source date change, as well as such information for all modules needed to build the product. Sample audit information includes the one line audit string (here for CPLEX)

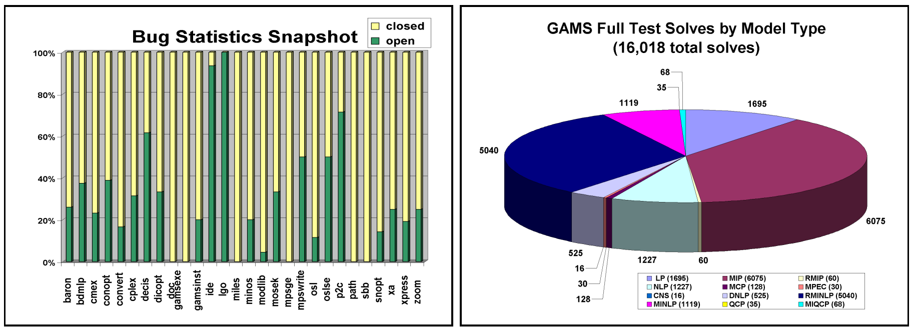
```
GAMS/Cplex    Jan 19, 2004 WIN.CP.NA 21.3 025.027.000.VIS For Cplex 9.0
```

We should note that many standard source management and version control tools exist, notably for example CVS (`http://www.cvshome.org/`). Although we do not utilize any of these, any SQA activities should include rigorous version control processes to manage source. Our processes mainly focus on the automation process in maintaining the integrity of the porting repository. The argument for use of a simpler system is that our products consist not only of source in a single language (such as C++), but contains products written in various languages (or consists of modules written in various languages), libraries, and other tools. Management of these various products and source is simplified by maintaining a simple directory tree structure with simple maintenance scripts. It should also be noted that some of the libraries (or source codes) are obtained from sources using their own source management system. But an argument for management tools such as CVS can certainly be made.

Bug Tracking

At GAMS we use various bug tracking systems in order to communicate effectively with our external solver developers. In-house, the system consists of an e-mail based system which users can submit bug reports to and change statuses. Reports can also be viewed online internally. Because outside solver developers (or clients, if we are dealing with consulting projects) may have

Figure 1. (Left) Snapshot of weekly bug statistics by module (in percent). (Right) GAMS full test. Solve aggregation by model type.



different bug tracking requirements, we also utilize different systems for notification for these parties. Flexibility is the key and we accommodate different systems to coordinate with external contacts.

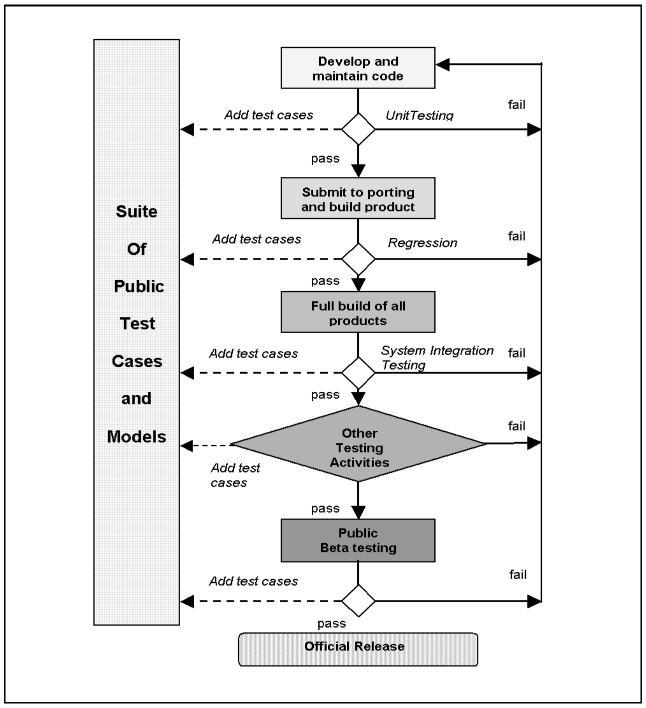
As an example, our in-house bug tracking system sends *weekly e-mail bug statistics* reports of open, closed, and total number of bug reports for each module, which can be used to determine when a stable distribution is ready. In Figure 1 (left) we show a sample weekly statistics chart of open and closed issues (in percent) for each product.

