Formal Modeling of Communication Traces

Sergiy Boroday[1], Hesham Hallal[1], Alexandre Petrenko[1], Andreas Ulrich[2]

1 CRIM
550 Sherbrooke West, Suite 100
Montreal, H3A 1B9, Canada
{Boroday, Hallal, Petrenko}@crim.ca

2 Siemens AG
Corporate Technology CT SE1, Otto-Hahn-Ring 6
81730 München, Germany
andreas.ulrich@siemens.com

Abstract: Development of distributed systems is complicated by the absence of reliable global timing, concurrency, and nondeterminism. To deal with these obstacles log files are produced by an instrumented system facilitating analysis, testing, and debugging. This paper presents a formal framework for the analysis of distributed system logs based on event trace concept. A partially ordered trace of events executed by a distributed system is modeled by a collection of communicating automata. We present an implementation of the analysis approach in SDL based on ObjectGEODE. A formalization of a property of an event trace, being a replica of another trace, is discussed.

1. Introduction

Concurrency and distribution have emerged as viable options for the design of complex systems. Although asynchrony and geographic distribution add to the value of distributed systems; the same characteristics render especially their validation, testing and maintenance difficult. Meanwhile, the possibility of using formal methods as a means for the development and validation of distributed systems restores the hope that both time-to-market and validation cost could be slashed. Once a formal specification of the behavior of a system becomes available, many development activities, such as code generation and test derivation could easily be automated. However, the development of distributed systems rarely yields formal specifications of their behavior that would make formal methods fully applicable. Recognizing this fact, research in both academia and industry aims at developing tools for debugging and testing that do not require formal specifications in the first place. Such tools usually rely on monitoring functions and produce log files of execution traces that can be analyzed further. This type of analysis is also known as passive testing.

The analysis of distributed system logs is also performed routinely by system administrators. However, effectiveness of manual analysis is low since patterns of interactions could be masked due to a concurrent and distributed character of the system. Thus, the development of tools and methods for automating the process of analyzing log files is important to diagnose and troubleshoot problems in distributed systems and networks faster and by lesser cost.

A general approach to trace analysis taken in this research is outlined as follows. The distributed system is instrumented in a way that events executed by the distributed processes are collected. Such events typically denote the send and receive of messages, local actions, and others. A trace is produced that includes all the events collected during a system run, and an appropriate analysis tool could be employed to check the trace against some user-defined properties. There exists a large body of work on developing various tools to visualize traces, see e.g. [Bl93, LF98, Wa00]. Their goal is to facilitate efforts of the designer or tester for locating and correcting bugs by filtering out unrelated information and by offering a proper visualization of traces. The analysis is performed manually either online (simultaneously with the system execution) or post mortem. Another group of methods targets the analysis phase by offering means to verify certain properties in the distributed system under test (SUT), see, e.g. [DC94, Fr94, Ja94].

Developing a tool for checking properties in execution traces, one faces the choice of either elaborating algorithms for a specific class of properties and implementing them in a specialized tool or relying on a general-purpose model checker. In the first scenario, the daunting task is to implement the algorithms from scratch. On the other hand, reuse of an off-the-shelf model checker allows the tool developers to count on reliable, highly sophisticated, and versatile products, in which many years of research and development have already been invested.

In this paper, we present an automata-based approach to model traces collected during the execution of a distributed system. Our approach relies on a partially ordered set (poset) of events, where the partial order is the traditional *happened-before* relation [CL85], and its related lattice, known as the lattice of ideals, consistent cuts or, simply, global states. Intuitively, such a lattice represents the joint behavior of the distributed processes observed in a SUT. We propose to model the processes of the SUT using finite automata. Then, we prove that the composition of these automata is isomorphic to the lattice of ideals of the poset, which allows us to translate the problem of property testing in event traces to a typical model checking problem. Our framework for modeling and analysis of traces treats both synchronous and asynchronous communications simultaneously.

