**Using Software Assurance and Algorithm Testing to Verify   
a One-dimensional Transport Model**

E. Ateljevich1Ph.D., P.E., K. Zamani2, F. A. Bombardelli2 Ph.D., and J. Anderson1Ph.D., P.E.

1Bay-Delta Office, Delta Modeling Section, California Department of Water Resources (DWR), 1416 Ninth Street Sacramento, CA,95814; PH (916) 645-5111; FAX (916) 653-6077; e-mail: eli@water.ca.gov, jamiea@water.ca.gov.

2Department of Civil and Environmental Engineering, University of California, Davis, 2001 Ghausi Hall, Davis, CA 95616; PH (530) 752-0949; FAX (530) 752-7872; email: [kzamani@ucdavis.edu](mailto:kzamani@ucdavis.edu), [fabombardelli@ucdavis.edu](mailto:fabombardelli@ucdavis.edu).

**ABSTRACT**

In this paper we describe a framework for software verification of a transport model. The framework is crafted according to principles from both the software testing (sometimes known as Software Quality Engineering) and numerical testing fields. Herein, we describe the components and implementation of the test suite, which includes unit tests, regression tests and algorithm tests of convergence and accuracy. Our tests are applied to a transport model including advection, dispersion and reaction (source) components. We make use of analytical solutions obtained from the literature and from the Method of Manufactured Solutions. We discuss the layered complexity of our algorithm tests, and the similarities and tension between the silent, automatic perspective of software testing and the verbose, graphical output required for public reporting of numerical verification and validation results. Applying this rigourous testing framework to our transport code provides assurances to the developers and end users that the code performs as expected.The testing framework described in this paper provides a useful starting point and some practical examples for researchers and practitioners wanting to verify transport codes.

**INTRODUCTION**

In this paper, we describe our approach and experiences developing a software verification framework for a one dimensional (1D) transport model of advection, dispersion and reactions or sources (ADR). We begin by describing the motivation and requirements for testing. Our acceptance criteria are driven by the requirements for the model, but are crafted according to principles from both the software and numerical testing fields. We then describe the components and implementation of the test suite, emphasizing the incremental nature of the tests, quantitative criteria for testing, and the similarities and tension between the silent, automatic perspective of software testing and the verbose, graphical outputs required for public reporting of numerical verification results. The testing framework described in this paper was developed as part of a project to create a new transport module for the Delta Simulation Model 2 (DSM2) (reference???), a 1-D hydrodynamic and transport model for flow and water quality in the Sacramento-San Joaquin Delta. Our target problems include river and estuary advection, and 1D approximations of common mixing mechanisms and source terms associated with conservative and non-conservative water quality kinetics including sediment transport. Since the transport module was created from scratch, it provided an opportunity to structure the code to be rigorously tested using principles of software quality engineering. The code was developed in a modular fashion so that each module performed a single task or a group of related tasks. A unit test was then created to verify that each module was performing correctly. In addition to the software tests, a suite of system tests was developed to assure that the code properly represents the key components of advection, dispersion and reaction. The transport code is described briefly below followed by the development of the testing framework.

**1D TRANSPORT MODEL**

The model used to illustrate the testing framework is based on the 1D transport equations in conservative form:



Time evolution

Source/Reaction

Advection

Dispersion

where *x* is the distance, *t* is time, *A* is the wetted area, *C* is the scalar concentration, *u* is the flow velocity, *K* is the longitudinal dispersion coefficient, and *R* is the source term (deposition, erosion, lateral inflow and other forms of sources and sinks). Equation (1) describes the mass conservation of a pollutant in dissolved phase, or suspended sediment away from the streambed.

The problem domain includes estuaries and river channels and even some small open water areas roughly approximated as channels. The main transport process is advection, and the mixing mechanisms we anticipate are turbulent diffusion, gravitational circulation, and shear dispersion. We anticipate the shear dispersion to dominate over the turbulent diffusion. We also expect the gravitational circulation to exert an important role in mixing. We additionally contemplate significant, non-linear source terms, though none of the above processes are so quickly varying as to constitute truly stiff reactions.