Testing

Our SQA activities focus mainly on testing, in particular on automated and reproducible tests. In Figure 2 we show a flowchart of our general testing activities. The tests cover the full life cycle of development, build and integration. It should be noted that during each testing phase, potential new relevant test cases are added to the master list of test models, which ensures that (1) previous bugs will not occur again, (2) tests can be run automatically during system integration testing, and (3) tests are publicly available.

This means that, for example, during unit testing, if a particular defect is found, the relevant test to uncover the bug is added to the suite of tests, so that during full integration testing, the bug will be uncovered, if it has not been fixed. Such activities are particularly effective in increasing software quality from one build iteration to the next and foster an environment of *continual quality improvement*. Our test cases are available in the following *testing libraries* (or suites), all of which are in the public domain.

Figure 2. GAMS SQA testing activities.



- ***GAMS Model Library.*** A collection of over 300 models, including practical models that can easily be extended to production models. The collection `http://www.gams.com/modlib/modlib.htm`. covers all supported model types.

- ***GAMS Testlib Quality Models.*** A new library of models developed for testing and quality control. These models are designed for use by GAMS staff and solver developers to determine solver correctness, base module correctness, special functions, as well as performance.

- ***GAMS World Models.*** A collection of real-world models, as well as libraries from the academic literature. See `http://www.gamsworld.org/`.

In order to provide uniform access to these tools and models for users of other modeling systems, models can be converted into other common formats using the CONVERT utility (both as a GAMS and an online utility) for use by non-GAMS users. See [3] for details.

Unit testing is usually performed by the developer on a limited integration system on a local development machine. Regression testing may involve a *GAMS initial test*, which solves a sample model of each model type with all available solvers. It also includes the models from Testlib, which check the GAMS base module for possible defects. If the results of this testing activity are satisfactory, the full system integration test is initiated. Tests are run as pass/fail, where pass/fail is determined by the threshold limit specified for model/solver status return code combinations. See Section 3 (Uniform Return Status).

The full system integration test (referred to as *GAMS full test*) involves solving all demo GAMS models from the model library with all relevant solvers and solver combinations. For details, see the GAMS Library model *slvtest.gms* (`http://www.gams.com/modlib/libhtml/slvtest.htm`). This includes a total of over *16,000 total solves* for each platform. An aggregations of solves per model type is shown in Figure 1 (right).

Furthermore, the full test also includes running the full suite of Testlib quality models. These tests include tests to verify proper behavior for failure modes. For example, we may test for domain input violations of nonlinear functions. The number of quality tests run on each platform differs, since not all solvers are available for all platforms and some modules may not exist for certain platforms. For example, the Excel interface is tested only on the Windows platform. The total number of quality test models is about 118, with some containing *numerous* pass/fail tests. New quality tests are added continuously.

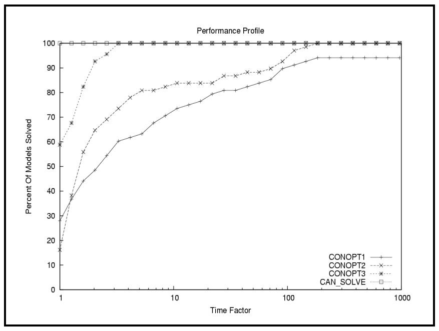
The tests are run in fully automated mode, usually after every full build, with results sent via e-mail to the person building the system. The e-mail includes any possible anomalies, as well as a summary of model and solve status combinations. If there are anomalies, the model and solve combination is revealed, so that the bug can easily be reproduced on a local machine. The e-mail reports also serve the function of archiving test results for future analysis and comparison.

