PROCEEDINGS

SEKE 2011

The 23rd International Conference on Software Engineering & Knowledge Engineering

Sponsored by

Knowledge Systems Institute Graduate School, USA

Technical Program

July 7-9, 2011

Eden Roc Renaissance Miami Beach, Florida, USA

Organized by

Knowledge Systems Institute Graduate School

Semantic Web Technologies

Software Testing and Debugging

Using Coverage and Reachability Testing to Improve Concurrent Program Testing Quality

Simone R. S. Souza[1], Paulo S. L. Souza[1], Mario C. C. Machado[1],
Mário S. Camillo[1], Adenilso Simão[1] and Ed Zaluska[2]
[1] Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo
P.O. 668 – São Carlos – Brasil – 13560-970
{srocio, pssouza,mmachado, adenilso}@icmc.usp.br, mariocamillo@gmail.com
[2] Eletronics and Computer Science – University of Southampton
ejz@ecs.soton.ac.uk

Abstract

The testing of concurrent software is a challenging task. A number of different research approaches have investigated adaptation of the techniques and the criteria defined for sequential programs. A major problem with the testing of concurrent software that persists is the high application cost due to the large number of the synchronizations that are required and that must be executed during testing. In this paper we propose a complementary approach, using reachability testing, to guide the selection of the tests of all synchronization events according to a specific coverage criterion. The key concept is to take advantage of both coverage criteria, which are used to select test cases and also to guide the execution of new synchronizations, and reachability testing, which is used to select suitable synchronization events to be executed. An experimental study has been conducted and the results indicate that it is always advantageous to use this combined approach for the testing of concurrent software.

1. Introduction

Concurrent applications are inevitably more complex than sequential ones and, in addition, all concurrent software contains features such as nondeterminism, synchronization and inter-process communication which significantly increase the difficulty of validation and testing.

For sequential programs, many testing problems were simplified with the introduction of testing criteria and the implementation of supporting tools. A testing criterion is a predicate to be satisfied by a set of test cases which can be used as a template for the generation of test data [10].

Extending previous work on sequential program structural testing criteria, we have proposed structural testing criteria for the validation of concurrent programs, applicable to both message-passing software [14] and shared-memory software [12]. These testing criteria are designed to exploit information about the control, data and communication flows of concurrent programs, considering both sequential and parallel aspects.

The use of these criteria significantly improves the quality of the test cases, providing a coverage measure that can be used in two important testing procedures. In the first one, the criteria can be used to guide the generation of test cases, where the criteria are used as guideline for test data selection. The second testing procedure is related to the evaluation of a test set; in this case, the criteria can be used to determine when the testing activity can be terminated based on sufficient coverage of the required elements. The main contribution of the proposed testing criteria is to provide an efficient coverage measure for evaluating the progress of the testing activity and the quality of test cases.

This approach uses static analysis of the program under test to extract relevant information for testing, which is straightforward to apply and generate relevant information for coverage testing. The problem is the large number of infeasible elements generated that must be analyzed. An element is infeasible if there is no set of values for the parameters (the input and global variables) that cover that element. Complete determination of infeasible elements is an extremely difficult problem and it is not possible to determine them automatically.

Lei and Carver [7] proposed a method (based on reachability testing) to obtain all of the executable synchronizations of a concurrent program (from a given execution of the program) in a way that reduces the number of redundant synchronizations. As this method uses dynamic information, only feasible synchronizations are generated, which

is a considerable advantage. However, a difficulty with this method is the high number of possible combinations of synchronization that are generated. For complex programs, this number is very high, which limits the practical application of this approach. In [1], Carver and Lei proposed a distributed reachability testing algorithm, allowing different test sequences be executed concurrently. This algorithm reduces the time to execute the synchronizations, but the authors do not comment about the effort necessary to analyze the results from these executions.