From the theoretical perspective, our framework can be considered as a generalization of the framework proposed in [Ch95], where event causality is studied in traces with synchronous or asynchronous communications. While the authors mention the possibility of dealing with traces in both communication modes, they do not provide a formal treatment of such traces. The communication of threads running on a single machine is naturally modeled by rendezvous, while the interaction of threads running on machines that are geographically distributed is essentially asynchronous message passing. Moreover, rendezvous could be used as an abstraction of some message exchange patterns. This explains the mixed nature of the framework developed in this paper. Instead of modeling synchronous communication by a pair of events, as is done in several previous works, we use just a single abstract rendezvous event. This allows us to treat an event trace simply as a partially ordered set of events, so existing techniques based on partial orders remain applicable. Notice also that [Ch95] as well as [En02] mostly concentrate on studying the hierarchy of various classes of traces, while we elaborate on modeling partial orders using automata for the purpose of analysis. Compared to [En02], as well as to other papers on the semantics and model checking of message sequence charts (MSC) [AY99, LL97], our framework includes local events and rendezvous that are ignored there. In [LL95], a global state graph (Büchi automaton) is built from an MSC similar to [Fr94] and [Ja94], which consider the ideal lattice. The paper [Ja94] describes an approach for trace analysis based on building the ideal lattice from a trace and demonstrates that property verification could be performed while the lattice is built. The class of properties is restricted to those allowing automata representation (e.g., LTL), while the use of a general model checker advocated in this paper allows us to consider more complex properties. Our early attempt to formalize distributed trace modeling and analysis is presented in [He03]. This work presents a revised and generalized definition of the event trace along with other contributions.

This paper is organized as follows. In Section 2, we formally define event traces that include send, receive and local events, as well as rendezvous. In Section 3, we detail our approach of modeling distributed processes by means of automata. Section 4 explains an implementation of our approach based on the commercial tool ObjectGEODE. Then, in Section 5, we discuss a specific property of trace, related to the problem of trace replay.

2. Event Traces

A distributed system consists of sequential processes P_1, …, P_n that perform local actions and communicate by exchanging messages and by performing rendezvous (synchronization points). In our framework, a process reports its own actions by generating a finite non-empty set E_i of events that are classified into mutually exclusive sets of *local* (l), *send* (s), *receive* (r), and *rendezvous* (z) events, that are the only events observable outside the system. A local event indicates some change in the state of a process. Let *match* (r) be the send-receive *matching* function, which for the receive r of each message yields the corresponding send of the same message. The use of such a function (and not an arbitrary binary relation) is motivated by the fact that each message is sent only once. If a message is repeated we consider it as a completely new message, so we could rely on the well-established poset theory rather than on the theory of pomsets. Clearly, there is no "hanging" unmatched receive events if and only if the matching function is completely-defined, the absence of "hanging" sends corresponds to a surjective function, and the case where multi-casting is not involved corresponds to an injective matching function. The inversion of the matching function $match^{-1}$ consists of all pairs (s, r) such that $s = match(r)$. It is called the send-receive *precedence* relation and denoted $\prec_p$. The send-receive precedence relation is a strict partial order. Rendezvous events are the only common events between processes; in other words, any non-empty set $E_i \cap E_j$, where $i \neq j$, contains only rendezvous events.

Each event occurs in a process P_i only once; the events of a process P_i constitute a couple $T_i = (E_i, \prec_i)$ called the *local trace* of P_i. In the simplest case, process P_i is sequential, so the events in T_i are totally ordered and each $\prec_i$ is an irreflexive total order. However, in this work, we do not restrict ourselves to sequential processes. We allow each process P_i to be concurrent, and $\prec_i$ to be a partial order. Some events of the same process could be concurrent because a log tool reports them as such or their execution order is not important for some reasons. Thus, the definition of event trace in [He03] is generalized here to account for the fact that local orders are not necessarily total orders.