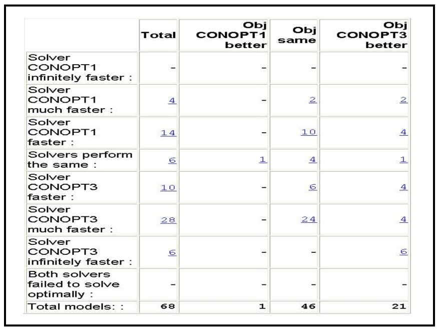
A sample excerpt of the full test summary sent via e-mail is shown below. The results are shown in terms of number of solves falling into each model status and solve status return codes. The absence of a model status of 13 (Error No Solution) or solver status code of 13 (Error System Failure) generally indicates a successful system test. In this case, a threshold of 3 has been set, so that any model / solve status return combinations with a ranking less than or equal to 3 are marked as failures (not shown in the Table). See Table 1 for the model/solver return code ranking matrix.

```
Total SOLVE records = 16018  all return codes

solvestatus  15969  RC= 1  1 NORMAL COMPLETION
solvestatus     15  RC= 2  2 ITERATION INTERRUPT
solvestatus     34  RC= 4  4 TERMINATED BY SOLVER
```

Figure 3. PAVER performance of CONOPT3 with respect to previous versions. (Left) Profile plots. (Right) Timing comparisons.





	Total	Obj CONOPT1 better	Obj same	Obj CONOPT3 better
Solver CONOPT1 infinitely faster :	-	-	-	-
Solver CONOPT1 much faster :	4	-	2	2
Solver CONOPT1 faster :	14	-	10	4
Solvers perform the same :	6	1	4	1
Solver CONOPT3 faster :	10	-	6	4
Solver CONOPT3 much faster :	28	-	24	4
Solver CONOPT3 infinitely faster :	6	-	-	6
Both solvers failed to solve optimally :	-	-	-	-
Total models: :	68	1	46	21

```
modelstatus   8130  RC= 1  1 OPTIMAL
modelstatus   3210  RC= 2  2 LOCALLY OPTIMAL
modelstatus      1  RC= 3  3 UNBOUNDED
modelstatus   2322  RC= 4  4 INFEASIBLE
modelstatus    912  RC= 5  5 LOCALLY INFEASIBLE
modelstatus   1400  RC= 8  8 INTEGER SOLUTION
modelstatus     26  RC=10  10 INTEGER INFEASIBLE
modelstatus      6  RC=15  15 SOLVED UNIQUE
modelstatus     10  RC=16  16 SOLVED
modelstatus      1  RC=19  19 INFEASIBLE-NO SOLUTION
```

Miscellaneous Tests and QA Activities

The testing activities above uncover many deficiencies, although some other issues may not be addressed by these tests. Quality assurance is further enhanced through the following procedures and using the following tools:

- ***New development using standard I/O libraries.*** New solvers can be attached to GAMS using standard FORTRAN, C or Delphi I/O libraries. The use of standard libraries to communicate with GAMS reduces the number of bugs that are introduced, increases reliability, as well as performance. The libraries have been thoroughly tested for robustness, correctness and performance over many years and provide a reliable way to introduce new solver technologies.

- ***Performance.*** Performance is still an important criteria in SQA, both for solver developers and commercial users. Benchmarks run using large test sets, can automatically be examined using automation tools, such as the public-domain PAVER Server [13] to verify solver performance and compare to other solvers. For example, in [6], performance of a new version of CONOPT (CONOPT3) is compared to existing versions (CONOPT1 and CONOPT2). The resulting performance profile plots [4] created automatically using the PAVER Server, verified that CONOPT3 indeed does have tremendous performance increases from previous versions. See the results in Figure 4. The left side shows the performance profiles and the right figure the resource timings of CONOPT1 and CONOPT3. The use of such tools enables users to quickly identify trends of large data sets.

- ***Client models.*** Since GAMS has a heavy commercial client base, which demands reliable software, the use of model library models may not be sufficient. Their integrated models may be interfaced with complex databases and make use of varying modules that interact in specific ways. Thus, GAMS Development has started an initiative of running client models in their full interfaced capacity. This ensures that models will solve on new distributions as they did with previous releases.

- ***Independent Solution Verification.*** The EXAMINER tool [8] can be used for examining points and making an unbiased, independent assessment of their merit. It is also useful when comparing the solutions returned by two different solvers. Finally, a tool like the EXAMINER allows one to examine solutions using different optimality tolerances and optimality criteria in a way that is not possible when working with the solvers directly.