Our algorithms include an explicit scheme for advection based on the finite-volumes method (FVM) and the Lax, two-step method with van Leer flux limiter; it also includes an implicit, time-centered Crank-Nicolson scheme for dispersion. The advection and reaction solver are coupled as a predictor corrector pair, and dispersion is implemented using operator splitting. Two features of the algorithm are particularly important. First, the scheme requires a flow field (flow discharges and flow areas) that preserves mass continuity. In some cases, tests from the literature were written in non-conservative or primitive form and had to be reworked in conservative form. Second, we employ operator splitting and wanted to exercise the equations with and without known vulnerabilities (such as time-varying boundaries and nonlinear source terms) of this class of algorithm.

**TESTING REQUIREMENTS**

testing requirements and the

The target accuracy is strict second order for individual operators and near second order for the algorithm as a whole. Second order allows coarser discretization for a modest increase in work. A second-order algorithm gives us a buffer of accuracy as details like networks of channels and coarse boundary data are added. At the time of writing this paper, our splitting is first order Godunov splitting. Numerous authors (e.g. Leveque 1986) have observed that near second-order accuracy can be achieved with first order splitting, and the design of the tests probes this point.

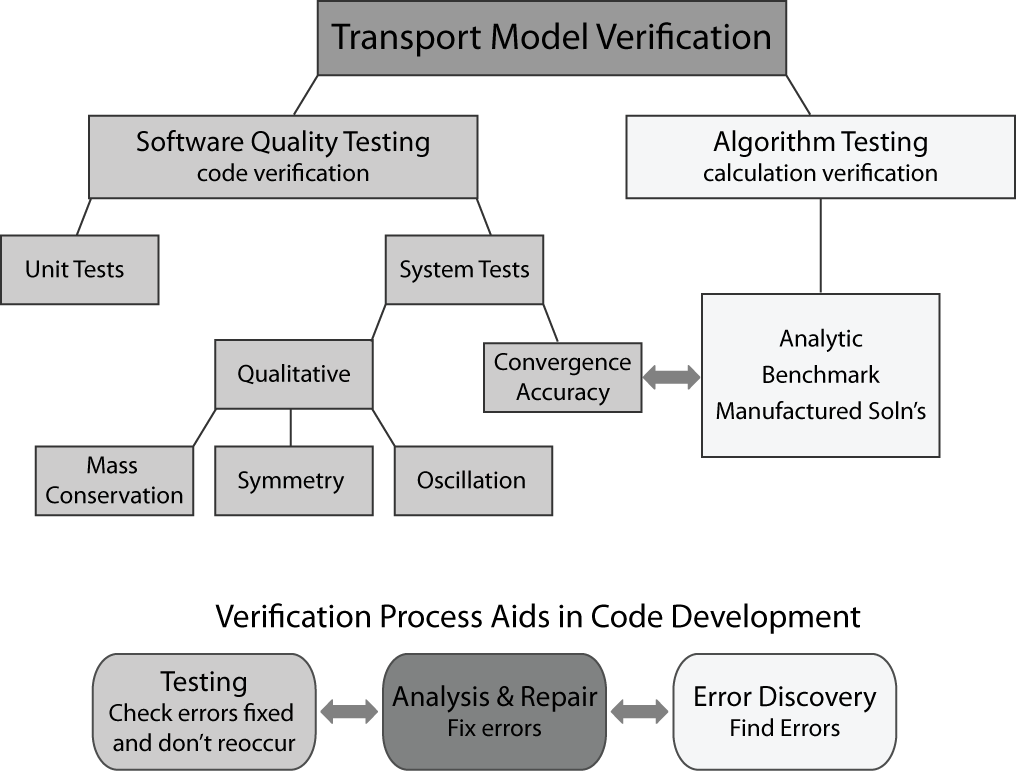
The main criteria used for testing the 1-D transport code include:

* Second order accuracy of the numerical scheme
* ?????

**TESTING PRINCIPLES**

Flow and transport codes inherently comprise both numerical algorithms and pieces of software. Well-developed testing literature exists for both. Oberkampf and Trucano (2002) describe some elements of software quality engineering (SQE) in the context of numerical verification, and note some cultural reasons why it is seldom implemented.

Figure 1 is adapted from this work and depicts the relationship between software testing components and algorithmic testing such as convergence tests. We regard numerical verification as our key responsibility and the numerical verification toolset as our greatest assets. Nonetheless, we also comment below on how these tools feature as tests and how, at times they seem in tension with the principles of good software testing.

****

**Figure 1. Schematic showing the relationship between software testing components and algorithmic testing.**

Adapted from Oberkampf and Trucano (2002)

**Software testing principles.** The principles that we want to emphasize are:

1. Testing should be automatic and continuous.
2. The approach should foster exact specification of every unit of code.
3. Testing should provide assurance of whether a set of specifications is met.