Lei and Carver's [7] method is essentially complementary to our approach. They do not address how to select the test case which will be used for the initial run, while we use the static analysis of the program to select the optimum test cases in advance.

In this paper we propose a complementary approach, using reachability testing to target coverage testing for synchronization events. The idea is to take appropriate advantage of both approaches: information about synchronizations provided by the coverage criterion are used to decide which race variants will be executed, selecting only synchronizations that have not already been covered by existing test cases. It is therefore possible to execute each synchronization at least once and to use reachability testing to select only those synchronizations that are feasible.

This paper is structured as follows. In Section 2 we describe related work on the testing of concurrent software, presenting more details on the coverage testing and reachability testing approaches. In Section 3 we present the test strategy proposed in this paper. In Section 4, an experimental study to evaluate our test strategy is presented and the results obtained are discussed. Finally, in Section 5 we present our conclusions together with future work.

2. Concurrent Program Testing

Traditional testing techniques are often not well-suited to the testing of concurrent or parallel software, in particular when nondeterminism and concurrency features are significant. Many researchers have developed specific testing techniques addressing such issues and in addition there have been initiatives to define suitable testing criteria [16, 18, 17, 8, 11, 15]. The detection of race conditions and mechanisms for replay testing have also been investigated [4, 7, 2, 3].

Yang [18] describes a number of challenges for the testing of parallel software: 1) developing static analysis; 2) detecting unintentional races and deadlock in nondeterministic programs; 3) forcing a path to be executed when nondeterminism might exist; 4) reproducing a test execution using the same input data; 5) generating the control flow graph for nondeterministic programs; 6) providing a testing framework as a theoretical base for applying sequential testing criteria to parallel programs; 7) investigating the applicability of sequential testing criteria to parallel program testing; and 8) defining test coverage criteria based on control and data flow.

Lei and Carver [7] proposed the *reachability testing* for generating all feasible synchronization sequences (and only them). This method guarantees that every partially-ordered synchronization will be exercised exactly once without repeating any sequences that have already been exercised. The method involves the execution of the program in a semi-deterministic way; the execution is deterministic up to a given point, from which it runs nondeterministically. The resulting synchronization sequence (sync-sequence), which is feasible, is analyzed and a new feasible sequence (if possible) is computed. The authors employ a reachability schema to calculate the synchronization sequence automatically. The reachability testing uses dynamic information to execute all feasible synchronization sequences, generating all *race variants* from one particular execution.

The reachability testing process is illustrated in Figure 1 (extracted from [7]). The figure shows a space-time diagram in which vertical lines represent four threads of a concurrent program. The interaction between processes is represented by arrows from a send event to a receive event. Diagram Q_0 shows the one execution of the program, generating the synchronizations: (s_1^{T1}, r_1^{T2}), (s_2^{T4}, r_2^{T2}), (s_3^{T2}, r_3^{T3}), (s_4^{T4}, r_4^{T3}). V_1, V_2 and V_3 are *race variants* of Q_0 and feasible executions generated during reachability testing execution. A problem here is the high number of possible combination of synchronization that are generated and (for complex software) this number can be very high, restricting any practical application of the strategy. This approach has the important advantage that it will generate only feasible synchronization sequences, which is an important consideration when reducing the cost of the testing activity.

2.1. Structural Testing for Concurrent Programs

In this section we describe our test model and criteria for validation of message-passing software [14]. The test model captures control, data and communication information. The model considers that a fixed and known number of processes n is created at the initialization of the concurrent application. These processes may each execute different programs. However, each one executes its own code in its own memory space. The concurrent program is defined by a set of n parallel processes $Prog = \{p^0, p^1, \ldots p^{n-1}\}$. Each process p has its own Control Flow Graph CFG^p, that is built using the same concepts as traditional software [10]. In other words, a CFG of a process p is composed of a set of nodes N^p and a set of edges E^p. Each node n in the process p is represented by the notation n^p and corresponds to a set of commands that are sequentially executed or can be

