

# Mutation Analysis to Verify Feature Matrices for Isolating Errors in Simulation Models

Simon Carter<sup>1,2</sup>, Malcolm Graham<sup>1</sup>, Paul Strooper<sup>2</sup>, and Zhiguo Yuan<sup>1</sup>

<sup>1</sup>Advanced Wastewater Management Centre

<sup>2</sup>School of Information Technology and Electrical Engineering  
The University of Queensland, Queensland 4072

s.carter@awmc.uq.edu.au

malcolmg@awmc.uq.edu.au

pstroop@itee.uq.edu.au

zhiguo@awmc.uq.edu.au

## Abstract

Software simulation models are computer programs that need to be verified and debugged like any other software. In previous work, a method for error isolation in simulation models has been proposed. The method relies on a set of *feature matrices* that can be used to determine which part of the model implementation is responsible for deviations in the output of the model. Currently these feature matrices have to be generated by hand from the model implementation, which is a tedious and error-prone task. In this paper, a method based on mutation analysis, as well as prototype tool support for the verification of the manually generated feature matrices is presented. The application of the method and tool to a model for wastewater treatment shows that the feature matrices can be verified effectively using a minimal number of mutants.

*Keywords:* simulation, modelling, debugging, error isolation, mutation analysis, feature matrices.

## 1 Introduction

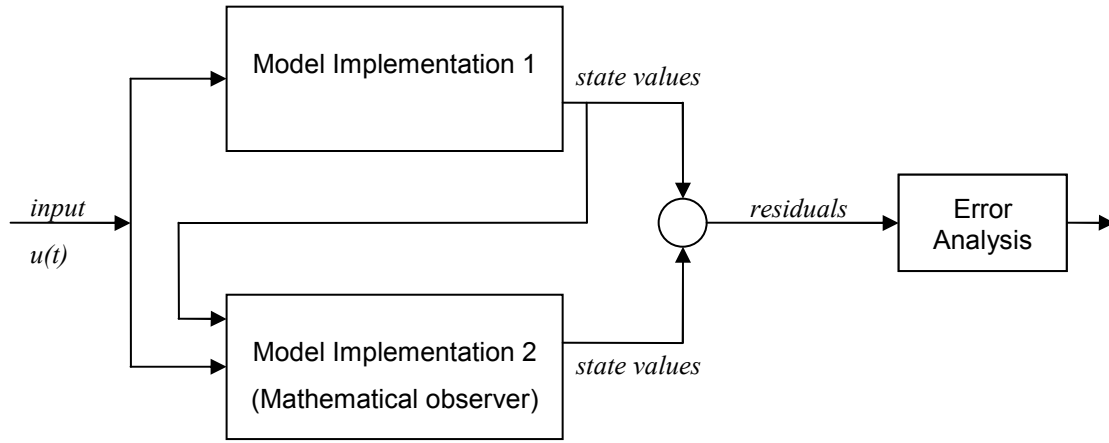
Modelling and simulation software is used in numerous applications. These models are computer programs that need to be verified and debugged like all other software. In previous work, Yuan et al. (2002) and Lennox and Yuan (2002) have developed the FEature MATrix based Debugging (FEAMAD) method for error isolation of simulation models. The method involves running two implementations of a model in parallel and analysing the differences in the outputs to locate the cause of these differences in the source code of the models. Although the method so far has only been applied to wastewater treatment models (Lennox et al. 2001, Graham et al. 2002), it can be applied to any models involving first-order ordinary differential equations.

One crucial step in the method is to generate a set of *feature matrices* that capture the essential parts of a model and that can be used to determine which part of the code is responsible for certain deviations in the output of

the model. Currently these feature matrices are generated by hand, which is a tedious and error-prone activity. An erroneous set of feature matrices may result in complete failure of the isolation algorithm. The verification of feature matrices is therefore a crucial step in the isolation method. In this paper, we present a method and prototype tool support based on *mutation analysis* to verify manually generated feature matrices.

