pTest: An Adaptive Testing Tool for Concurrent Software on Embedded Multicore Processors

Shou-Wei Chang, Kun-Yuan Hsieh and Jenq Kuen Lee
Department of Computer Science
National Tsing-Hua University, Hsin-Chu, Taiwan
{swchang, kyshieh}@pllab.cs.nthu.edu.tw, jklee@cs.nthu.edu.tw

***Abstract*—More and more processor manufacturers have launched embedded multicore processors for consumer electronics products because such processors provide high performance and low power consumption to meet the requirements of mobile computing and multimedia applications. To effectively utilize computing power of multicore processors, software designers interest in using concurrent processing for such architecture. The master-slave model is one of the popular programming models for concurrent processing. Even if it is a simple model, the potential concurrency faults and unreliable slave systems still lead to anomalies of entire system. In this paper, we present an adaptive testing tool called pTest to stress test a slave system and to detect the synchronization anomalies of concurrent software in the master-slave systems on embedded multicore processors. We use a probabilistic finite-state automaton(PFA) to model the test patterns for stress testing and shows how a PFA can be applied to pTest in practice.**

I. Introduction

Embedded multicore processors have been widely adopted in the consumer electronics market to meet the ever-increasing performance requirements of mobile computing and multimedia applications. Such architecture is conventionally composed of clusters of processing cores connected by the on-chip communication networks, high-bandwidth memory subsystems, and integrated peripheral interfaces. The TI OMAP [1] dual-core processors adopting heterogeneous cores to balance performance requirements and save power consumption is a typical example of a multicore design. One of the popular programming models on the embedded multicore systems is the master-slave model. The master-slave model is a simple model for concurrent processing and is widely used on asymmetric multicore processors to utilize distributed computing resources more effectively [2]. In the master-slave model, a cluster of processing cores is classified into two categories, the master and slave, where the executable processes in the slave cores are controlled by the remote processes in the master processing cores.

Although the master-slave model is a simple model, the embedded multicore system that adopts it still may crash by unreliable software. Two of the obvious factors that result in the failures of embedded multicore systems are the crash of the slave system under heavy loads and synchronization anomalies, such as deadlock and starvation, which may occur in concurrent programs. The common functional testing methodologies are not sufficient to find these software faults during the development stage. Various tools have been developed to test concurrent software and the robustness of complex systems. For example, ConTest [3] and CHESS [4] are significant tools for finding concurrency bugs in multi-threaded software by systematically exploring all possible thread interleaving. ConTest debugs multi-threaded programs by randomly interleaving the execution of threads. Compared to ConTest, CHESS uses model checking techniques [5] to provide higher fault coverage. However, model checking is not efficient when searching infinite state spaces and the implementation may be incorrect due to the demands of disciplined design [6].

In this paper, we present an adaptive testing tool called pTest on embedded multicore systems to perform a stress test on a slave system for verifying the correctness of services provided by the slave system and detecting the synchronization anomalies of concurrent processes in the embedded multicore systems. pTest constructs a probabilistic finite-state automaton(PFA) [7], [8] from the probability information and the regular expression provided by users. We use a PFA to describe the slave system services and generate each test pattern as a set of the slave system services arranged in rational order. A PFA provides the quantitative, probabilistic information to resolve nondeterministic choices about which elements to be included in the test patterns. The concept of pTest is first to construct the PFA from the regular expression to automatically generate adaptive test patterns and then according to these test patterns, a committer on the master core automatically issues remote commands to test the slave system at runtime. To precisely construct the PFA, the probability distributions are forwarded to the pattern generator of pTest. pTest records the execution status of test activities and the state of processes in master-slave systems. When pTest detects that the slave system crashes or faults, it terminates the current job and helps users reproduce the bugs.