One goal of tests is that they be a continuous assessment of the code. The entire testing system is a *regression* suite that establishes a gauntlet through which future code changes must be passed. A consequence of automation is that tests must be phrased in terms of binary *assertions,* true and false statements that can be tested without human intervention and that reveal whether the aspect of the code under consideration is correct. Convergence criteria are a rigorous basis for assertions, either by requiring strict convergence criteria (“the algorithm is -second order accurate in time and space”) or a regression criterion (“convergence will not get any worse than last time the code was tested”).

The software testing literature further distinguishes between *unit tests* of atomic routines and *system tests* of larger subtasks. For example, the evaluation of a gradient might be a unit of code. Convergence tests can be thought of as system tests.

The unit testing point of view is that code must be exercised over a range of inputs that covers every line. For instance, to test a gradient routine with a slope limiter, a developer would want to cover:

1. well-behaved cases in the middle of the mesh.
2. behavior near the edges of the mesh, where one-sided differences may be used instead of central differences.
3. cases that test the limiters with steep or zero gradients in both directions.

Any system test will certainly exercise the central cases, which in any event can seldom be wrong without being obvious. However, a system test might miss errors in special cases. For example, a system test may miss a bug in the limiter for the case of steep decreasing slopes for several reasons. First, convergence is often assessed with limiters turned off, as they are locally order reducing. Second, it is hard to fiddle with the problem in just the right way to make sure the left, right, and center cases of the gradient limiter are all triggered. This is particularly true when trying to exercise all the other units of code the same way – parameter changes made to fully exercise one unit of code may lessen the coverage of another unit.

Although the software and algorithm tests are separate, information learned from one test can aid in the further development of another test. We began our coding with near-100% coverage by unit tests. These tests were part of the debugging and development processes. Later, discoveries made in the context of system tests were analyzed and pushed back into unit tests whenever possible. The unit test was expanded to verify that the newly discovered error from the algorithm test was fixed and does not reoccur.This flow of information is indicated in Figure 1.

**Numerical verification and algorithmic testing.** Algorithm tests such as convergence tests serve multiple purposes. They are intended in part to discover bugs (*system test*) and in part to convince ourselves and others of the merit of the algorithm to solve the problems to which it is directed (*acceptance test*).

One of the well-recognized and the standard verification methods of computational-fluid-dynamics codes is based on the notion of mesh convergence. Mesh convergence for models that solve partial differential equations is assessed by successively refining the spatial and temporal discretization. As the mesh converges, the error estimates (usually an Ln norm such as the sum of the absolute error or the sum of the squared error) should decrease at a rate that is usually called the *order of convergence*. By checking convergence, we ensure that the model is consistent with an underlying formulation rather than numerical artifacts. Failure to converge usually represents either a bug in the implementation or a difficulty of the algorithm on a class of problem. The transport code in this paper aims for second order accuracy.

The verification toolkit is largely targeted at providing test problems and methods to estimate error in situations where an analytical solution is not available from the literature. When nonlinearity, spatially varying coefficients and other complexities are introduced, tricks must be introduced to obtain good test problems.

Depending on the context, error and convergence are usually estimated one of two ways:

* When successive refinements are assessed relative to an analytical solution, we have a direct estimate of error and the ratio allows us to estimate a *convergence rate.*
* When successive grids are compared to one another, we can invoke the concept of Richardson extrapolation and Grid Convergence Index (Roache, 2009) to indirectly estimate error and convergence even when no solution is available.

In practice, we feel that use of the Method of Manufactured Solutions (MMS, Wang et al., 2009) is able to supply analytical verification problems for most cases of interest to us. Direct application of Richardson extrapolation is useful for *in situ* analyses of problems that cannot be altered, but we make no use of it here.

At least in theory, convergence rates can be stipulated as a project requirement and software testing assertion. Convergence rates, not absolute error, are what numerical methods tend to promise and they are very useful in the discovery of code defects. Still, the main goal in practice is a more accurate solver. Therefore, the superiority of methods should be assessed based on both convergence and accuracy.

The convergence ratio in a very coarse grid oscillates around its main value; as the grid size is refined, convergence becomes monotonic until the mesh size reaches a point where the machine precision overtakes the truncation error of the numerical scheme. At this point error norms do not change, and the convergence rate is zero. Convergence ratios should be checked for intermediate grid sizes, preferably at the scale of the real phenomenon and discretization used in practice. In the conclusions, we describe the challenge of dealing with tests that returned failed results when the convergence was just slightly below the target level.