Mutation analysis (DeMillo et al. 1978) or software fault injection (Voas and McGraw 1998) has traditionally been used to evaluate the adequacy of test sets. The idea is to generate faulty versions (mutants) of a piece of software by changing one or more parts in the software. By then running the original version of the software and the mutant on the same set of test cases, it is anticipated that at least one of the test cases in the test set should be able to detect a difference in the output between the two programs. If not, it could indicate a weakness in the test set (or that the mutant is functionally equivalent to the original code, which is a problem that must be addressed when using mutation analysis). Mutation analysis relies on the *coupling effect*, which is the assumption that tests that isolate simple errors will typically also expose more complex errors, and the *competent programmer hypothesis*, which states that programmers create nearly correct programs, implying that tests that catch simple mutation errors are also likely to reveal any real errors introduced by the programmer.

In this study, we generate mutants of a simulation model to support the verification of a set of feature matrices that were generated by hand. In particular, we generate a number of mutants of the original model and then run the error-isolation software on the original model and the mutant. If the feature matrices are correct, then the error-isolation software should be able to correctly identify how the original model was changed to create the mutant. If it cannot, then this indicates a fault in the feature matrices. We have implemented a prototype tool to mutate MATLAB implementations of computer models and have evaluated the method and the tool on the Activated Sludge Model No. 1 (ASM1) wastewater treatment model (Henze et al. 1987). The experiment indicates that the method works well and that a minimal number of mutants is sufficient to obtain confidence in the correctness of this particular set of feature matrices.



**Figure 1** Generation and analysis of error residuals for coding error isolation

## 2 The FEAMAD Method

### 2.1 FEAMAD Overview

Simulation coding error isolation is a specific type of software verification problem. Much of the coding effort, and hence many of the errors, involve the code directly implementing the mathematical equations of the model. It is therefore possible to use mathematical properties of the model to link the ‘causes’ (coding errors) to the ‘symptoms’ (unexpected behaviours of the model). The FEAMAD method isolates coding errors by making use of these mathematical properties (Yuan et al. 2002, Lennox and Yuan 2002).

The FEAMAD method is built upon the concept of back-to-back testing (Figure 1). Also known as comparison testing, back-to-back testing involves independent implementation of mathematical models into multiple versions (two in the FEAMAD method) by separate software engineering teams using the same specification. In testing, the test cases designed are applied to each version of the software. If the outputs from every version are the same, it is assumed that every implementation is correct. If the outputs are different, each of the implementations are investigated to determine if a defect in one or more of the versions is responsible for the difference. With the FEAMAD method, the back-to-back testing technique is extended from the simple cross verification of different implementations to the debugging of each implementation. This is done by converting one of the implementations (Model Implementation 2 in Figure 1) into a mathematical observer, which receives not only the normal inputs of the model, but also the state variables of the other implementation (Model Implementation 1 in Figure 1) as its inputs. This is done so that the residuals thus generated carry an easily identifiable feature for each class of coding errors, the identification of which, through residual analysis, results in the isolation of the corresponding errors.

The FEAMAD method relies on classifying the possible coding errors according to their places in a model structure and generating a *feature matrix* for each class of errors. A feature matrix is a signature of the

corresponding error and takes the form of a constant vector or a constant matrix. A set of feature matrices:

$$M = \{F_i \mid i = 1, 2, \dots, N\}$$

is thus obtained *a priori* for a given model, where  $F_i$  is the feature matrix of the  $i$ th class of coding error and  $N$  is the number of error classes. The residual generation algorithm described above is designed such that, should an error belonging to the  $i$ th class be present in the coded model, the residuals depend on the  $i$ th feature matrix in the following way:

$$residuals = F_i d_i(t) \quad (1)$$

where  $d_i(t)$  is a time-varying vector, which is typically unknown, generally reflecting the magnitude of the corresponding coding error. The residuals will reside in the subspace spanned by  $F_i$ . In other words, the  $i$ th class of errors should have occurred if the residuals are found to reside in this subspace by the residual analysis algorithm. If multiple errors are present in the code, it can be shown that residuals reside in the subspace spanned by all the feature matrices involved (Yuan et al. 2002). Therefore, multiple errors can also be identified.

