AMOEBA-RT: Run-Time Verification of Adaptive Software *

Ji Zhang, Betty H.C. Cheng**, Heather J. Goldsby,
{zhangji9,chengb,hjg}@cse.msu.edu

Department of Computer Science and Engineering
Michigan State University, 3115 Engineering Building
East Lansing, Michigan 48824 USA

Abstract. Increasingly, software must dynamically adapt its behavior in response to changes in the supporting computing, communication infrastructure, and in the surrounding physical environment. Assurance that the adaptive software correctly satisfies its requirements is crucial if the software is to be used in high assurance systems, such as command and control or critical infrastructure protection systems. Adaptive software development for these systems must be grounded upon formalism and rigorous software engineering methodology to gain assurance. In this paper, we briefly describe AMOEBA-RT, a run-time monitoring and verification technique that provides assurance that dynamically adaptive software satisfies its requirements.

1 Introduction

Increasingly, software must adapt its behavior in response to changes in the supporting computing, communication infrastructure, and in the surrounding physical environment [1]. As such, a number of research projects have been investigating techniques to support dynamic adaptation [2–7]. Assurance that the adaptive software correctly satisfies its requirements is crucial if the software is to be used in high assurance systems, such as command and control or critical infrastructure protection systems. We previously introduced the Adapt-operator extended Linear Temporal Logic (A-LTL) [8] to formally specify adaptation properties for adaptive software. We consider adaptive software to be a system comprising a number of steady-state programs and adaptations among these steady-state programs. Specifically, a *steady-state program* is a non-adaptive program suited for a specific set of environmental conditions, and an *adaptation* is a transition from one steady-state program (the *source program*) to another steady-state program (the *target program*). For our approach, the developer specifies the adaptation properties, designs the steady-state programs and the adaptations among

* This work has been supported in part by NSF grants EIA-0000433, CNS-0551622, CCF-0541131, IIP-0700329, CCF-0750787, Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, Air Force Research Lab under subcontract MICH 06-S001-07-C1, Siemens Corporate Research, and a Quality Fund Program grant from Michigan State University.

** Please contact this author for all correspondences.

these steady-state programs, and then executes the adaptive system. In this paper, we describe AMOEBA-RT a run-time monitoring and verification technique to verify that dynamically adaptive software adheres to A-LTL and LTL properties.

Model checking is an attractive means to check for adherence to functional properties. Recent research efforts have demonstrated the use of static *model checking* to verify critical properties in adaptive software [7, 9]. Critical *adaptation properties* that need to be verified can be expressed in A-LTL and LTL. We previously developed the AMOEBA model checker [10] that modularly verifies A-LTL and LTL adaptation properties in adaptive software, thereby, significantly reducing the complexity of model checking of adaptive software. However, due to the state explosion problem, static model checking techniques alone are insufficient to provide assurance for complex adaptive programs. Run-time verification [11–14] is an attractive complement to static verification. Run-time verification monitors executions of a software system and uses a model checker to verify that the behavior of a software system adheres to a set of formal specifications, including temporal logic properties. Since only one execution path is examined at a time, the state explosion problem is effectively avoided in run-time model checking. Currently, to the best of our knowledge, there does not exist a run-time model checker that verifies adaptation properties specified in A-LTL and LTL.

In this paper, we introduce AMOEBA-RT, an A-LTL and LTL run-time model checker for adaptive software. In AMOEBA-RT, the run-time state information of an adaptive program is collected and then analyzed for adherence to the formal specifications. To that end, the adaptive software program is instrumented using an aspect-oriented approach [15] to collect run-time state information. As such, the aspect-oriented approach is non-invasive, meaning that the source code for the adaptive software is not directly altered. At run-time, the instrumented code sends the collected state information to a run-time model checking server that runs as a separate process. The run-time model checking server uses an automaton-based approach to determine whether the state information received from the adaptive program satisfies the adaptation properties specified in A-LTL and LTL.