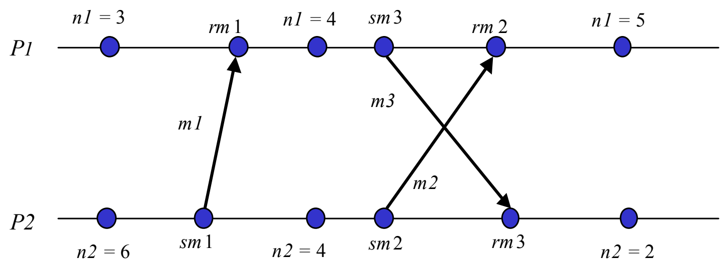


Fig. 1: A trace diagram

The *happened-before* relation $\prec$ on the set of events E of the system $P_1, \ldots, P_n$, $E = E_1 \cup \ldots \cup E_n$, with send-receive matching function *match* is a transitive closure of the $\prec_p \cup \prec_1 \cup \ldots \cup \prec_n$ (that is the smallest transitive relation which contains local orders and send-receive precedence relation).

A couple consisting of the set of events and the happen-before relation (E, $\prec$) is said to be an *event trace* if the happened-before relation $\prec$ on the set of events E is irreflexive (no event happens before itself). Since the happened-before relation is transitive by definition, an event trace is defined when the relation on E is a strict partial order.

A diagram of an event trace is shown in Fig. 1. Unconnected bullets correspond to local events, and bullets connected by edges to receive events matching send events for messages $m1$, $m2$, and $m3$.

Let Σ be a set, and $<$ be a strict partial order on this set. A set $P \subseteq \Sigma$ is called an *ideal* if $e \in P$ and $e' < e$ implies $e' \in P$. Let $I(\Sigma, <)$ denote the set of all the ideals for pair (Σ, $<$). It is known that $I(\Sigma, <)$ and a set inclusion relation form a lattice, which we denote by $(I(\Sigma, <), \subseteq)$ and call it the *ideal* lattice.

The ideal lattice of an event trace (also called the lattice of consistent cuts or global states) represents all the possible interleavings of events and provides a simple way to check properties of concurrent systems [Ja94].

Here, the question arises how to build the ideal lattice of an event trace. Several methods have already been discussed, see, e.g., [Ja94]. However, the practicality of such methods can be viewed only in light of satisfying the goal of trace analysis: verifying properties based on an event trace. The work in [Ja94] proposes an efficient method to build the ideal lattice and indicates the possibility of using a standard model checker to verify properties on the lattice directly, a task that any well known tool, e.g., SPIN [Ho97], SMV [SMV], or ObjectGEODE [Te02], could solve. The use of the obtained lattice as an input to a model checker faces a scalability problem since the model checker could simply reject a specification whose size exceeds its input capacity. We believe that a more efficient solution to the problem lies in applying a model checker to a modular specification instead of a monolith one. Indeed, modular specifications are more compact and allow a model checker to fully exploit sophisticated search techniques avoiding a full state space search inevitable in case of monolith specifications.

3. Modeling Event Traces Using Automata

In this section, we describe how a given event trace is modeled by a system of communicating automata and demonstrate that the system exactly characterizes the happened-before relation of the event trace, i.e., the composition of the automata and the lattice of ideals are isomorphic. We first recall a few definitions from automata theory.

An *automaton* A is a tuple $<\Sigma, Q, q_0, \rightarrow_A, Q_f>$, where Σ is a finite set of actions, Q represents a finite set of states, q_0 is the initial state; $\rightarrow_A \subseteq Q \times \Sigma \times Q$ is a transition relation, and $Q_f \subseteq Q$ is a set of final states. We use the operator $||$ to compose automata. Given $A_1 = <E_1, Q_1, q_{01}, \rightarrow_{A1}, Q_{f1}>$ and $A_2 = <E_2, Q_2, q_{02}, \rightarrow_{A2}, Q_{f2}>$, the *composition automaton*, denoted $A_1 \,||\, A_2$, is a tuple $< E, Q, q_0, \rightarrow, Q_f>$, where $E = E_1 \cup E_2$, $q_0 = (q_{01}, q_{02})$; $Q \subseteq Q_1 \times Q_2$, $\rightarrow$, and Q_f are the smallest sets obtained by applying the following rules.