5. Client Model Testing

The use of client models for SQA testing is important because it generally involves much more complex interfacing between different modules and gives us a better idea of real, large-scale commercial applications. It also provides a unique motivation: it gives clients assurance that their optimization applications are compatible with new GAMS system releases and gives the expected results. By expected results, we mean not only the ability to solve, but also to find the same solution[2] and have similar performance in doing so. The latter is important because even minor changes in solvers (for example changes in default settings), may unexpectedly cause tremendous increases in solve time, which may be unacceptable for the particular application time frame. In this section we describe a client application, which we use for our in-house QA client testing.

MARKAL [9] is one of the most widely used energy/environment/economy planning models, playing a central role in Climate Change analysis for numer-

[2]To define what we mean by having the ’same solution’ is often an unexpectedly difficult problem and a source of frustration for the innocent user. In general, it is impossible to exactly reproduce solutions to optimization models between different software releases, computing platforms, or real-time events. The term ’same solution’ needs to be defined for each application.

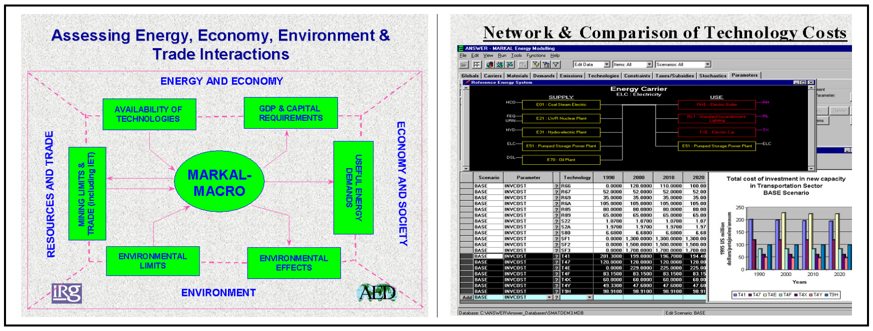
ous countries and communities around the world. Development is coordinated by an international group, the Energy Systems Analysis Program (ETSAP). For current information, see `http://www.etsap.org/markal/main.html`. The model and its data are managed by several different application environments and the model can be operated in different modes offering an almost infinite number of possible model instances. One particular application environment, ANSWER, a graphical interface is shown in Figure 5. Because of the interfacing capabilities, and the large number of components in the MARKAL suite, additional failures are possible, which may not be covered by the simpler models in the testing suites, described previously. The simpler integration testing also does not do any performance analysis, which is often vital in practical commercial modeling applications. A typical model run may involve dozens of optimization steps and intricate data manipulation, all on very large data sets. To make automatic testing possible, substantial restructuring of the internal MARKAL model management is required. Isolation of the core components and communication with the application environment is done via the GAMS DATA Exchange (GDX) format. It is now possible to automatically generate complex job streams that can be operated outside the MARKAL application environment. Those job streams are then used in routine application support and may end up in the customer test suite.

The client model test set consists of a number specific of job streams with additional pass/fail tests designed and implemented jointly with the client. The job streams and all required data and subsystems are archived and become part of a QA support contract. The execution of such test suites is fully automated and QA certificates are generated and archived to allow audits on the QA process. Exceptions are tracked, and a GAMS system will not go into a beta release unless all exceptions have been resolved which, in some cases, may require a redefinition of tests or even making changes in the client models. In a more traditional manufacturing environment, one would call this *preventive* maintenance.

Proprietary and confidential aspects of data and solution processes add some additional complications. In some cases it may be necessary to encrypt certain parts of the models and/or data before a client is ready to share a job stream. GAMS provide a number of tools to extract and transform or hide model and data components to meet the client's need for not disclosing vital information.

The client models such as the MARKAL test suite become the manifestation of the commitment to the continued process of quality assurance for both the client and the software developer. The test suites and the automatic testing procedures are shared by the client and the developer and thus define precisely what quality means for a specific application. Client model testing has become a win for both client and developer and has been made possible by automating the testing process and by sharing the test instances.