AMOEBA-RT has been used to verify and detect execution errors in a number of adaptive components in wireless communication applications, including an adaptive Java pipeline program [10]. The remainder of the paper is organized as follows. Section 2 provides background information on the adapt-operator extended LTL, three commonly-used adaptation semantics, and the analysis of adaptation properties. In Section 3, we briefly introduces the AMOEBA-RT architecture. In Section 4, we illustrate the run-time verification using the adaptive Java pipeline example. Lastly, Section 5 summarizes the paper and discusses future work.

2 Specifying Adaptation Properties

This section describes the formal specification language used to specify adaptation properties, A-LTL, and illustrates how A-LTL can be used to specify commonly occurring adaptation semantics. AMOEBA-RT can then check for adherence to these adaptation properties at run-time.

To specify adaptation requirements, we have proposed A-LTL (Adapt operator-extended LTL) [8], an extension to LTL with the adapt operator ($\stackrel{\Omega}{\rightarrow}$). Informally, a software program satisfying "$\phi \stackrel{\Omega}{\rightarrow} \psi$" (read as ϕ adapts to ψ with adaptation constraint Ω) means that the program initially satisfies ϕ, and at a certain state A, it fulfills all the obligations demanded by ϕ and stops being constrained by ϕ, and in the next state B, starts to satisfy ψ, where ϕ and ψ are two temporal logic formulae. The state sequence (A, B) satisfies Ω, where Ω is an LTL formula evaluated on a sequence of two states. A given state's *obligations* are the necessary conditions that the state must satisfy for the program to satisfy its specification. Formal details of A-LTL may be found elsewhere [8].

In the following, we summarize three commonly occurring basic adaptation semantic interpretations from the literature [16–19] specified in terms of A-LTL. There are potentially many other adaptation semantics. In all three adaptation semantics, we denote the source and the target programs local properties as S_{SPEC} and T_{SPEC}, respectively. If applicable, the restriction condition during adaptation is R_{COND}. We use the term *fulfillment states* to refer to the states where all the obligations of the source program are fulfilled (i.e., S_{SPEC} is satisfied), thus making it safe to terminate the source behavior and ensuring that the system does not become inconsistent during adaptation.

One-Point Adaptation: After receiving an adaptation request A_{REQ}, the program adapts to the target program T_{SPEC} at a certain point during its execution. The prerequisite for one-point adaptation is that the source program S_{SPEC} should always eventually reach a fulfillment state during its execution.

$$(S_{SPEC} \wedge \Diamond A_{REQ}) \stackrel{\Omega}{\rightarrow} T_{SPEC}. \tag{1}$$

Formula 1 states that the program initially satisfies S_{SPEC}. After receiving an adaptation request, A_{REQ}, it waits until the program reaches a fulfillment state, i.e., all obligations generated by S_{SPEC} are satisfied. Then the program stops being obligated to satisfy S_{SPEC} and starts to satisfy T_{SPEC}. This semantics is straightforward and is explicitly or implicitly applied by most approaches (e.g., [16, 17, 19]) to deal with simple cases that do not require constraining the source behavior or overlapping the source and the target behavior.

Guided Adaptation: After receiving an adaptation request, the program first constrains its source program behavior by a restriction condition, R_{COND}, and then adapts to the target program when it reaches a fulfillment state.

$$\left(S_{SPEC} \wedge (\Diamond A_{REQ} \stackrel{\Omega_1}{\rightarrow} R_{COND})\right) \stackrel{\Omega_2}{\rightarrow} T_{SPEC}. \tag{2}$$

Formula 2 states that initially S_{SPEC} is satisfied. After an adaptation request, A_{REQ}, is received, the program should satisfy a restriction condition R_{COND} (marked with $\stackrel{\Omega_1}{\rightarrow}$). When the program reaches a fulfillment state of the source, the program stops being constrained by S_{SPEC}, and starts to satisfy T_{SPEC} (marked with $\stackrel{\Omega_2}{\rightarrow}$). The *hot-swapping* technique introduced by Appavoo *et al* [16] and the safe adaptation protocol [19] use the guided adaptation semantics.