To provide a better understanding of the feature matrix concept, the generation of feature matrices for a simple example model is provided below.

### 2.2 Example Feature Matrix Generation

Consider the following model consisting of a set of three ordinary differential equations (ODEs), which describe changes in three state variables ( $x_1, x_2, x_3$ ), two pre-calculations for reaction rates ( $r_1, r_2$ ), and one parameter ( $\alpha$ ).

*State derivative equations:*

$$\frac{dx_1}{dt} = r_1 - r_2 \quad (2)$$

$$\frac{dx_2}{dt} = \frac{1}{\alpha} r_1 \quad (3)$$

$$\frac{dx_3}{dt} = \frac{(1-\alpha)}{\alpha} r_1 - r_2 \quad (4)$$

Pre-calculations for reaction rates:

$$r_1 = \frac{x_2}{1+x_2} x_1 \quad (5)$$

$$r_2 = \frac{x_1}{1+x_1} \quad (6)$$

Isolatable errors in an ODE model such as this belong to one of three classes:

- (i) errors that affect a single state equation,
- (ii) errors that affect a pre-calculation (in this case a reaction rate equation) that may appear in more than one state equation, or
- (iii) errors in parameters that may appear in more than one state equation.

The feature matrices for each of these classes of errors are discussed in turn below.

#### (i) Feature matrices for state equation errors:

The form of the feature matrices for state equations (i.e. for equations (2)-(4) in the example model described above) is simple. An error in the  $i$ th state equation affects only the  $i$ th derivative, and hence its feature matrix is the  $i$ th column of the identity matrix  $I_{m \times m}$ , where  $m$  is the number of state variables. Therefore, for the example model described above the feature matrices for the state equations are:

$$FM_{S1} = [1 \ 0 \ 0]^T$$

$$FM_{S2} = [0 \ 1 \ 0]^T$$

$$FM_{S3} = [0 \ 0 \ 1]^T$$

#### (ii) Feature matrices for pre-calculation errors:

In a similar manner as that for state equation feature matrices, deriving the feature matrices for pre-calculations is essentially a matter of determining how an error in a particular pre-calculation affects the state derivatives (and hence affects the error residuals). For the case of reaction rate 1 (equation 5) it can be seen that an error in this pre-calculation has an affect on the first derivative by a factor of 1, the second derivative by a factor of  $(1/\alpha)$ , and on the third derivative by a factor of  $(1-\alpha)/\alpha$ . Therefore the feature matrix for this pre-calculation is:

$$FM_{R1} = \left[ 1 \quad \frac{1}{\alpha} \quad \frac{(1-\alpha)}{\alpha} \right]^T$$

and similarly for the second pre-calculation (reaction rate 2):

$$FM_{R2} = [-1 \ 0 \ -1]^T$$

#### (iii) Feature matrices for parameter errors:

The feature matrix derivation for a parameter again follows the same methodology as for the states and pre-calculations; that is, determine how an error in the parameter affects each derivative. One minor difference in this example for generating the parameter feature matrix, as opposed to the pre-calculation and state feature matrices, is that the derivatives are not linear with respect to the parameter,  $\alpha$ . Therefore a feature matrix for the term  $(1/\alpha)$  is generated here instead. This is essentially still a feature matrix for  $\alpha$  because isolating an error in the term  $(1/\alpha)$  corresponds to an error in  $\alpha$ . The feature matrix for the parameter in this example is:

$$FM_{\alpha} = [0 \ 1 \ 1]^T$$

Note that the  $r_1$  term does not become a part of this feature matrix because it is a time-varying vector and forms part of the  $d_i(t)$  term as explained by equation (1).