Figure 4. MARKAL: (Left) Integrated model overview. (Right) ANSWER user interface, data spreadsheet and graph.


Assessing Energy, Economy, Environment & Trade Interactions
ENERGY AND ECONOMY
AVAILABILITY OF TECHNOLOGIES
GDP & CAPITAL REQUIREMENTS
RESOURCES AND TRADE
MINING LIMITS & TRADE (including IET)
MARKAL-MACRO
USEFUL ENERGY DEMANDS
ECONOMY AND SOCIETY
ENVIRONMENTAL LIMITS
ENVIRONMENTAL EFFECTS
ENVIRONMENT
Network & Comparison of Technology Costs
ANSWER - MARKAL Energy Modelling
File Edit View Run Tools Functions Help
Edit Data
Items: All
Scenarios: All
Globals Carriers Materials Demands Emissions Technologies Constraints Taxes/Subsidies Stochastics Parameters
Reference Energy System
Energy Carrier
ELC : Electricity
SUPPLY
USE
HCO
FEQ
URN
HYD
ELC
DSL
E01 : Coal Steam Electric
E21 : LWR Nuclear Plant
E31 : Hydro-electric Plant
E51 : Pumped Storage Power Plant
E70 : Oil Plant
RHE : Electric Boiler
RL1 : Standard Incandescent Lighting
TXE : Electric Car
E51 : Pumped Storage Power Plant
RH
RL
TX
ELC
Total cost of investment in new capacity in Transportation Sector BASE Scenario
1995 US million dollars/petajoules/annum
Years
T41 T47 T4E T4F T4X T4Y T9H
Database: C:\ANSWER\Answer_Databases\SMATDEM3.MDB
Edit Scenario: BASE

Scenario	Parameter		Technology	1990	2000	2010	2020
BASE	INVCOST	?	R66	0.0000	120.0000	110.0000	100.00
BASE	INVCOST	?	R67	52.0000	52.0000	52.0000	52.00
BASE	INVCOST	?	R69	35.0000	35.0000	35.0000	35.00
BASE	INVCOST	?	R6A	105.0000	105.0000	105.0000	105.00
BASE	INVCOST	?	R85	80.0000	80.0000	80.0000	80.00
BASE	INVCOST	?	R89	65.0000	65.0000	65.0000	65.00
BASE	INVCOST	?	S22	1.0700	1.0700	1.0700	1.07
BASE	INVCOST	?	S2A	1.9700	1.9700	1.9700	1.97
BASE	INVCOST	?	S80	6.6800	6.6800	6.6800	6.68
BASE	INVCOST	?	SF1	0.0000	1,300.0000	1,300.0000	1,300.00
BASE	INVCOST	?	SF2	0.0000	1,500.0000	1,500.0000	1,500.00
BASE	INVCOST	?	SF3	0.0000	1,700.0000	1,700.0000	1,700.00
BASE	INVCOST	?	T41	201.3000	199.0000	196.7000	194.40
BASE	INVCOST	?	T47	120.0000	120.0000	120.0000	120.00
BASE	INVCOST	?	T4E	0.0000	229.0000	225.0000	225.00
BASE	INVCOST	?	T4F	83.1500	83.1500	83.1500	83.15
BASE	INVCOST	?	T4X	60.0000	60.0000	60.0000	60.00
BASE	INVCOST	?	T4Y	49.3300	47.6000	47.6000	47.60
BASE	INVCOST	?	T9H	98.9100	98.9100	98.9100	98.91
Add BASE	INVCOST	?					

6. Conclusions

We have addressed some of the procedures necessary for implementing software quality assurance in mathematical modeling systems, showing how SQA for MP software differs from SQA activities for more traditional software systems. The key steps are automated testing and automated configuration management tools, which foster continual quality improvement. In particular, the focus of our testing activities is on reproducible full lifecycle testing. Our testing framework, which includes model collections, data collection tools, and data analysis tools, allows seamless full lifecycle testing inside the GAMS system, with many components available for outside use by non-GAMS customers and researchers in general. We hope that these tools will illustrate the importance of QA activities for academics, commercial users, solver vendors, as well as modeling system developers and hope they can be of benefit to the mathematical community in improving their software quality assurance methodologies.