Overlap Adaptation: The target program behavior starts before the source program behavior stops. During the overlap of the source and the target behavior, a restriction condition is applied to ensure that the source program reaches a fulfillment state.

$$\Big((S_{SPEC} \wedge (\Diamond A_{REQ} \xrightarrow{\Omega_1} R_{COND})) \xrightarrow{\Omega_2} true\Big)$$
$$\wedge \ \Big(\Diamond A_{REQ} \xrightarrow{\Omega_1} (T_{SPEC} \wedge (R_{COND} \xrightarrow{\Omega_2} true))\Big). \quad (3)$$

Formula 3 states that initially S_{SPEC} is satisfied. After an adaptation request, A_{REQ}, is received, the program should start to satisfy T_{SPEC} and also satisfy a restriction condition, R_{COND} (marked with $\xrightarrow{\Omega_1}$). When the program reaches a fulfillment state of the source program, the program stops being obliged by S_{SPEC} and R_{COND} (marked with $\xrightarrow{\Omega_2}$). The *graceful adaptation protocol* introduced by Chen *et al* [17] and the *distributed reset protocol* introduced by Kulkarni *et al* [18] use the overlap adaptation semantics.

3 Run-Time Model Checking

AMOEBA-RT extends the AMOEBA model checker [10] with support for run-time monitoring and run-time verification of requirements specified in A-LTL and LTL. AMOEBA-RT has two primary capabilities: First AMOEBA-RT uses an aspect-oriented technique to instrument and achieve run-time monitoring of the executing adaptive software. Second, AMOEBA-RT uses a run-time model checking server to support run-time verification of the A-LTL/LTL adaptation specifications. In the following, we provide additional details about each capability.

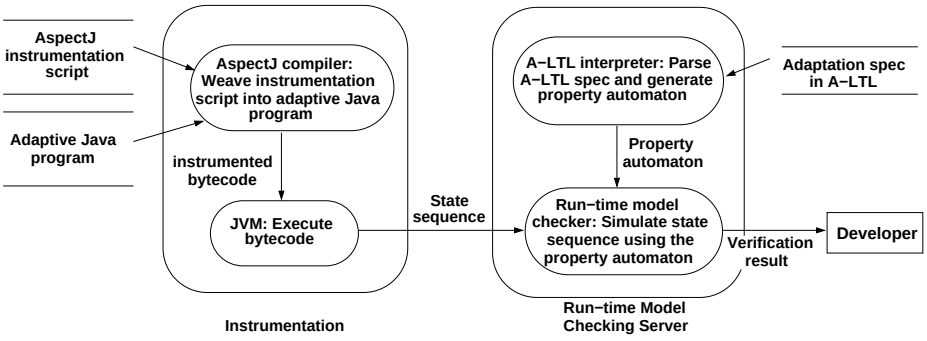

Fig. 1. The dataflow diagram for AMOEBA-RT verification

3.1 Run-time Monitoring

AMOEBA-RT instruments the adaptive system to achieve run-time monitoring. Figure 1 depicts the overall architecture of AMOEBA-RT. The instrumented code collects information about the run-time state of the adaptive software and transmits the information to the run-time model checking server. Using AspectJ [15], an aspect-oriented

extension to Java, our AMOEBA-RT instrumentation defines pointcuts around method calls that indicate a change in run-time state and uses advice to collect the run-time state information that is transmitted to the run-time model checking server.

Our approach is non-invasive in that the AspectJ compiler compiles the Java source files and an aspect file specifying the instrumentation, and then generates instrumented Java bytecode files. The Java bytecode files are then executed on a general JVM. During run-time, the instrumentation code collects run-time state information and sends the information to the run-time model checking server in sequence. When the adaptive program terminates, an end of execution message is attached to the end of the sequence and sent to the run-time model checking server.