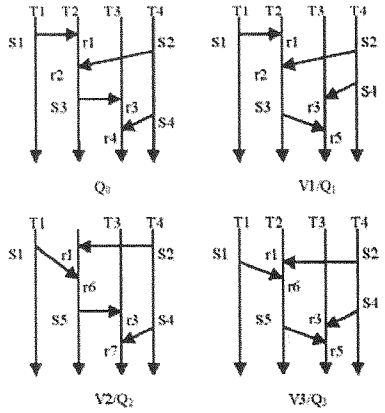

Figure 1. Example of reachability testing [7]

associated to a communication primitive (send or receive). The model considers both blocking and non-blocking receives, such that all possible interleaving between send-receive pairs can be represented. $Prog$ is associated with a Parallel Control Flow Graph ($PCFG$), which is composed of the CFG^p (for $p = 0 \ldots n-1$) and by the representation of the communication between the processes. A synchronization edge (sync-edge) (n_i^a, n_j^b) links a *send* node in a process a to a *send* node in a process b. These edges represent the possibility of communication and synchronization between processes.

A set of coverage testing criteria is defined, based on ($PCFG$): All-Nodes; All-Edges; All-Nodes-R; All-Nodes-S and All-Edges-S (related to control and synchronization information) and All-C-Uses; All-P-Uses; All-SUses; All-S-C-Uses and All-S-P-Uses (related to data and communication information) [14]. The All-Edges-S criterion requires that the test set executes paths that cover all the sync-edge associations of the concurrent program under testing; the All-S-uses criterion requires that the test set executes paths that cover all the *s-use* associations. An *s-use* is an association between a node n^p, that contains a definition of a variable x, and a sync-edge that contains a communications use of x.

An example of a $PCFG$ is shown in Figure 2. There are four processes, consisting of two different codes. Synchronization pairs are represented by dotted lines — for example, the pair $(2^0, 2^m)$ is one sync-edge between process p^0 and p^m. Each sync-edge is associated with one or more s-use associations, and related to a variable represented in $PCFG$.

Nondeterminism is the key issue addressed in this test model. As it is impossible to determine statically when a synchronization is feasible, a conservative approach is assumed, where every pair of send and receive events which have the appropriate types are considered

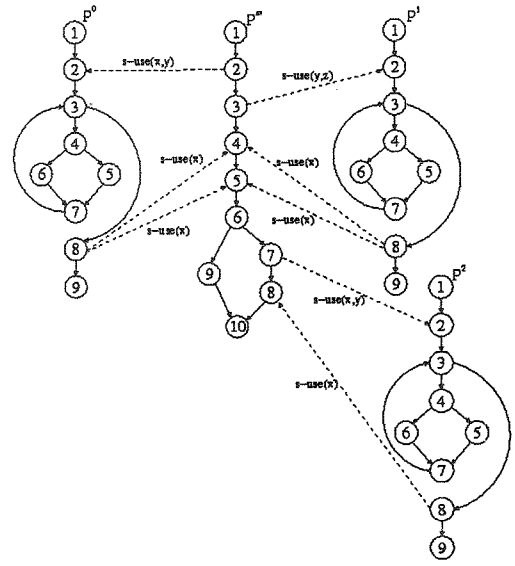

Figure 2. Example of a Parallel Control Flow Graph

as a possible matching. Considering the example of Figure 1, the required sync-edges are: (s_1^{T1}, r_1^{T2}), (s_2^{T4}, r_1^{T2}), (s_4^{T4}, r_1^{T2}), (s_1^{T1}, r_2^{T2}), (s_2^{T4}, r_2^{T2}), (s_4^{T4}, r_2^{T2}), (s_1^{T1}, r_3^{T3}), (s_2^{T4}, r_3^{T3}), (s_3^{T2}, r_3^{T3}), (s_4^{T4}, r_3^{T3}), (s_1^{T1}, r_4^{T3}), (s_2^{T4}, r_4^{T3}), (s_3^{T2}, r_4^{T3}), (s_4^{T4}, r_4^{T3}). Some sync-edges are infeasible (e.g., (s_1^{T1}, r_3^{T3})) but are required. Using controlled execution [2], it is possible to force the execution of feasible sync-edges.