- If q_1 -$a$$\rightarrow_1$ q'_1 and $a \notin E_2$ then (q_1, q_2) -$a$$\rightarrow$ (q'_1, q_2).
- If q_2 -$a$$\rightarrow_2$ q'_2 and $a \notin E_1$ then (q_1, q_2) -$a$$\rightarrow$ (q_1, q'_2).
- If q_1 -$a$$\rightarrow_1$ q'_1 and q_2 -$a$$\rightarrow_2$ q'_2 then (q_1, q_2) -$a$$\rightarrow$ (q'_1, q'_2).
- $Q_f = Q \cap (Q_{f1} \times Q_{f2})$.

The composition is associative; it can be applied to finitely many automata. The composition of n automata $C_1 \dots C_n$ is an automaton $C = C_1 \,||\, \dots \,||\, C_n$ over the alphabet $E = E_1 \cup \dots \cup E_n$.

3.1 Ideal Lattice Automaton

An ideal lattice is usually visualized as a graph, called the covering digraph [Ja94], in which nodes represent the elements of the poset and edges represent the relations between the elements (with the omission of transitive edges). We represent the ideal lattice by an automaton, called the *ideal lattice automaton*, which corresponds to the covering digraph and is defined as follows.

Given an set Σ along with a partial order $<$ on Σ, the *ideal lattice automaton* of Σ and $<$ is a tuple $<\Sigma, I(\Sigma, <), \varnothing, \rightarrow_T, \{\Sigma\}>$, where Σ is the set of actions, $I(E, <)$ is the set of states and $\varnothing$ is the initial state, $\rightarrow_T = \{(P, a, R) \mid P, R \in I(\Sigma, <), R = P \cup \{a\}$, and $R \neq P\}$, $\{\Sigma\}$ is the set of final states.

Each word accepted by the ideal lattice automaton of the event trace contains exactly one instance of an event in E and thus defines a total order that is a linearization of the happened-before relation $\prec$, i.e., the word is one possible interleaving of the events in the trace. The accepted language of the ideal automaton consists exactly of all linearizations of the strict partial order $\prec$ [Ja94, BP82].

The states of the ideal lattice automaton of an event trace represent all possible states of the system that generated the event trace; thus we can use the ideal lattice automaton to check properties of the system exhibited during its execution. The ideal lattice automaton is minimal.

3.2 Composition of Automata

Given an event trace $(E, \prec)$, we define a system of communicating automata, where n ideal lattice automata of local traces model the processes $P_1, \ldots, P_n$ that accept the corresponding local traces $T_1, \ldots, T_n$, respectively, and automata that model message delays inherent to asynchronous communication.

Each message delay automaton is an ideal lattice automaton of two totally ordered events that reflects the precedence of a send event over the matching receive. Let C be a composition of automata that are the ideal lattice automata of local traces and message delay automata.

Theorem 1. The composition automaton C and the ideal lattice automaton T are isomorphic.

The proof is omitted due the lack of space. It is based on the fact that if the transitive closure of a union of posets is a poset itself, then the lattice automaton of the closure is isomorphic to the composition of the lattice automata of posets. The proof of Theorem 1 for the case of the totally ordered local traces is presented in [He03].

The theorem shows that the composition of the automata representing local traces and the ideal lattice automaton of the distributed system can be used interchangeably to reason about the system. In particular, model checking technology can be applied on the composition of automata rather than using the monolithic ideal lattice automaton to check properties on the collected traces of distributed systems more efficiently.