3.2 Run-Time Analysis

As depicted in Figure 1, the AMOEBA-RT run-time model checking server checks the conformance of the sequence of state information received from the instrumented code with the adaptation requirements specified in A-LTL/LTL. An A-LTL interpreter processes the adaptation requirements, and outputs a property automaton, i.e., a finite state automaton that accepts the exact set of execution paths satisfying the specification.

AMOEBA-RT constructs a property automaton for the property being verified by extending the logic rewrite rules introduced by Bowman and Thompson [20]. In the property automaton, each node comprises two fields, 'p' and 'q,s' where p is a propositional logic formula indicating the condition satisfied by the node itself, and q is an A-LTL formula indicating the property that must be satisfied by its next states. The next nodes $t_1, t_2, \cdots, t_k$ of a node s are non-overlapping, i.e., the p values of these nodes are logically disjoint. Therefore, the property automata constructed are deterministic, i.e., we can always choose the appropriate next node based on the conditions in the current state. If a run-time execution path is accepted by the property automaton, then it satisfies the specification. In this way, the property automaton serves to verify a given execution sequence at run-time.

Formally, a *property automaton* is a tuple (S, S_0, T, P, N), where S is a set of states. S_0 is a set of initial states where $S_0 \subseteq S$.
$T : S \rightarrow 2^S$ maps each state to a set of next states.
$P : S \rightarrow proposition$ represents the propositional conditions that must be satisfied by each state. $N : S \rightarrow formula$ represents the conditions that must be satisfied by all the next states of a given state.

Given a set of A-LTL/LTL formula Φ, we generate a property automaton $PROP(\Phi)$ with the following features:

- For each member $\phi \in \Phi$, create an initial state $s \in S_0$ such that $P(s) = true$, $N(s) = \phi$.
- Let pe, p_i, and q_i be propositional formulae. For each state $s \in S$, let the partitioned normal form [20] of $N(s)$ be
  $(pe \wedge empty) \vee \bigvee_i (p_i \wedge \bigcirc qi)$, then it has a successor $s'_i \in S$ for each p_i field with $P(s'_i) = p_i$ and $N(s'_i) = q_i$.
  The $(pe \wedge empty)$ part of the partitioned normal form depicts the condition when a sequence is $empty$, where $empty \equiv \neg \bigcirc true$ [20], and pe is a proposition that

must be true when the state is the last state. In the $\bigvee_i(p_i \wedge \bigcirc q_i)$ part of the formula, the propositions p_i partitions $true$, and q_i is the corresponding condition that must hold when p_i holds in the current state.

A path of a property automaton is an infinite sequence of states $s_0, s_1, \cdots$ such that $s_0 \in S_0$, $s_n \in S$, and $s_i, s_{i+1} \in T$, for all i $(0 \leq i < n)$. We say a path of a property automaton $s_0, s_1, \cdots$, *simulates* an execution path of a program $s'_1, s'_2, \cdots$, if $P(s_i)$ *agrees with* s'_i for all i $(0 < i)$. We say a property automaton *accepts* an execution path from initial state $s \in S_0$, if there is a path in the property automaton starting from s that simulates the execution path. It can be proved [10] that the property automaton constructed above, from initial state $s \in S_0$, accepts exactly the set of executions that satisfy $N(s)$.[1] Thus, we are able to use the property automaton to verify that an execution path satisfies Φ.

Implementation We implemented this approach as the AMOEBA-RT prototype. Specifically, AMOEBA-RT uses the property automaton to simulate the sequence of run-time state information received from the instrumentation module in parallel with the adaptive software.

If the property automaton returns *failure* during or at the end of an execution, then the execution violates the A-LTL property and the state sequence (i.e., a counter-example) is recorded in a bug report. Otherwise, the model checking server returns *success*. If an execution violates the A-LTL property, then there are two possibilities. First, if the execution represents a valid behavior of the system, then the A-LTL property violated by the execution needs to be modified. Second, in cases where the system behavior is erroneous, the developer must modify the system to adhere to the A-LTL property.