It is not necessary to execute all combinations of possible synchronization as long as at least one execution of each sync-edge pair is included. A problem in this approach is the high number of infeasible sync-edges that are generated and need to be analysed. Nevertheless, it is interesting because it uses information generated statically to direct the selection of test cases and to assess the coverage of the program under test. We believe that the choice of the test case can influence the results obtained, improving the overall testing activity quality. This has led directly to the test strategy presented in the next section.

3. Proposed test strategy

In this paper we propose a test strategy that combines both reachability testing and coverage testing to execute synchronization events. The main motivation of this strategy is to improve coverage testing; however, the approach can also be applied to improve reachability testing perfor-

mance. In this case, reachability testing can be applied using an approach that selectively exercises a set of sync-sequences according to a specified coverage testing criterion. Exhaustive testing is not always practical and they pointed out the need to use mechanisms to guide the selection of the sync-sequences during reachability testing [7].

In Figure 3 the proposed test strategy is illustrated. This figure does not show all the steps necessary to apply the coverage testing criterion, only those important to our strategy.

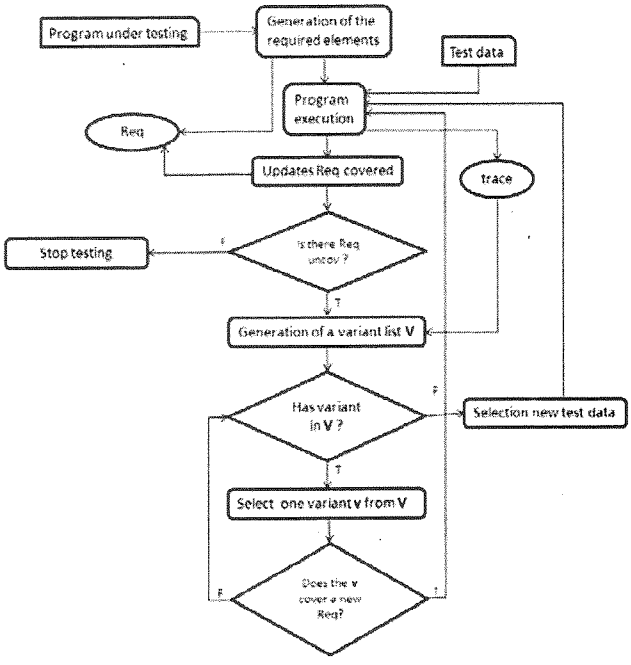

Figure 3. Test Strategy using Reachability Testing and Coverage Testing Criterion

First, a required elements list Req is generated from a given concurrent program, based on the all-s-uses and all-edges-s criteria. An initial test dataset is produced and the program is executed to generate an execution trace, containing a record of all the nodes and the sync-edges executed by the test dataset. The elements covered by the test dataset execution are marked in Req and any required element not yet covered identified. Usually, a procedure is used to select a new test dataset to improve the coverage of Req.

Considering reachability testing in this context, the next step is the generation of a list V of the variants, based on the sync-edges executed. For each sync-edge all possible variants are then generated. The difference here from reachability testing, is that only the variants required to cover a required element that is not yet covered are included. Therefore, when a new variant v is selected from V, it is verified only if it executes a new requirement of Req. Otherwise, another variant is selected or a new test dataset is generated (when V is empty). The procedure to execute a variant v is the same as that defined by Lei and Carver [7]: the controlled execution ensures that the synchronization of the v always occurs during the execution. After execution of the variant v, the execution trace is obtained and the required elements covered for this execution are marked in list Req. Considering now the variant v, new variants are generated and added to the list of the variants V. The procedure to execute variants or new test datasets is repeated while any required elements remain to be covered.