Generating the feature matrices is not a difficult task for such a simple model, but as the model size increases this task becomes more and more prone to human error. Given that the feature matrices form the basis for error isolation it is crucial that they are formulated correctly. Currently, the feature matrices are generated manually and they must therefore be verified before being used for error isolation.

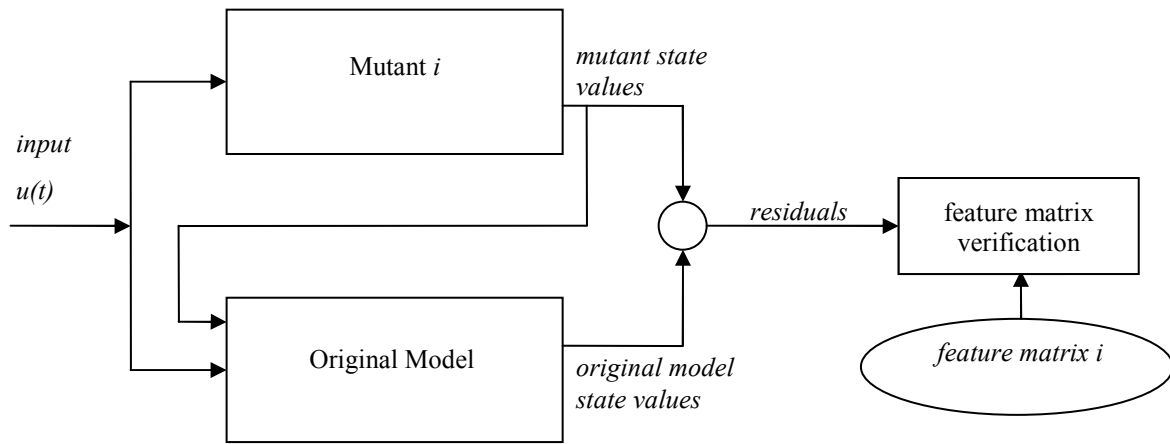
## 3 Mutation Analysis for Feature Matrix Verification

### 3.1 Methodology

Mutation analysis has traditionally been used to seek a measure of the coverage of a test suite by running the test suite over a suitably large number of mutants of a program. In our application we use the idea of program mutation to seek a measure of the correctness of the associated set of feature matrices rather than associated test data. If the set of feature matrices has been derived correctly for the MATLAB model under investigation, it should be able to identify errors in that model.

The proposed feature matrix verification (FMV) methodology is performed in three steps:

- (I) For the set of feature matrices  $F_i$ ,  $i = 1..N$ , generate a mutant for each class of possible coding errors. This will give a minimal set of  $N$  mutants. In some cases more than one mutant may be required to reduce the possibility that mutants are functionally equivalent to the original.
- (II) Generate error residuals for each of the  $N$  mutant models, against the reference model using the FEAMAD method, to obtain the  $N$  'mutant' residuals (Figure 2).
- (III) For each of the  $N$  residuals, use the FEAMAD method to determine if the feature matrix of



**Figure 2** Generation and analysis of error residuals for feature matrix verification

the  $i$ th class of coding error,  $F_i$ , can isolate the corresponding error in the  $i$ th mutant. If the seeded error in the  $i$ th mutant is *not* isolated by  $F_i$ , then  $F_i$  must be incorrect. Note that if the seeded error in the  $i$ th mutant *is* isolated by  $F_i$ , it remains uncertain if  $F_i$  is correct or not. In this case it cannot be assumed that  $F_i$  is correct, because an error in  $F_i$  may leave it functionally equivalent. An example is presented in Section 4.4. However the confidence we can place in the feature matrices that were able to isolate the corresponding errors is increased.