4 An Illustrative Example

In some multi-threaded Java programs, such as proxy servers, data are processed and transmitted from one thread to another in a pipelined fashion. The Java pipeline is implemented using a pair of piped I/O classes, which can be **synchronous** or **asynchronous** functions. The asynchronized version is preferable when CPU load is low [23]. However, when the CPU load is high, the synchronized version performs better. The data transmission is achieved by accessing shared buffers. A sync buffer and an async buffer are used for the synchronized and asynchronous pipeline components, respectively. Previously, we have constructed an adaptive version of the Java pipeline classes where the system can monitor CPU workload and use an adaptation decision maker to select the optimal implementation for specific run-time conditions.

We specify the adaptation requirements for the adaptive Java pipeline program in A-LTL as follows. As such, before adaptation, the system (i.e., the source program) is required to input data from the synchronized pipeline in response to the outputs. That

[1] We ignore the eventuality constraint [21] (a.k.a self-fulfillment [22]) at this point. However, later steps will ensure eventuality to hold in our approach.

is, for each output data x in the synchronized mode, the system must eventually input data x. In LTL:

$$\Box(SyncOutput(x) \rightarrow \Diamond SyncInput(x)). \tag{4}$$

The program behavior after adaptation can be specified in a similar manner. The system (i.e., the target program) is required to input data from the asynchronous pipeline in response to the outputs. In LTL:

$$\Box(AsyncOutput(x) \rightarrow \Diamond AsyncInput(x)). \tag{5}$$

For both the synchronized and asynchronous pipelines, when an output event occurs, an input obligation is generated. In other words, if the output is generated, then there should be a subsequent input event to read the generated output, thus discharging the input obligation. Formulae (4) and (5) state that an execution must fulfill all input obligations before it terminates.

In the adaptation from the source program to the target program, we allow the write operation of the asynchronous pipeline to overlap with the read operation of the synchronized pipeline. Therefore, we apply the overlap adaptation semantics introduced in Section 2 to the specification of the adaptation. During the overlapped period, the restriction condition is that the synchronized pipeline should not output data, and the asynchronous pipeline should not input data. The requirement for the adaptation from the source to the target can be specified using the overlap adaptation semantics as follows:

$$\begin{aligned}
&(((\Box(SyncOutput \rightarrow \Diamond SyncInput) \wedge (\Diamond A_{REQ} \\
&\qquad \xrightarrow{\Omega} \Box \neg SyncOutput)) \\
&\qquad\quad \xrightarrow{\Omega} true) \\
&\qquad\qquad \wedge (\Diamond A_{REQ} \\
&\xrightarrow{\Omega} (\Box(AsyncOutput \rightarrow \Diamond AsyncInputs) \wedge (\Box \neg AsyncInput \\
&\qquad\quad \xrightarrow{\Omega} true)))).
\end{aligned} \tag{6}$$

This formula states that the system should adapt from the source program (in the synchronized mode) to the target program (in the asynchronous mode) in response to the adaptation request A_{REQ}. The source and target programs overlap. During the overlapped period, the source must not output data, and the target must not input data. The output obligation generated in the synchronized mode must be fulfilled before the adaptation completes.

4.1 Instrumentation and Model Checking

To monitor the run-time execution conditions of the adaptive Java pipeline program, we use aspect-oriented programming to insert instrumentation code into the adaptive system. Currently, the AspectJ script for instrumentation is generated manually; future work will explore automated support.