4. Using SDL Tools in Trace Analysis

To implement our approach we use SDL (Specification and Description Language). SDL is formal, visual, and based on an extended automata (EFSM) language that is standardized by the ITU [Z100]. It is widely used in the telecommunication area but gains also popularity in others safety critical fields such as automotive, aerospace, and medical software.

Use of an extended automata language to model simple automata may seem unnecessary. However, SDL treats different message-related data and allows dealing with variables, predicates, and elements of programming to model local partial orders in a natural and compact way. Among available SDL model checkers, we chose ObjectGeode as it provides a powerful extended automata based property specification language GOAL, rendezvous extensions to standard SDL, and, even more interestingly, it supports individual message channels that allow one to model the message delay automata. However, to ensure compatibility with other SDL model checkers, we define our system of communicating automata based on the standard SDL [He99] as suggested in [MZh99]. While not all features described in theory are implemented yet, e.g., mixed communications, our tool supports already many features not discussed above, such as basic process and event abstractions and a property specification interface based on a repository of a commonly used property specification repository [Dw99]. For detailed analysis of the trace, not only property satisfaction, but also existence of linearizations on which the property holds is checked (property satisfiability). Such linearizations, as well as counter-example linearizations, on which the property does not hold, if detected, could be generated as ObjectGeode scenarios and visualized.

Previously, we reported case studies of the trace-based analysis of a track controller [Ul03] and a sliding window protocol implementation [He03]. The block diagram and a snapshot of the tool are shown in Fig. 2 and Fig. 3, respectively. A snapshot of ObjectGeode tool is shown in Fig 4.

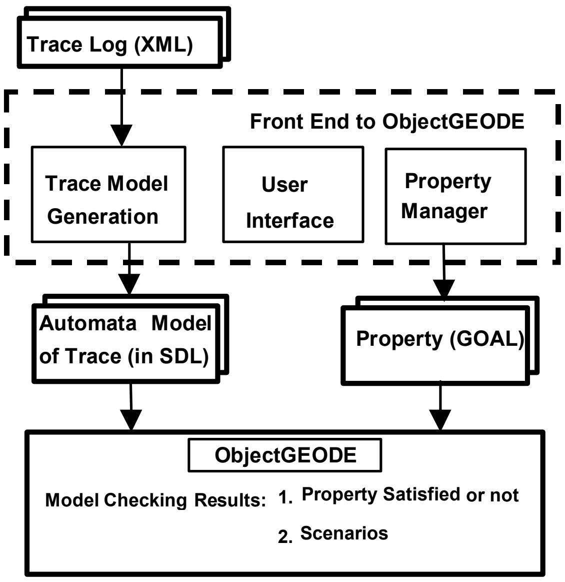
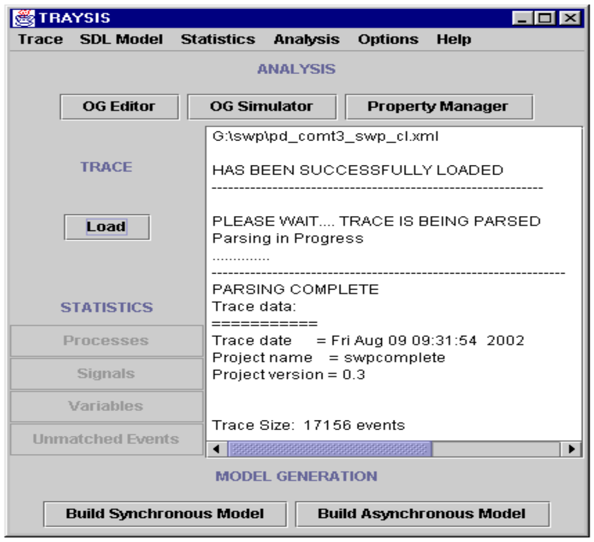