We have applied pTest on TI OMAP dual-core processor to examine its effectiveness. In the current design, we assume that most users do not know the probability distributions given to pTest to construct the relative PFA. The knowledge about probability distributions can be learned through system profiling or by providing analytic model to estimate the transition activity of real systems. We give the case study to discuss how pTest models the runtime system called pCore [9] running on the DSP core in the TI OMAP dual-core processor as slave. The evaluation of effectiveness of pTest runs the test

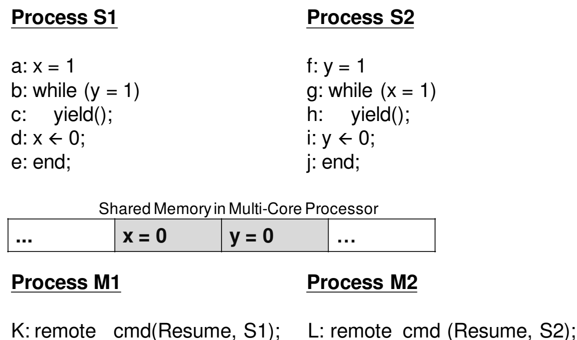

Fig. 1. Example of a concurrency fault.

cases to examine the robustness of pCore and to find out the concurrency bugs from dual-core programs.

The remainder of this paper is organized as follows. Section II first discusses the concept of stress testing and then provides an overview of pTest. Section III presents the formal description and the algorithm of our proposed testing methodology. Section IV shows case studies and evaluates the effectiveness of pTest. Finally, the conclusion and the future work are presented in Section V.

II. Overview

A. Stress Testing

In practice, stress testing is an efficient method used to examine the robustness of complex systems under heavy load [10], [11]. Stress testing can be used to detect garbage collection issues, memory leaks, and concurrency faults caused by the unpredictable progress of concurrent processes. For example, assume that there are four processes, S1, S2, M1, and M2, running in the embedded multicore system as shown in the Figure 1. Both S1 and S2 are suspended in the slave system and the other two are run in the master system. The scheduling policy of the slave system and the master system are preemptive priority-based and time-sharing scheduling policy respectively. The priority of the process *S1* is lower than that of the process *S2*. In addition, the process *M1* invokes the remote process *S1* by calling *remote_cmd(Resume, S1)* and the function *yield()* means that the current process yields the processor to other waiting processes. The system can finish all of the processes in the execution order $L \rightarrow f \rightarrow g \rightarrow K \rightarrow i \rightarrow j \rightarrow a \rightarrow b \rightarrow d \rightarrow e$. However, if the execution order is $K \rightarrow a \rightarrow L \rightarrow f \rightarrow g \rightarrow h \rightarrow b \rightarrow c \rightarrow g \rightarrow h \rightarrow \ldots$, the system enters the deadlock state. The state *d, e, i, j* are unreachable. The callback function wrapper is one of approaches to determine if a process is terminating or not. If processes do not terminate or stay in the same state for a period of time, the system may contain synchronization anomalies. The potential concurrency faults can be found out by performing a stress test. Furthermore, the effects of code coverage influences the quality of fault detection. To detect more software faults, the code coverage analysis is a useful information for stress testing on large software systems.

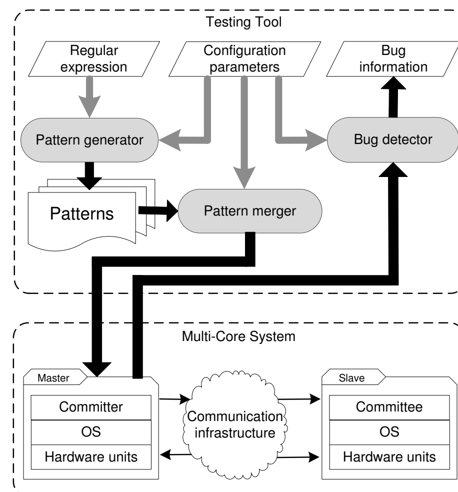

Fig. 2. Software architecture of pTest.

B. Overview of the Adaptive Testing Tool

pTest is designed to execute a stress test on a runtime system running on a specialized processing unit of the embedded multicore processors. pTest runs on the master system to issue a large number of commands for stress testing the runtime behavior of the slave system. Figure 2 shows the software architecture of pTest, which is comprised of three key components described as follows:

- **Pattern generator** The work of pattern generator is to produce test patterns by running a PFA. It interprets the regular expression and probability distributions to construct the corresponding PFA. In addition, the knowledge about the probability distributions is forwarded to the pattern generator in advance. Both the regular expression and the corresponding PFA recognize the same patterns that is a sequence of the slave system services arranged in rational order. Each test pattern represents a set of the possible slave system services associated to a process on slave system.
- **Pattern merger** Because each process in the slave system is controlled by the remote processes in the master system, the process execution order in the master system affects the process execution order in the slave system. To simulate the concurrent execution in master-slave systems, the pattern merger extracts subsequences from each test pattern produced by the pattern generator and then systematically merges all subsequences into one final test pattern. The work of the pattern merger is to generate the interleaved test patterns. It is similar to a process scheduler.
- **Bug detector** The bug detector tracks the progress of test activities until it detects the potential system failures and then it terminates the test activity that results in these failures. The execution records of each test activity including the state of a process of a slave system and the execution status of commands are reserved by a slave

system and the committer. When the potential system failures have been detected, the bug detector dumps the related information to help users reproduce the bugs.