In this example, the "instrumentation concern" is encapsulated in an `Instrumentation` aspect, saved in a file named `Instrumentation.aj`. Specifically, we define a pointcut `Main` to identify the `main()` method of the adaptive Java pipeline program. Figure 2 depicts the *before advice* we defined for the `Main` pointcut. Specifically, at the very beginning of the entire program, we insert code to initialize the property automaton in the run-time model checking server by sending an A-LTL formula to the server. The `AmoebaChecker` class implements a stub that is responsible for the communication with the model checking server. Its constructor method takes three parameters: The first two parameters specify the IP address and the port number for the model checking server, respectively. The third parameter specifies the A-LTL property to be verified. Figure 3 depicts the `after` advice we defined for the `Main` pointcut. Specifically, at the very end of each execution, we insert code to send an `''EOE''` message to the run-time model checking server to terminate the model checking.

```
1  before() :Main() {
2     AmoebaChecker.checker = new AmoebaChecker ("192.168.1.101", 2211,
3       "((([](SyncOutput->SyncInput) /\\ (<>AREQ _>[] !SyncOutput)) _>true)
4       /\\ (<>AREQ _> ([] (AsyncOutput-> <> AsyncInput) /\\
5       ([]!AsyncInput _> true ))))" );  }
```

Fig. 2. Before advice for `Main` pointcut

```
1  after() :Main { AmoebaChecker.checker.terminate(); }
```

Fig. 3. After advice for `Main` pointcut

Second, we use pointcuts to identify the locations of the adaptive Java pipeline program at which the sync shared buffer and the async shared buffer are accessed and therefore should be instrumented. Figure 4 illustrates the pointcut definition for the `SyncOutput` message. Line 1 defines that the point is within the `receive()` method of the async piped input and sync piped input classes. Line 2 defines that the point is at the location where the `buffer` is accessed. When the buffers are accessed for read/write, an input/output message will be generated and sent to the run-time model checking server through network communication. Figure 5 shows the advice definition for `SyncOutput`. This code defines that it is a *before advice* for the `sync_output` pointcut. Before each access to the sync buffer, the advice inserts instrumentation code that invokes the `nextState()` method of the `checker`, which sends the `SyncOutput` message to the run-time model checking server.

We executed the instrumented adaptive Java program and verified the program against the overlap adaptation requirement in Formula (6) using AMOEBA-RT. To demonstrate that the model checker is actually effective in catching errors, in a second experiment, we deliberately introduced some errors in the adaptive system. This time,

```
1  public pointcut sync_output() : withincode (*sync.PipedInputStream.receive(..))
2  && get (byte[] buffer );
```

Fig. 4. Pointcut definition for `SyncOutput`

```
1  before() : sync_output() { checker.nextState ("SyncOutput"); };
```

Fig. 5. Before advice for `SyncOutput` pointcut

AMOEBA-RT caught violations of the property in some of the random executions. As a response to the violations, AMOEBA-RT recorded the execution paths in a bug report that is currently processed offline. The bug reports documents counter-examples, that is, the paths of execution that lead to a property violation. The bug report in the above experiment showed that during those execution paths, the property was indeed violated.

5 Conclusions

In this paper, we introduced AMOEBA-RT, a run-time verification approach for adaptive software. AMOEBA-RT comprises instrumentation that supports run-time monitoring and a run-time model checker that supports verification. To instrument the adaptive system, we used a non-invasive aspect-oriented technique that separates the run-time monitoring and model checking concern from the functional logic of the adaptive system. The AMOEBA-RT model checking server interprets A-LTL specifications and verifies execution sequences against A-LTL specifications at run-time.

There are numerous possible directions for future work, including applying AMOEBA-RT to additional case studies. Also, we are investigating the use of counter-examples generated by AMOEBA-RT as input to the decision-making process for adaptation. Ideally, the adaptive system would be able to detect property violations and then adapt to repair itself. Additionally, we are interested in enabling a developer to visualize the run-time execution path of the adaptive system on the design models [24]. We envision that this capability could be used to better understand the relationship between environmental conditions and adaptation, as well as the need for additional steady-state systems.