4. Experimental Study

In this section, we present an experimental study that indicates that the approach combining reachability and coverage criteria testing improve overall testing quality. The ValiMPI tool was used to conduct this study. ValiMPI is a tool developed to test concurrent programs implemented in MPI (Message Passing Interface), proposed originally to support the coverage testing mentioned in Section 2 [13, 6]. ValiMPI functionality has been extended to implement the reachability testing strategy proposed by Lei and Carver [7] and hence test the strategy proposed in this paper.

Eight different MPI programs were used in this study, implementing classical concurrency algorithms. The complexity is given by the number of sends s and receives r of each program: **sieve of Eratosthenes** - (7s and 9r) an algorithm for finding all prime numbers up to a specified integer [9]; **gcd** - (7s and 7r) to calculate the greatest common divisor of three numbers, using successive subtractions between two numbers until one of them is zero; **mmult** - (15s and 27r) to implement matrix multiplication using domain decomposition; **philosophers** - (11s and 10r) to implement the dining philosophers problem; **pairwise** - (16s and 16r) where each process n_i receives a data X_i and is responsible for computing the interactions $I(X_i, X_j)$,for $i \neq j$. For this, a structure with N channels is used, where each communication channel represents a pair source-destination — these channels are used to connect the N tasks into a unidirectional ring; **reduction** - (4s and 4r) to implement the reduction operation of distributed data, considering add, multiplication, greater than and less than operations; **qsort** - (28s and 52r) to implement quicksort, based on the parallel algorithm presented in Grama [5]; and **jacobi** - (23s and 37r) to implement Jacobi-Richardson iteration for solving a linear system of equations.

Three different test scenarios were executed:

1. **Selection of adequate test case using coverage criteria (CovT)**: using the criteria all-s-uses and all-edges-s, test cases were manually generated to exercise the required elements of these criteria, s-use associations

and sync-edges, respectively. Infeasible elements were identified to evaluate the coverage of an initial test set.

2. **Application of reachability testing (RT)**: using the initial test set generated during Scenario CovT, reachability testing was undertaken, according to the algorithm proposed by Lei and Carver [7].

3. **Application of our test strategy (RTCovT)**: using the criteria all-s-uses and all-edges-s, (related to synchronization), reachability testing was executed guided by the required elements of these criteria, following the steps discussed in Section 3.

Table 1 presents the sync-sequences generated by Reachability Testing (column RT) and by our test strategy (column RTCovT), using two different test sets: T_1, generated by CovT and containing test cases adequate to execute both sync-edges and s-uses; and T_2, which is a subset of T_1 containing only effective test cases (i.e. T_2 contains only test case contributing to the execution of the sync-edges). RT executes, for each test case, all variants of the each sync-edge, even those variants already executed previously. For this reason, the number of the sync-sequences generated by RT is higher than the sync-sequences generated by our test strategy. These results indicate that is possible to reduce the cost of the reachability testing using coverage testing. A fundamental problem with reachability testing is to decide when the testing activity can be considered complete; our test strategy contributes to the work on this problem. We compared the advantages of using our test strategy compared to the alternatives discussed previously.

Table 1. Number of the sync-sequences executed

Programs	T1	T1		T2	T2	
		RT	RTCovT		RT	RTCovT
sieve	10	60	13	4	20	7
gcd	13	24	14	7	14	9
mmult	8	48	11	4	24	5
philosophers	1	1680	2	1	1680	2
pairwise	4	4	4	1	1	1
reduction	4	4	4	1	1	1
qsort	3	794	18	3	266	16
jacobi	9	1710	13	6	377	11

Table 2 shows the results of the coverage obtained using the coverage testing (CovT) and our test strategy (RTCovT) for all-edges-s and all-s-uses criteria. For this analysis, for each program, the same test set T (generated on an *ad-hoc* basis) was used to execute the two test scenarios and the two coverage criteria. Our test strategy (RTCovT) indicates the potential to improve the criteria coverage because our strategy executes a greater number of sync-edges and s-uses than the traditional coverage testing, establishing that it is a good strategy to reduce the overall application cost of the test. Some of the programs in the test had no improvement in coverage because in these cases the T set already covered all feasible elements for the criteria.