Each processing core in embedded multicore processors is connected by the on-chip communication network. The common inter-processor communication mechanisms adopted in such processors are processors polling events through shared memory and sending events by triggering interrupts. The master-slave systems implement the software communication infrastructure based on such communication mechanisms to exchange messages between master core and slave core. pTest can use a native communication library to link committer and committee across cores. According to the regular expression and configuration parameters, pTest automatically generated the adaptive test pattern to the committer. The committer issues the remote commands for the committee through the software communication infrastructure to start the testing work.

III. Enabling Adaptive Testing

A. Probabilistic Finite-State Automata

Probabilistic finite-state automata are used in various domains such as mutation testing [12], machine translation [13], and bioinformatics [14]. The PFA is a promising model to specify systems that introduces probabilistic choice to deal with possible actions. For example, in practice, a hidden Markov model (HMM) [15] that emits a sequence of symbols according to probability distributions is the most common type of probabilistic finite-state automata. In this section, we give a formal definition of the PFA that is simplified by removing initial state probabilities and final state probabilities to meet our requirement in this paper. We introduce the concept of a PFA to generate test patterns according to probability distributions.

Definition 1: A probabilistic finite-state automaton is a six-tuple $(Q, \Sigma, \delta, q_0, F, P)$, where:

1) Q is a finite set of states;
2) Σ is a finite alphabet;
3) $\delta \subseteq Q \times \Sigma \times Q$ is the state transition relation;
4) $q_0 \in Q$ is the initial state;
5) $F \subseteq Q$ is the set of final states;
6) $P : \delta \to \mathbb{R}^+$ is the transition probability function such that:

$$\sum_{\forall a \in \Sigma, q' \in Q} P(q, a, q') = 1, where\ q \in Q. \quad (1)$$

Figure 3 shows a simple graphical representation of PFA with three states, $Q = \{q_0, q_1, q_2\}$, only one initial state, q_0, a four-symbol alphabet, $\Sigma = \{a, b, c, d\}$, and four state transition probabilities, $\{P(q_0, a, q_1) = 0.6, P(q_0, b, q_2) = 0.4, P(q_1, c, q_1) = 0.3, (q_1, d, q_2) = 0.7\}$. Each transition has an associated probability that is a real number strictly between 0 and 1. The sum of the probabilities of all possible transition for a given state is equal to 1. The regular expression describing the language recognized by this simple PFA is $(\mathrm{ac}^*\mathrm{d}) \mid \mathrm{b}$.

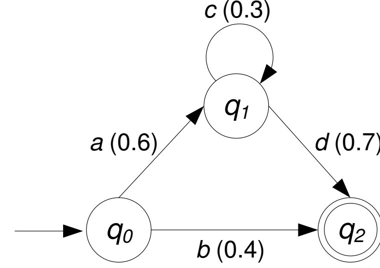

Fig. 3. A simple PFA.

B. Adaptive Testing Mechanism

As mentioned previously in Section II-B, there are three key components, which include pattern generator, pattern merger, and bug detector, in pTest. Algorithm 1 written in pseudocode illustrates the procedure of the adaptive testing.

Algorithm 1 : Adaptive Testing Procedure

```
procedure AdaptiveTest(RE, n, s, op)
 1: for i = 1 to n do
 2:     T[i] ← PatternGenerator (RE, PD, s)
 3: end for
 4: M ← PatternMerger(T, n, op)
 5: CreateChildProcess()
 6: if CurrentProcess = ChildProcess then
 7:     BugDetector(op)
 8: else
 9:     Committer(M)
10: end if
end procedure
```

The configuration parameters are *n*, *s*, and *op*. The procedure, $AdaptiveTest$, of pTest initially invokes the procedure, $PatternGenerator$, to run the PFA to generate the set T of *n* test patterns. The size of each test pattern is denoted by *s*. Then the generated test patterns are systematically merged into one, M, by invoking the procedure, $PatternMerger$. The work of $PatternMerger$ is to generate the interleaved test patterns to explore all feasible process interleaving in a master-slave system. It acts as a scheduler to perform interesting concurrency scenarios for systematical testing. In addition, the bug detector is run as a new process to fully monitor the progress of the testing. The parameter, *op*, indicates the pattern merger to produce the specific test pattern that can help the bug detector find out the specific bug such as slave system crashes or concurrency faults. The bug detector tracks the execution history of each test activity to find out the potential failures in master-slave systems. Eventually, according to the test pattern, the committer issues the corresponding commands to enable the remote testing for a slave system.