### 3.2 Mutant Generation

We have implemented a prototype tool consisting of eight modules (approximately 900 lines) written in C. The tool was built with the use of two files describing valid lexical tokens and a grammar for the MATLAB language found on the WWW<sup>1</sup>. The file containing lexical definitions was passed through the Flex<sup>2</sup> tool, which automatically produced a lexical scanner. Similarly the grammar file was input to the Bison<sup>3</sup> tool to generate a MATLAB parser. Code was added to the parser to build a syntax tree of the parsed program and a separate routine written to transform the syntax tree into a pretty-printed standard form for input to the MATLAB package.

To generate mutants of a MATLAB program we parse the program, traverse the syntax tree to find relevant statements, apply mutation rules to these statements, and finally output the mutant syntax tree in pretty-printed form. Care was taken to avoid generating mutants that are functionally equivalent to the original, as these cannot be distinguished.

## 4 Experimental Evaluation

### 4.1 ASM1 and Feature Matrices

To test the FMV methodology, the Activated Sludge Model No. 1 (ASM1) wastewater treatment model (Henze et al. 1987) was used. ASM1, which describes biological removal of organic carbon and nitrogen from wastewater by microorganisms, is one of the most widely utilised models in the field of wastewater treatment.

ASM1 is a small-scale model, consisting of 13 first-order ordinary differential equations, 8 biological reaction rates and 5 stoichiometric model parameters. Although implementing this model into MATLAB code is essentially a transcription process, there are numerous possibilities for the programmer to introduce errors, which are difficult to detect manually. For example, an incorrect value could be entered for a parameter, a wrong sign (e.g. + instead of -) could be entered in a process rate equation, or a term could be omitted in a state derivative equation. The possible errors were categorised into 26 classes (representing errors in the 13 differential equations, 8 reaction rates, and 5 parameters), and the feature matrix for each of these classes was established in Lennox and Yuan (2002).

### 4.2 Mutant Generation Rules

In this case study, 26 mutants of ASM1 were generated, each designed to verify one of the 26 feature matrices. The MATLAB implementation of ASM1 used in this study consists of three sections: the declaration of the model parameters, a block of code calculating the process reaction rates, and a set of state equations calculating the derivatives of the thirteen state variables.

The mutants were generated using the following rules:

- A mutant to a parameter is created by multiplying the parameter value by a constant, in this application 1.25.
- ASM1 has eight process reaction rates, consisting of arithmetic expressions involving the parameters and model state variables. We have implemented a number of mutation rules as follows: set a reaction rate to a constant; multiply a reaction rate by a constant; exchange one arithmetic operator for another; eliminate a state variable or substitute one for another; eliminate terms involving the parameters and state variables. For this application we have only used one of these rules, generating eight mutants by setting each reaction rate to zero. We expect to use the other mutants in future work.

- Similarly, ASM1 has thirteen state equations comprising arithmetical combinations of parameters, state variables, process reaction rates and model inputs. We have created thirteen simple mutants by setting each of these equations to zero. Again, as in the case of mutating the reaction rates, the tool can generate more mutants and we expect to use these in future work.

At present our mutation program can only cope with model implementations of a particular form as we exploit the structure and style of particular implementations. Further work would allow our mutation program to deal with other types of model structure.

### 4.3 Validation of the FMV method

The FMV method was validated in two ways. Firstly we tested the method using the known set of correct feature matrices to show that they contain no errors (i.e. to confirm that all of the errors in the mutants are correctly isolated). Secondly, to show that the method will detect incorrect feature matrices, we generated 26 erroneous sets of feature matrices by introducing known errors to one of the feature matrices in each set. By applying the FMV method to these erroneous sets of feature matrices we should be able to detect which feature matrix in the set is incorrect. Note that this second part of validation is not part of the FMV method itself, rather we are using erroneous sets of feature matrices to evaluate the effectiveness of the method.