Fig. 2: The block diagram of the method



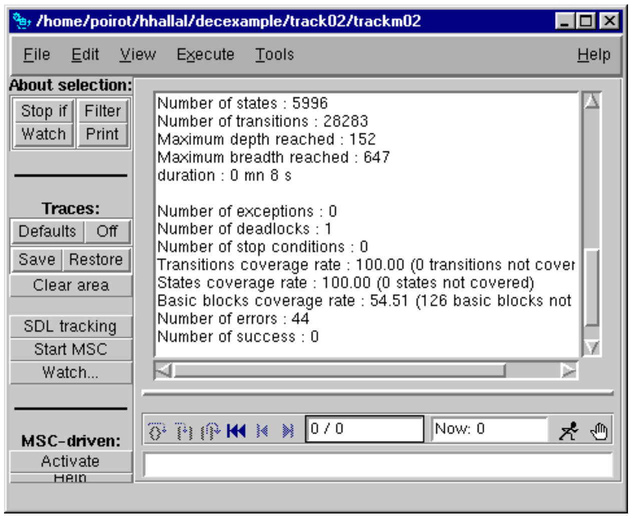
```
G:\swp\pd_comt3_swp_cl.xml

HAS BEEN SUCCESSFULLY LOADED
------------------------------------------------------------

PLEASE WAIT.... TRACE IS BEING PARSED
Parsing in Progress
..............
-----------------------------------------------------------------
PARSING COMPLETE
Trace data:
===========
Trace date      = Fri Aug 09 09:31:54  2002
Project name    = swpcomplete
Project version = 0.3


Trace Size:  17156 events
```

Fig. 3: The TRAYSIS tool



```
Number of states : 5996
Number of transitions : 28283
Maximum depth reached : 152
Maximum breadth reached : 647
duration : 0 mn 8 s

Number of exceptions : 0
Number of deadlocks : 1
Number of stop conditions : 0
Transitions coverage rate : 100.00 (0 transitions not cover
States coverage rate : 100.00 (0 states not covered)
Basic blocks coverage rate : 54.51 (126 basic blocks not
Number of errors : 44
Number of success : 0
```

Fig. 4: Model checking results

5. Replay of Event Traces

The problem of replay or reproducing traces of distributed systems is important for regression testing, performance evaluation, especially after an engineering change of the system under consideration, for test harness generation for a given part of the system from recorded event traces, as well as for debugging and fault location. It is acknowledged [Di96] that replay could only be achieved on a certain level of abstraction, however, the problem of determining an appropriate abstraction level has not yet been addressed in a formal way. In this paper, we study the conditions under which one trace could be considered as a replica of another trace with a different happened before relation. In other words, we attempt to define an abstraction level that differs from the mere equality of the happened-before relations.

Most works on replay simply provide platform specific solutions, such as modifications of the system or development of special environments for trace replay. The replay tools [Di96] attempt to deliver exact reproduction of a trace. Since this is usually impossible for realistic traces and systems; some of the tools provide a mechanism to report a failure of the replay, others do not. We believe that the problem needs a more formal setting. Specifically, a formal definition of which trace replays another trace is required. Such a definition should be weaker than mere trace equality.

Consider an event trace extracted from a log file consisting of Send and Receive events produced by an instrumented distributed system. Local events of communicating processes are usually abstracted in the context of replaying event traces. Let each event have a timestamp in the log file and other parameters indicating the type, source and destination of the transmitted message among other properties. Each local order in the event trace that is determined from timestamps of events is a total order.

The definition of event traces in Section 2 implies that a given log file *reproduces* a given event trace if it contains a replica of the event trace. It is intuitively clear that uncontrollable behavior of the distributed system, which once yielded a given event trace, may prevent the system from reproducing the same trace next time the system is executed. Therefore, the question arises whether the replication condition could be relaxed to take in account specific types of uncontrollable behavior.