Algorithm 2 written in pseudocode illustrates the procedure of the pattern generator. The pattern generator generates a test pattern with size *s* at each invocation. A generated test pattern

Algorithm 2 : Pattern Generator Procedure

procedure $PatternGenerator$(*RE, PD, s*)

1: *NFA* $\leftarrow$ $ConvertToNFA$(*RE*)
2: *PFA* $\leftarrow$ $ConstructPFA$(*NFA, PD*)
3: *Q* $\leftarrow$ *Initial state in the PFA*
4: $P[1] \leftarrow Q$
5: **for** $i = 2$ to s **do**
6:     **if** *Q has probabilistic choices* **then**
7:         *Q'* $\leftarrow$ $MakeChoice$(*Q, PFA*)
8:         $P[i] \leftarrow$ *Q'*
9:         *Q* $\leftarrow$ *Q'*
10:     **else**
11:         *Q'* $\leftarrow$ *The reachable state*
12:         $P[i] \leftarrow$ Q'
13:         *Q* $\leftarrow$ *Q'*
14:     **end if**
15: **end for**
16: **return** P

end procedure

is denoted by P. It first constructs the nondeterministic finite-state automaton(NFA) by interpreting the regular expression, *RE*, and then converts the NFA to the PFA by attaching the probability distributions, *PD*. When the state transition occurs, the current state in the PFA calls the procedure, $MakeChoice$, to obtain the next state if it has a probabilistic choice to make.

C. State Recording of Concurrent Processes

We define the related expression to clearly describe the state recording of concurrent processes in master-slave systems for multicore processors. The state recording of concurrent processes is useful for the bug detector to monitor the progress of the testing. Each process in a slave system is controlled by the corresponding remote process in a master system. We assume that there is a one-to-one correspondence between processes in a slave system and processes in a master system and build the expression to meet our requirement in this paper.

Definition 2: The expression of the state recording of concurrent processes in a master-slave system is a five-tuple $(q_m, q_s, TP, SN, \delta_S)$, where:

1) q_m is a state of a master process;
2) q_s is a state of a slave process;
3) TP is the test pattern for the slave process;
4) SN is the sequence number of the state of the test pattern;
5) δ_S is the subsequence of the test pattern.

Figure 4 shows a sample expression of state recording of concurrent processes with two state records, CP_1 and CP_2. In this sample expression, the set of states for a master process is $\{m_1, m_2, m_3\}$ and the set of states for a slave process is $\{s_1, s_2\}$. The test pattern has three states, $\{p_1, p_2, p_3\}$. In the record CP_1, the first field is the last state of a master process before it enters a state that issues remote commands to control

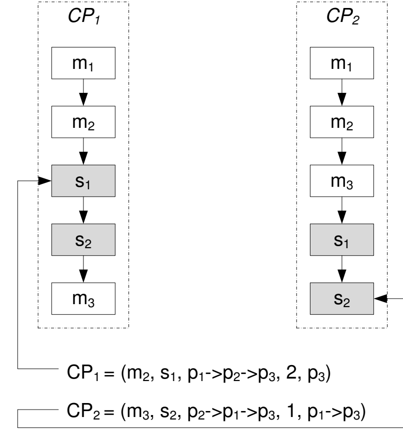

Fig. 4. A sample expression of state recording of concurrent processes.

the corresponding slave process. The current state of a slave process and the given test pattern are stored, respectively, in the second and third field. Furthermore, the sequence number given in the fourth field of the record CP_1 indicates that the current state of the test pattern is p_2. The fifth field of the record CP_1 represents the subsequence of the test pattern, p_3, should be executed in the next time. The implication of the record CP_2 is the same as the record CP_1.