Table 2. Coverage using coverage criteria and the test strategy

Programs	All-Edges-S		All-S-Uses	
	CovT	RTCovT	CovT	RTCovT
sieve	80.95%	90.48%	56.67%	76.67%
gcd	100.00%	100.00%	80.00%	85.00%
mmult	93.33%	93.33%	94.74%	94.74%
philosophers	100.00%	100.00%	75.00%	75.00%
pairwise	100.00%	100.00%	83.33%	83.33%
reduction	100.00%	100.00%	100.00%	100.00%
qsort	51.61%	89.25%	43.54%	67.35%
jacobi	100.00%	100.00%	84.06%	85.51%

Table 3 shows the results for the Jacobi algorithm example as different test cases ($tc1$ to $tc9$) are processed. Our testing strategy always provides better test coverage and maximal coverage is achieved after only five test cases have been considered. Table 4 provides similar results for the mmult algorithm example with test cases $tc1$ to $tc8$. For this example maximum coverage is achieved after only two of the test cases have been considered.

Table 3. Evolution of the jacobi program coverage

testcases	All-Edges-S		All-S-Uses	
	CovT	RTCovT	CovT	RTCovT
$tc1$	19.30%	19.30%	8.70%	8.70%
$tc2$	38.60%	49.12%	26.09%	34.78%
$tc3$	82.46%	94.74%	60.87%	71.01%
$tc4$	84.21%	96.49%	68.12%	78.26%
$tc5$	94.74%	100.00%	78.26%	82.61%
$tc6$	94.74%	100.00%	78.26%	82.61%
$tc7$	94.74%	100.00%	78.26%	82.61%
$tc8$	100.00%	100.00%	82.61%	84.06%
$tc9$	100.00%	100.00%	84.06%	85.51%

5. Conclusions

In this paper we have presented a new test strategy to validate concurrent programs, using a combination of coverage criteria and reachability testing. The coverage criteria are used both to select test cases and to determine the execution of new synchronizations, while the reachability testing is used to select appropriate synchronizations to be executed.

Table 4. Evolution of the mmult program coverage

testcases	All-Edges-S		All-S-Uses	
	CovT	RTCovT	CovT	RTCovT
$tc1$	60.00%	93.33%	63.16%	89.47%
$tc2$	60.00%	93.33%	68.42%	94.74%
$tc3$	73.33%	93.33%	78.95%	94.74%
$tc4$	86.67%	93.33%	89.47%	94.74%
$tc5$	86.67%	93.33%	89.47%	94.74%
$tc6$	86.67%	93.33%	89.47%	94.74%
$tc7$	86.67%	93.33%	89.47%	94.74%
$tc8$	86.67%	93.33%	94.74%	94.74%

The combination of the two approaches has the potential to deliver significant reduction in the overall testing cost. Due to the high number of synchronizations in a typical concurrent program, the execution of these synchronizations using reachability testing alone can be impractical; while in the case of the coverage criteria used by itself, these synchronizations generate a high cost because of the number of infeasible synchronizations that must be analyzed.

The test strategy described in this paper contributes in two ways: 1) by using structural criteria to minimize the number of sequences in reachability testing; and 2) by guiding the generation of test cases based on the structural criteria, using reachability testing to increase the test coverage. An experimental study has been undertaken to evaluate this approach. The results indicate that is promising to adopt this test strategy, with an improvement in test coverage in every case considered.