Uncontrollable behavior could sometimes be caused by inherent concurrency of actions taken by a constituent process alone. Each process is only instrumented to generate Send and Receive events, nothing can be deduced about its internal structure or behavior. It is just a black box whose external behavior is reflected in the given trace, but the process may run in fact on several processors enabling concurrent execution of actions. A process might possess also several communication ports where events occur independently. Two messages received within a short time interval might not necessarily be processed in the order received. Events emitted by a process are time-stamped. It is reasonable to restrict the analysis to the case when the instrumented processes always serialize even concurrent events. This implies that the difference of timestamps of two consecutive events emitted by a process may sometimes be within a margin of errors of a time-stamping mechanism. No definite conclusion could be made about the actual order of the corresponding actions executed by the process. When the given trace is to be replayed such actions may happen in any order. Hence, each local order is replaced by a partial order based on a given error margin for timestamps.

This allows us to state a weaker definition of replay relation on traces, namely, that two traces should not contradict to each other, rather than exactly reproduce each other.

Given two event traces over the same event set E with the happened-before relations $\prec$ and $\prec'$, we say that one event trace *reproduces* another one if $\prec \cup \prec'$ is irreflexive.

It is obvious that two event traces reproduce each other if and only if they have a common interleaving of events (linearization).

The replay relation is symmetric but not transitive. If we further relax the requirement for traces to be defined on the same set of events, the resulting definition of the replay property could fit the needs of regression testing, where certain new events corresponding to added functionality could be ignored.

When the replay property of traces is formally defined, the problem of enforcing replay reduces to identification of a minimal set of causal dependencies that should be imposed on the system (by modification, process wrapping, etc) to guarantee trace reproduction. This constitutes our current research.

6. Conclusion

We presented a generalized formal framework for modeling event traces of distributed systems that starts from a log file collected during execution of the system. The model of an event trace consists of a collection of communicating automata that allows one to reason about properties of the system using standard model checking techniques.

Our future work will concern the development of a library of pre-defined property templates, as well as methods and tools for trace replay.

Bibliography

[Al95] Algayres, B. et.al.: GOAL: Observing SDL behaviors with GEODE. In (Braek, R.; Sarma, A.): Proc. 7th SDL Forum, Oslo, Norway, 1995. Elsevier Science Publishers B. V. (North Holland), Amsterdam, 1995; pp. 359–372.

[AY99] Alur, R.; Yannakakis, M.: Model Checking of Message Sequence Charts. In (J.C.M. Baeten, S. Mauw): Proc. 10th International Conference on Concurrency Theory (CONCUR'99), Eindhoven, the Netherlands, 1999. LNCS 1664. Springer, 1999; pp. 114–129.

[Bl93] Black, J.P. et.al.: Linking Specifications, Abstraction, and Debugging, CCNG Technical Report E-232, Computer Communications and Network Group, University of Waterloo, Nov. 1993.

[BP82] Bonnet, R.; Pouzet, M.: Linear Extensions of Ordered Sets. In (Rival, I.): Ordered Sets. D. Reidel Publishing Co, Dordrecht, the Netherlands, 1982; pp. 125–170.

[CL85] Chandy, K.; Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computing Systems, 3(1). 1985. ACM Press, New York, USA; pp. 63–75.

[Ch95] Charron-Bost, B. et.al.: Synchronous, Asynchronous, and Causally Ordered Communications. Distributed Computing, 9(4). Springer, 1996; pp. 173–191.

[Di96] Dionne, C. et.al.: A Taxonomy of Distributed Debuggers Based on Execution Replay. In (Arabnia, H.R.): Proc. 2nd Int. Conf. on Parallel and Distributed Processing Techniques and Applications, Sunnyvale, California, 1996. CSREA Press, Las Vegas, USA; pp. 203–214.

[Dw99] Dwyer, M. et.al.: Patterns in Property Specifications for Finite-state Verification. In (Boehm, D. et.al.): Proc. 21st Int. Conf. on Software Engineering, Los Angeles, 1999. IEEE Computer Society Press, Los Alamitos, CA, USA, 1999; pp. 411–420.