To simulate a wide range of possible feature matrix errors, four types of errors were introduced, being either:

1. The movement of a term in the feature matrix up or down in the vector. For example, if a correct feature matrix term was  $[0,0,1,0,0,0]$  then the introduced error could be to change the vector to  $[0,1,0,0,0,0]$  or  $[0,0,0,0,1,0]$ , etc.;
2. An incorrect parameter name (NB: the terms of a feature matrix may contain constants *and* parameter values);
3. Incorrect signs (e.g. '+' instead of '-', or vice versa); or
4. The wrong numerical value for a constant.

### 4.4 Results and Discussion

Using the FMV methodology detailed in section 3.1, the residuals generated from each of the 26 mutants were analysed against the correct set of feature matrices and the 26 erroneous sets of feature matrices. In each analysis, only the  $i$ th feature matrix, corresponding to the residuals from the  $i$ th mutant, was supplied to the isolation algorithm. For example, consider the first erroneous set of feature matrices, into which an error has been introduced to the first feature matrix in the set of 26. For the residuals generated from the first mutant, the isolation algorithm is executed using only the first feature matrix in the set of 26 (which in this case happens to contain an error). The isolation result could either be the

number 1, meaning that a coding error of class 1 has been detected in mutant 1, or an 'X', meaning that no errors were detected. This process is then repeated for the other 25 feature matrices in the set, using the residuals from the other 25 mutants.

If the result is a number ( $i$ ) which corresponds to the  $i$ th mutant, this reveals that the feature matrix in question has correctly isolated the error in the corresponding mutant. If the result is 'X', it means the  $i$ th feature matrix has not isolated the error in the  $i$ th mutant and is concluded to be incorrect.

All of the erroneous feature matrix terms (i.e. the  $i$ th terms corresponding to the  $i$ th mutants), with the exception of case 4, were isolated using the proposed approach. That is, in every case except 4, an error introduced to a feature matrix resulted in its inability to isolate the corresponding error in the mutant, and therefore the corresponding feature matrix is exposed as an incorrect one.

In case 4, even though an error had been introduced to the fourth feature matrix in the set of 26, the error introduced did not change the subspace property of the correct feature matrix and therefore it was still able to isolate the corresponding error in the mutant. This issue stems from the general properties of feature matrices and cannot be resolved with the proposed methodology.

In a real FMV process the correct set of feature matrices are unknown. For completeness though we executed the FMV process using the correct set of feature matrices for ASM1, and given that the errors in each of the mutants were correctly isolated, this provided additional confirmation that the FMV methodology located no errors in the correct set of feature matrices.

## 5 Related Work

Due to the specific nature of the FEAMAD method there are few relevant references in the literature. We are aware of two other approaches of isolating errors in software using multiple implementations.

In Hildebrandt and Zeller (2000) and Zeller and Hildebrandt (2002), a prototype of the *delta debugging algorithm* is described. This algorithm simplifies verbose test results into a *minimal test case* by repeated automatic testing, comparing the current version of a program with a previous one. Zeller provides a short précis of the delta debugging approach in Zeller (2001), arguing that the task of debugging should be largely automated. Such a process should also give insight into the causes of uncovered bugs. Zeller further expands on this idea in Zeller (2002), developing the notion of *cause-effect chains* to automatically explain the causes of program failures.

The work by Abramson et al. (1996), on a technique called *relative debugging*, has similar objectives to coding error isolation. Abramson et al. (1996) address the tedious problem of verifying sequential versions of scientific models by automating the process of comparing a modified model (the new version) against a correct reference model (the old version). Although this

technique relies on automated run-time comparison of the internal states of the models as a pathway to isolating coding errors, the similarity to our isolation technique ends there. Once it is known that the model (new version) contains errors, the relative debugging technique relies on user intervention to repeatedly refine assertions and reinvoke the debugger. Our coding error isolation technique utilises knowledge of the mathematical properties of the model to automatically isolate errors.