As acceptance tests, algorithm tests should be conducted over a range of problems that exercise the major physical features that are to be modeled. The community may help with this by providing benchmarks, but we were unable to ascertain that any widely accepted benchmarks for a 1D transport code. As system tests we believe that the tests should be *glass box*, targeting known or discovered vulnerabilities of the algorithm. The layering of the advection, dispersion and reaction components of our convergence tests, for instance, is specifically motivated by our desire to use operator splitting.

Finally, distinction might be made between the reportable set of algorithmic tests and system tests aimed at defect discovery. Important examples are tests of symmetry, such as whether a 1D model gives the same result when the upstream and downstream result is swapped. Others are positivity preservation of constituents, mass conservation and oscillation detection. In the case of positivity preservation and mass conservation, it is typical to abstract this code for use both in the test suite and in the driver as a user option.

Overall, we agree with the conclusions of Overkampf and Trucano (2002) that system tests – particularly convergence tests – expose bugs well, particularly when an attempt is made to test symmetrically and over special cases. We feel that the hierarchical approach we describe in the next section further helps to isolate problems. Nevertheless, a close reading of Overkampf and Trucano (2002) does reveal that the convergence tests sometimes initially failed to pick up bugs that are exactly the sorts unit tests might catch (e.g. gaffes in corner cells).

**ALGORITHM TEST SUITE DESCRIPTION**

The algorithm testing used a hierarchical building block approach that tests key processes individually and then tests the processes in combinations of increasing complexity (Figure 2). For the example presented here, the tests were developed for a one-dimensional transport code that will be applied to an estuary.  Thus the key processes tested are the operators of advection, dispersion and reaction (e.g. growth or decay).  For each combination of operators presented in Figure 2, we start simple, using quiescent or steady-uniform flow and sequentially add complexity (different dimensions of the analysis) in the following ways:

* We developed tests in which a certain mass distribution is transported forwards, backwards, and back and forth; those tested cases included constant flow velocity and time-varying flow velocity for tidal conditions These tests indicate diverse levels of complexity of the flow field.
* In some tests, we represented spatially uniform coefficients, whereas we used variable coefficients in others. This is a crucial issue in practical cases such as flow and transport in an estuary. For instance, we employed constant dispersion coefficients in Tests 1 to 4, and a variable dispersion coefficient in Tests 12, 13 and 18.
* We also considered different boundary conditions. We devised tests in which the boundaries were located far enough so we eliminated their influence on the numerical results, and we selected tests in which the boundaries have a large role on the time evolution of the distribution of mass
* Finally, the last dimension of the analysis included non-linear source (reaction) terms. Thus, , we used a MMS to address the case of an ADR solver subjected to a non-linear reaction.

These tests were conducted for a range of parameter values. Typically the Courant number (a measure of numerical stability of the algorithm), domain length, and dispersion and decay coefficients were fixed, and the grid spacing and time steps were adjusted to maintain the same Courant number. Detailed descriptions of the tests are beyond the scope of this paper and will appear in a planned journal article.

Our hierarchical suite can identify with good precision exactly which added layer of complexity causes a drop in order of accuracy.

**Similarities and Tensions between Software and Algorithm Testing**

**CONCLUSIONS AND LESSONS LEARNED**

Our test suite succeeds both in finding bugs and in elucidating the strengths and weaknesses of a 1-D transport algorithm on various types of problems. We feel that our test suite is comprehensive for a class of problems that lack benchmarks. We have been able to establish second order convergence on a wide variety of problems despite operator splitting and excellent accuracy on even some of the problems where we miss convergence goals – such as problems with active time-varying boundaries or nonlinearity.

The key issues we have had to sort through are establishment of meaningful test assertions and the tension between test criteria and the need to demonstrate quality. These difficulties mostly lay in the algorithm tests, because the unit test component of the suite does not have a dual job description. A challenge with unit tests seems to be culture: generating the will to write them and the skills to write them in a way that covers the unusual cases. There are automatic tools to help discover whether this type of coverage is complete.