IV. Evaluation

This section presents the evaluation of our proposed testing mechanism described in Section III. We have implemented pTest on OMAP5912 OSK platform. This platform contains a heterogeneous dual-core system-on-a-chip(SoC) processor, OMAP5912, which is composed of a 192-MHz ARM 926EJ-S processor, and a 192-MHz TI C55x DSP processor with 160 Kbytes of internal memory. The two processor can exchange events and data via four mailboxes and 250 Kbytes of shared internal SRAM. pTest was executed in Linux running on the ARM core as master and performing the stress test on a runtime system called pCore running on the DSP core as slave. The communications between master system and slave system on OMAP5912 processor were handled by a middleware called pCore Bridge [16] which provides the basic communication mechanisms.

A. Case Study: pCore

We now describe the real testing case to demonstrate the usage of pTest. We applied the PFA of pCore to pTest. pCore is a microkernel designed for specialized processing units, such as a VLIW DSP processor, of a embedded multicore processor. The basic execution unit in pCore is a task referred to a thread in the POSIX standard [17]. pCore supports up to 16 concurrent threads on the specialized processing unit. Each task is typically forked with a unique priority by a thread in Linux to perform sub-functions. pCore provides preemptive priority-based scheduling policy that always schedules the task with highest priority to run. Two main features in the development of pCore are providing efficient kernel services with tiny

kernel size and supporting dual-core/multicore communication protocols.

TABLE I
KERNEL SERVICES OF PCORE FOR TASK MANAGEMENT

	Abbreviation	Description
task_create	TC	Create a task
task_delete	TD	Delete a task
task_suspend	TS	Suspend a task
task_resume	TR	Resume a task
task_chanprio	TCH	Change the priority of a task
task_yield	TY	Terminate the current running task

Table I lists the related kernel services provided by pCore for task management. In the development of concurrent programs under master-slave model, each task in pCore is controlled by the corresponding remote thread in Linux. It is a one-to-one correspondence between tasks in pCore and threads in Linux. By surveying the activities of tasks in pCore, the regular expression describing the behavior of tasks can be modeled as

$$RE = TC((TCH)^* \mid TSTR(TCH)^*)^*(TD\$ \mid TY\$). \quad (2)$$

Task creation is the initial state during the life cycle of a task in (2). After a task is created with a unique priority, the rest of the task operations include priority change, suspending task, resuming task and task termination can be performed in a legal execution order. For example, the task resuming operation can be performed only when the corresponding task is suspended. pTest interpreted the above regular expression and the given probability distributions to construct the corresponding PFA as shown in Figure 5. The probability distributions were obtained

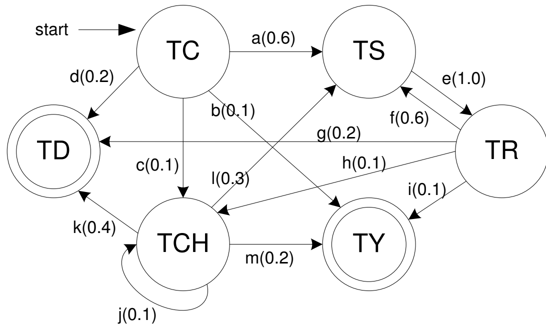

Fig. 5. The graphic representation of the PFA for pCore.

through our experiences in developing concurrent programs under the master-slave model for pCore on OMAP systems. The pattern generator of pTest ran the PFA of pCore to produce the test patterns for pattern merger of pTest. The test patterns were used to verify if pCore would meet the demand for task services.

B. Fault Discovery

To evaluate the effectiveness of pTest, we designed two test cases to test the robustness of pCore and to check the correctness of concurrent programs in our dual-core environment. When the bug detector of pTest detected bugs in pCore or concurrency bugs, we reproduced them according to the information reported by the bug detector of pTest. In the first test case, pTest kept the number of active tasks at 16 in pCore to execute the stress test on pCore. All of 16 active tasks performed the same quick-sort algorithm to individually sort 128 integer elements. The size of integer data is 2 bytes and the stack size of each task is 512 bytes. pTest continued to create tasks and removed them when their work was done. During the first testing period, pTest detected the crash of pCore that was caused by the failure of garbage collection.

In the second test case, we attempted to verify if pTest could find the potential concurrency faults such as deadlock. We implemented a buggy version of the dining philosophers problem that could lead to deadlock. The algorithm consisted of three concurrent tasks in pCore and three shared resources that were mutually exclusive. A task needed two shared resources to resume its execution. We set the patten merger of pTest to produce the test pattern that forced these tasks to complete several set of cyclic execution sequences. pTest kept tracing the states of these tasks to determine if these tasks were terminating or not. A potential deadlock situation was also discovered by pTest during the second testing period.