[DC94] Dwyer, M.; Clarke, M.: Data Flow Analysis for Verifying Properties of Concurrent Programs. In (Adrion, R.; Wile, D.): Proc. 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering, New Orleans, LA, USA, 1994. ACM Press, New York, USA; pp. 62–75.

[En02] Engels, A. et.al.: A Hierarchy of Communication Models for Message Sequence Charts. Science of Computer Programming, 44(3), 2002. Elsevier, Amsterdam; pp. 253–292.

[Fr94] Fromentin, E. et.al: On the Fly Testing of Regular Patterns in Distributed Computations. Internal Publication # 817, IRISA, Rennes, France, 1994.

[Ha03] Hallal, H. et.al: An Automata-based Approach to Property Testing in Event Traces. In (Hogrefe, D.; Wiles, D.): Proc. IFIP TC6/WG6.1 XV Int. Conf. on Testing of Communicating Systems (TestCom 2003). Sophia Antipolis, France, 2003. Kluwer Academic Publishers, Dordrecht, The Netherlands [to appear].

[Ha01] Hallal, H. et.al: Using SDL Tools to Test Properties of Distributed Systems. In (E. Brinksma, J. Tretmans): Proc. Workshop on Formal Approaches to Testing of Software (FATES) in affiliation with CONCUR'01; Aalborg, Denmark, 2001. BRICS Technical Report NS-01-4, Aug. 2001; pp. 125–140.

[Ho97] Holzmann, G.J.: The Model Checker SPIN. IEEE Transactions on Software Engineering, 23(5), 1997. IEEE Computer Society, Washington, DC; pp. 279–295.

[Ja96] Jackl, B.E.: Event-Predicate Detection in the Debugging of Distributed Applications. Master's Thesis. Department of Computer Science, University of Waterloo, 1996.

[Ja94] Jard, C. et.al.: In Proc. 14th IEEE Int. Conf. on Distributed Computing Systems, Poznan, Poland, 1994. IEEE Computer Society Press, Los Alamitos, 1994; pp. 396–403.

[LL95] Ladkin, P.B.; Leue. S.: Interpreting Message Flow Graphs. Formal Aspects of Computing, 7(5), 1995. British Computer Society, pp. 473–509.

[LL97] Leue, S; Ladkin, P.B.: Implementing and Verifying MSC Specifications Using Promela/Xspin. In (Grégoire, J.-C. et.al.): Proc. 2nd Int. Workshop on the SPIN Verification System, DIMACS Series Volume 32, American Mathematical Society, Providence, R.I., 1997; pp. 65–89.

[MZh99] Mansurov, N.; Zhukov, D.: Automatic Synthesis of SDL models in Use Case Methodology. In (Dssouli, R. et al): Proc. 9th SDL Forum, Montreal, Québec, Canada, 1999. Elsevier, Amsterdam; pp. 225–240.

[Ro97] Robert, G. et.al.: Deriving an SDL Specification with a Given Architecture from a Set of MSCs. In (Cavalli, A.; Sarma, A.): Proc. 8th SDL Forum, Evry, France, 1997. Elsevier, Amsterdam; pp. 22–26.

[Sm02] Smith, R.L. et.al.: An Approach Supporting Property Elucidation. In (Tracs, W. et.al): Proc. 24th Int. Conf. on Software Engineering, Orlando, FL, 2002. ACM, New York, USA, 2002; pp. 11–21.

[SMV] The SMV System, http://www-2.cs.cmu.edu/~modelcheck/smv.html. [online].

[OG] Telelogic: ObjectGEODE SDL Simulator Reference Manual. 2002.

[Ul03] Ulrich, A. et.al.: Verifying Trustworthiness Requirements in Distributed Systems with Formal Log-file Analysis. In Proc. 36th Annual Hawaii Int. Conf. on System Sciences (HICSS-36), Waikoloa, Hawaii, 2003. [CD ROM]

[Z100] ITU-T Recommendation Z.100: Specification and Description Language SDL. ITU General Secretariat, Geneva, Switzerland, 1999.