When it comes to algorithm tests, nominally we sought a second order convergence rate for all components of our algorithm. A convergence criterion seemed in-keeping with the way numerical model accuracy is expressed and less arbitrary than a hard-wired, scale-dependent absolute standard. Early on, however, it was clear that noisy convergence could spoil even a success when expressed as a hard assertion. It is challenging to deal with situations when a convergence test is failed by a value close to the criteria, e.g. 1.97 instead of 2.0, which surely would have passed a graphical acceptance test. We were also occasionally thwarted by sensitivity to problem parameters.

Given a stream of messages, we generally either fixed the code or we searched for bugs until both of the following things happened:

* Convergence properties corresponded well to the expected strengths and limitations of our algorithm; and
* The solution was accurate – qualitatively excellent when compared graphically to solutions and with relative errors of a hundredth of a percent.

We have done our best to support our claims on the first condition. Our hierarchical suite can identify with good precision exactly which added layer of complexity causes a drop in order of accuracy. The second condition reality check in our requirements for certain types of complex problems. We are in the process of changing our criteria in some cases to an absolute accuracy requirement paired with a regression standard for convergence (“don’t do worse than last time”).

* Codes must be written in a modular format with software testing in mind in order to apply the principals of software quality engineering
* Each piece of code must have a clear purpose and criteria for success
* There is a symbiotic realationship between software and algorithm tests; Code bugs detected with algorithm tests can lead to development of additional software tests to veryify that the bug is fixed and to provide assurance that it does not reoccur
* Symmetry tests are important to help discover errors that may be hidden by self correction of a symmetrical problem set up, such as tidal advection
* Test results are quantified relative to specified test critiera and provide more information than simply assessing graphs of expected vs computed results

.The software quality and algorithm testing framework described in this paper provides a useful starting point and some practical examples for researchers and practitioners wanting to verify transport codes. Having this rigourous test suite, allows developers to verify that (1) verify that each piece of code works properly both individually and as a whole, (2) additions to the code do not adversely affect existing code, and (3) to find and fix code bugs that might otherwise be missed. Providing the end user with test results and the ability to rerun the tests themselves, assures the user that the code performs as expected and quantifies the code’s strengths and weeknesses.

**ACKNOWLEDGMENTS**

This work has been possible due to the support given by the California Dept. of Water Resources (DWR) through a contract with the UC Davis. We thank Tara Smith and Dr. Francis Chung from DWR for their support of this project.

**REFERENCES**

Abbott, M. B., Price, W. A. (Eds.), (1994), “Coastal, Estuarial, and Harbour Engineers’ Reference Book”, Chapman & Hall.

Colella, P., Puckett, E. G., (1998), *“Modern Numerical Methods for Fluid Flow”,* Class Note.

Ferziger, J. H., Peric, M., (2002), “*Computational Methods for Fluid Dynamics”,* Springer.

Fischer, B. H., List, J. E., Koh, R. C., Imberger, J., Brooks, N. H., (1979),  *“Mixing in Inland and Coastal Waters”*, Academic Press, Inc.

Fletcher, C. A. J., (1991), *“Computational Techniques for Fluid Dynamics”*, Springer.

Leveque, J.R., (1986), *“Intermediate Boundary Conditions for Time-Split Methods Applied to Hyperbolic Partial Differential Equations”*, Mathematics of Computation, Vol. 47 (175), pp. 37-54.

Leveque, J. R., (2002), “Finite Volume Methods for Hyperbolic Problems”, Cambridge University Press.

Oberkampf, W. L., Trucano T. G., (2002), *“Verification and Validation in Computational Fluid Dynamics”*, SANDIA REPORT, No. SAND2002-0529.

Roache, P. J., (2009), *“Fundamentals of Verification and Validation”*, Hermosa Publishers.

Roache, P. J., Knupp, P. M., (1993), *“Complete Richardson Extrapolation”*, Communication in Numerical Method in Eng. Vol. 9, pp. 365-374.

Saltzman, J., (1994), *“An Unsplit 3D Upwind Method for Hyperbolic Conservation Laws”*, J. of Computational Physics, Vol. 115, pp. 153-168.

Wang, S. Y., Roache. P. J., Schmalz, R. A., Jia, Y., Smith, P. E., (2009), “Verification and Validation of 3D Free-Surface Flow Models”, ASCE, EWRI.

Zoppou, C., Knight, J. H., (1997), “*Analytical Solution for Advection and Advection-Diffusion Equation with Spatially Variable Coefficients*”, J. Hydraulic Eng., Vol. 123(2), pp. 144-148.