The field of software fault injection is comprehensively covered in Voas and McGraw (1998) who provide an overview of the approach together with specific techniques such as software mutation for testability assessment and extended propagation analysis for assessing external failures in software-based systems. Techniques for software vulnerability measurement and applications in reuse and maintenance are also discussed.

## 6 Conclusions

This paper has presented a novel application of program mutation whereby simulation models are mutated for the verification of their associated feature matrices. The case study presented demonstrates that the technique is capable of detecting erroneous feature matrices. Only a minimal number of mutations are required, thus eliminating the need to deal with equivalent mutants. Although the method requires only a small number of mutants for a simple model such as ASM1, this number would be significantly larger for more complex models. For example the minimal number of mutants required for the more complex Activated Sludge Model No. 2d (ASM2d) of Henze et al. (1999) is 62.

The mutation tool prototyped in this study will be applied to the verification and determination of other parameters involved in the FEAMAD method. An immediate application, which is currently under study, is to experimentally determine the thresholds used by the FEAMAD method to determine if the two model implementations produce identical outputs and, if not, within which subspace the residuals are located.

## 7 References

ABRAMSON, D., I. FOSTER, J. MICHALAKES and R. SOSIC (1996): Relative debugging: A new methodology for debugging scientific applications. *Communications of the ACM* **39**(11): 67-77.

DEMILLO, R. A., R. J. LIPTON and F. G. SAYWARD (1978): Hints on test data selection: Help for the practicing programmer. *Computer* **11**(4): 34-41.

GRAHAM, M. L., J. A. LENNOX and Z. YUAN (2002): Automatic error isolation in wastewater simulation models: A real life cross-platform application. *CDROM Proceedings of Enviro 2002 & IWA 3rd World Water Congress*, Melbourne.

HENZE, M., C. GRADY, W. GUJER, G. MARAIS and T. MATSUO (1987): Activated sludge model no. 1. International Association on Water Pollution Research and Control. London.

HENZE, M., W. GUJER, T. MINO, T. MATSUO, M. C. WENTZEL, G. V. R. MARAIS and M. C. M. VAN LOOSDRECHT (1999): Activated sludge model no.2d, asm2d. *Water Science and Technology* **39**(1): 165-182.

HILDEBRANDT, R. and A. ZELLER (2000): Simplifying failure-inducing input. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, USA, 135-145, ACM Press.

LENNOX, J., Z. YUAN and J. HARMAND (2001): A systematic approach to error isolation in computerized wastewater simulation models. *Water Science and Technology* **43**(7): 367-376.

LENNOX, J. A. and Z. YUAN (2002): An approach to verifying and debugging simulation models governed by ordinary differential equations part II - residuals analysis and a case study. *Accepted by International Journal for Numerical Methods in Engineering*.

VOAS, J. M. and G. MCGRAW (1998): *Software fault injection: Inoculating programs against errors*, Wiley.

YUAN, Z., M. L. GRAHAM and J. A. LENNOX (2002): An approach to verifying and debugging simulation models governed by ordinary differential equations part I - methodology for residual generation. *Accepted by International Journal for Numerical Methods in Engineering*.

ZELLER, A. (2001): Automated debugging: Are we close? *IEEE Computer* **34**(11): 26-31.

ZELLER, A. (2002): Isolating cause-effect chains from computer programs. *Accepted by Proceedings 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, Charleston, South Carolina, ACM Press.

ZELLER, A. and R. HILDEBRANDT (2002): Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* **28**(2): 183-200.

**Acknowledgement:** This study is supported by The University of Queensland via an internal development project, the Australian Research Council, and Hemmis N.V. (Belgium) via Project LP0230324.

---

<sup>1</sup> <http://www.angelfire.com/ar/CompiladoresUCSE/COMPI LERS.html>

<sup>2</sup> <http://www.gnu.org/software/flex/flex.html>

<sup>3</sup> <http://www.gnu.org/software/bison/bison.html>