Finally, we plan to evaluate the proposed test strategy in terms of revealing faults. Preliminary results have demonstrated that our strategy is effective in detecting faults. The test sets discussed above were evaluated using the fault taxonomy presented in [4] and 84.8% of the seeded defects were revealed on average. Further studies are being developed using different fault taxonomies and comparing the effectiveness of our strategy with the effectiveness of reachability testing.

6 Acknowledgments

The authors would like to thank CAPES and FAPESP, Brazilian funding agencies, for the financial support, under Capes process 1191/10-1 and FAPESP processes: 2008/04614-5, 2010/02839-0.

References

[1] R. Carver and Y. Lei. Distributed reachability testing of concurrent programs. *Concurrency and Computation: Practice and Experience*, 22(18):2445–2466, 2010.

[2] R. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Software*, pages 74–86, Mar. 1991.

[3] S. K. Damodaran-Kamal and J. M. Francioni. Nondeterminacy: Testing and debugging in message passing parallel programs. In *3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 118–128, New York, May 1993.

[4] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[5] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Addison Wesley, 2003.

[6] A. C. Hausen, S. R. Vergílio, S. Souza, P. Souza, and A. Simão. A tool for structural testing of MPI programs. In *8th IEEE Latin-American Test Workshop*, march 2007.

[7] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE TSE*, 32(6):382–403, June 2006.

[8] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *Proceedings of the ACM SIGSOFT symposium on the foundations of software engineering*, pages 533–536, New York, NY, USA, 2007. ACM.

[9] M. J. Quinn. *Parallel Computing : Theory and Practice*. McGraw-Hill, New York, 2nd. edition, 1994.

[10] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transaction Software Engineering*, 11(4):367–375, Apr. 1985.

[11] C. Robinson-Mallett, R. M. Hierons, J. Poore, and P. Liggesmeyer. Using communication coverage criteria and partial model generation to assist software integration testing. *Software Quality Control*, 16(2):185–211, 2008.

[12] F. S. Sarmanho, P. S. L. Souza, S. R. S. Souza, and A. S. Simão. Structural testing for semaphore-based multithread programs. In *International Conference on Computational Science, LNCS*, volume 5101, pages 337–346, 2008.

[13] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simão, T. G. Bliscosque, A. M. Lima, and A. C. Hausen. Valipar: A testing tool for message-passing parallel programs. In *International Conference on Software knowledge and Software Engineering (SEKE05)*, pages 386–391, Taipei-Taiwan, 2005.

[14] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simão, and A. C. Hausen. Structural testing criteria for message-passing parallel programs. *Concurrency and Computation: Practice and Experience*, 20:1893–1916, mar 2008.

[15] J. Takahashi, H. Kojima, and Z. Furukawa. Coverage based testing for concurrent software. In *28th International Conference on Distributed Computing Systems Workshops, 2008.*, pages 533–538, June 2008.

[16] R. N. Taylor, D. L. Levine, and C. Kelly. Structural testing of concurrent programs. *IEEE Transaction Software Engineering*, 18(3):206–215, Mar. 1992.

[17] W. E. Wong, Y. Lei, and X. Ma. Effective generation of test sequences for structural testing of concurrent programs. In *10th IEEE International Conference on Engineering of Complex Systems (ICECCSt'05)*, pages 539– 548, 2005.

[18] C.-S. D. Yang. *Program-Based, Structural Testing of Shared Memory Parallel Programs*. PhD thesis, University of Delaware, 1999.

ISBN-10: 1-891706-29-2 (paper)
ISBN-13: 978-1-891706-29-5

Additional Copies can be ordered from:
Knowledge Systems Institute Graduate School
3420 Main Street
Skokie, IL 60076, USA
Tel:+1-847-679-3135
Fax:+1-847-679-3166
Email:office@ksi.edu
http://www.ksi.edu

Proceedings preparation, editing and printing are sponsored by
Knowledge Systems Institute Graduate School

Printed by Knowledge Systems Institute Graduate School