V. CONCLUSIONS AND FUTURE WORK

Concurrent software is more difficult to test than sequential software. Moreover, the multicore programming brings more challenge for testing. This paper proposes an adaptive testing tool called pTest for concurrent software on embedded multicore processors that adopt the master-slave model. pTest has been implemented on TI OMAP dual-core processor. The regular expression describing the behavior of each task in pCore is inputted to pTest to construct the corresponding PFA. pTest uses the PFA to model the test patterns for stress testing pCore and also detects the potential concurrency faults of dual-core programs. The preliminary evaluation shows that pTest can be a suitable testing tool for embedded multicore processors.

The concepts of probability as well as the feasible combinations of test patterns provide us a novel idea to systematically test a concurrent program through its possible execution paths. To verify further the efficiency of pTest, we plan to identify the influence of probability distributions on the generation of test pattern for different testing scenarios. Moreover, pTest currently does not consider the problems of that the replicated test patterns can reduce the effectiveness of pTest. The fault coverage of pTest also does not be verified. We would like to examine these problems of pTest in the future work.

VI. ACKNOWLEDGMENT

This research was supported in part by the NSC under grant nos. NSC 97-2218-E-007-009, NSC 97-2218-E-007-008 and NSC 96-2220-E-007-030, and by the MOEA research project under grant nos. 95-EC-17-A-01-S1-034 and 96-EC-17-A-01-S1-034 in Taiwan.

REFERENCES

[1] *OMAP5912 Application Processor*, Texas Instuments.

[2] S. L. Shee, A. Erdos, and S. Parameswara, "Heterogeneous multiprocessor implementations for jpeg: a case study," in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, 2006, pp. 217–222.

[3] 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*, vol. 15, no. 3-5, pp. 485–499, Feb. 2003.

[4] M. Musuvathi and S. Qadeer, "Fair stateless model checking," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, 2008, pp. 362–371.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

[6] A. Groce and R. Joshi, "Random testing and model checking: building a common framework for nondeterministic exploration," in *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08)*, 2008, pp. 22–28.

[7] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, and R. C. Carrasco, "Probabilistic finite-state machines-part i," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 7, pp. 1013–1025, Jul. 2005.

[8] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, and R. C. Carrasco, "Probabilistic finite-state machines-part ii," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 7, pp. 1026–1039, Jul. 2005.

[9] K.-Y. Hsieh, Y.-C. Lin, C.-C. Huang, and J.-K. Lee, "Enhancing microkernel performance on vliw dsp processors via multiset context switch," *Journal of Signal Processing Systems*, vol. 51, no. 3, pp. 257–268, Jun. 2008.

[10] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional, 1999.

[11] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," in *Proceedings of the 4th conference on USENIX Windows Systems Symposium*, 2000, pp. 59–68.

[12] R. M. Hierons and M. G. Merayo, "Mutation testing from probabilistic finite state machines," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION'07)*, 2007, pp. 141–150.

[13] S. Bangalore and G. Riccardi, "Stochastic finite-state models for spoken language machine translation," in *Proceedings of the NAACL-ANLPWorkshop on Embedded Machine Translation Systems*, 2000, pp. 52–59.

[14] E. L. L. Sonnhammer, G. von Heijne, and A. Krogh, "A hidden markov model for predicting transmembrane helices in protein sequences," in *Proceedings of the Sixth International Conference on Intelligent Systems for Molecular Biology (ISMB)*, 1998, pp. 175–182.

[15] L. R. Rabiner and B. H. Juang, "An introduction to hidden markov models," *IEEE ASSP Magazine*, vol. 3, no. 1, pp. 4–16, Jan. 1986.

[16] K.-Y. Hsieh, Y.-C. Liu, P.-W. Wu, S.-W. Chang, and J. K. Lee, "Enabling streaming remoting on embedded dual-core processors," in *Proceedings of the 37th International Conference on Parallel Processing (ICPP'08)*, 2008, pp. 35–42.

[17] IEEE Standard, *IEEE Standard POSIX 1003.1c-1995 thread extensions*. IEEE, 1995, iSO/IEC 9945-1:1996.