References

1. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. IEEE Computer **37**(7) (2004) 56–64
2. Métayer, D.L.: Software architecture styles as graph grammars. In: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering, ACM Press (1996) 15–23
3. Taentzer, G., Goedicke, M., Meyer, T.: Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations, Springer-Verlag (2000) 179–193
4. Hirsch, D., Inverardi, P., Montanari, U.: Graph grammars and constraint solving for software architecture styles. In: Proceedings of the third international workshop on Software architecture, ACM Press (1998) 69–72

5. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proceedings of the 20th International Conference on Software Engineering, IEEE Computer Society (1998) 177–186
6. Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, Jr., E.J., Robbins, J.E.: A component- and message-based architectural style for GUI software. In: Proceedings of the 17th International Conference on Software Engineering, ACM Press (1995) 295–304
7. Kramer, J., Magee, J.: Analysing dynamic change in software architectures: a case study. In: Proc. of 4th IEEE International Conference on Configurable Distributed Systems, Annapolis (1998)
8. Zhang, J., Cheng, B.H.C.: Using temporal logic to specify adaptive program semantics. Journal of Systems and Software (JSS), Architecting Dependable Systems **79**(10) (2006) 1361–1369
9. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Proceedings of International Conference on Software Engineering (ICSE'06), Shanghai,China (2006)
10. Zhang, J., Cheng, B.H.C.: Modular model checking of dynamically adaptive programs. Technical Report MSU-CSE-06-18, Computer Science and Engineering, Michigan State University, East Lansing, Michigan (2006) `http://www.cse.msu.edu/~zhangji9/Zhang06Modular.pdf`.
11. Havelund, K., Rosu, G.: Monitoring Java programs with Java PathExplorer. In: Proceedings of the 1st Workshop on Runtime Verification, Paris, France (2001)
12. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: Proc. Parallel and Distributed Processing Techniques and Applications. (1999) 279–287
13. Drusinsky, D.: The temporal rover and the atg rover. In: Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification, London, UK, Springer-Verlag (2000) 323–330
14. Feather, M.S., Fickas, S., Van Lamsweerde, A., Ponsard, C.: Reconciling system requirements and runtime behavior. In: Proceedings of the 9th International Workshop on Software Specification and Design, IEEE Computer Society (1998) 50
15. The AspectJ Team: The AspectJ(TM) programming guide (2007) http://eclipse.org/aspectj.
16. Appavoo, J., Hui, K., Soules, C.A.N., et al.: Enabling autonomic behavior in systems software with hot swapping. IBM Systems Journal **42**(1) (2003) 60
17. Chen, W.K., Hiltunen, M.A., Schlichting, R.D.: Constructing adaptive software in distributed systems. In: Proc. of the 21st International Conference on Distributed Computing Systems, Mesa, AZ (2001)
18. Kulkarni, S.S., Biyani, K.N., Arumugam, U.: Composing distributed fault-tolerance components. In: Proccedings of the International Conference on Dependable Systems and Networks (DSN), Supplemental Volume, Workshop on Principles of Dependable Systems. (2003) W127–W136
19. Zhang, J., Yang, Z., Cheng, B.H.C., McKinley, P.K.: Adding safeness to dynamic adaptation techniques. In: Proceedings of ICSE 2004 Workshop on Architecting Dependable Systems, Edinburgh, Scotland, UK (2004)
20. Bowman, H., Thompson, S.J.: A tableaux method for Interval Temporal Logic with projection. In: TABLEAUX'98, International Conference on Analytic Tableaux and Related Methods. Number 1397 in Lecture Notes in AI, Springer-Verlag (1998) 108–123
21. M.Y.Vardi, Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of the 1st Symposium on Logic in Computer Science, Cambridge, England (1986) 322–331
22. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press (1985) 97–107
23. Zhang, J., Lee, J., McKinley, P.K.: Optimizing the Java pipe I/O stream library for performance. In: Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Published as Lecture Notes in Computer Science (LNCS), Vol. 2481, Springer-Verlag, College Park, Maryland, USA (2002)
24. Goldsby, H., Cheng, B.H.C., Konrad, S., Kamdoum, S.: A visualization framework for the modeling and formal analysis of high assurance systems. In: Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Genova, Italy (2006)