While some of the procedures described may seem disconnected, the focus is on reaching the highest level of maturity defined by the CMM. Most MP vendors, like GAMS, operate in a very small niche market and do not have the resources to follow conventional QA processes used in large enterprises that have revenues thousands of times larger than the entire MP niche market. How can we evolve from Level 1, the exciting but chaotic phase, to Level 5, the mature phase, and become a credible partner for other industries?

The main thrust should be to: (1) Automate the QA process and certification, (2) build the tools into the software and share the QA process (3) make the QA process transparent and reproducible, and (4) involve the clients (academic and commercial) and make them part of the QA process.

References

[1] Brooke, A. Kendrick, D., and Meeraus,A. (1988). *GAMS: A User's Guide*, San Francisco, CA: The Scientific Press.

[2] American Society for Quality. (2004). Online at http://www.asq.org/.

[3] Bussieck, M.R., Drud, A.S., Meeraus, A. and Pruessner, A. (2002). Quality Assurance and Global Optimization. C. Bliek, C. Jermann, A. Neumaier, eds. *Global Optimization and Constraint Satisfaction, First International Workshop on Global Constraint Optimization and Constraint Satisfaction, COCOS 2002*, LNCS2861. Springer Verlag, Heidelberg Berlin, (223-238).

[4] Dolan, E.D. and Moré, J.J. (2002). Benchmarking optimization software with performance profiles, *Math. Programming*, 91 (2) (201-213).

[5] Dolan, E.D. and Moré, J.J. (2000). Benchmarking optimization software with COPS, *Technical Report ANL/MCS-TM-246*, Argonne National Laboratory, Argonne, Illinois.

[6] Drud, A.S. (2002). Testing and Tuning a New Solver Version Using Performance Tests, INFORMS San Jose, *Session on "Benchmarking & Performance Testing of Optimization Software."*. See http://www.gams.com/presentations/present_performance.pdf.

[7] Fourer, R. and Gay, D.M. (1993). *AMPL: A Modeling Language for Mathematical Programming*, Redwood City: The Scientific Press.

[8] GAMS Development Corporation. (2004). *GAMS - The Solver Manuals*. GAMS Development Corporation, Washington, DC: http://www.gams.com/solvers/allsolvers.pdf.

[9] Hamilton, L.D., Goldstein, G.A. et al. (1992). MARCAL-MACRO: An Overview, Biomedical and Environmental Assessment Group, *Technical Report BNL-48377*. Analytical Sciences Division, Department of Applied Science, Brookhaven National Laboratory, Associated Universities.

[10] International Organization for Standardization. (2004). Online at http://www.iso.org.

[11] Mittelmann, H.D. (2003). An Independent Benchmarking of SDP and SOCP solvers. *Mathematical Programming*. 95, (407-430).

[12] Moore, J.W. (1998). *Software Engineering Standards: A User's Road Map*. IEEE Computer Society, Los Alamitos, CA.

[13] PAVER Server (2004). Online at `http://www.gamsworld.org/performance/paver`.

[14] Pressman, R.S. (1997). *Software Engineering: A Practitioner's Approach*, 4th Edition, Boston, MA: McGraw-Hill.

[15] Schrage, L.S. (1991). *Lindo - An Optimization Modeling System*, Scientific Press series, Fourth ed., Danvers, MA: Boyd and Fraser.

[16] Software Engineering Institute. (1994). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, MA: Addison-Wesley.

[17] Shcherbina, O. Neumaier, A. Sam-Haroud, D. Vu, X.-H. and Nguyen, T.V. (2003). Benchmarking Global Optimization and Constraint Satisfaction Codes. C. Bliek, C. Jermann, A. Neumaier, eds. *Global Optimization and Constraint Satisfaction, First International Workshop on Global Constraint Optimization and Constraint Satisfaction, COCOS 2002, LNCS2861*. Heidelberg Berlin: Springer Verlag. (223-238).