In *International Conference on Testing of Communicating Systems TESTCOM'03*, Sophia Antipolis, France, May 26-28, 2003.

Conference URL: http://www.testcom2003.org

An Automata-based Approach to Property Testing in Event Traces

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

[1] CRIM, 550 Sherbrooke West, Suite 100, Montreal, H3A 1B9, Canada,
{Hallal, Boroday, Petrenko}@crim.ca
[2] Siemens AG, Corporate Technology SE1, Otto-Hahn-Ring 6, 81739 München, Germany,
andreas.ulrich@siemens.com

Abstract. We present a framework for property testing where a partially ordered execution trace of a distributed system is modeled by a collection of communicating automata. We prove that the model exactly characterizes the causality relation between the events in the observed trace. We present the implementation of this approach in SDL, where ObjectGEODE is used to verify properties, and illustrate the approach with an industrial case study.

Keywords: monitoring, passive testing, distributed traces, property checking, SDL

1. Introduction

In the age of the Internet, distributed systems have emerged as a viable option to pave the information highway towards brighter horizons. However, the shine of distributed systems has always been dimmed by the cost associated with their design and validation. Although asynchrony and geographic distribution add to the value of distributed systems; the same characteristics render especially their validation difficult. Meanwhile, the possibility of using formal methods as a reliable means for the development and validation of distributed systems restores the hope that both the time-to-market and the validation cost could be slashed. Once a formal specification of the behavior of a system becomes available, many development activities, such as 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 of distributed systems and produce logfiles of execution traces that can be analyzed further. This type of analysis is also known as passive testing of distributed systems.

A general approach to trace analysis could be 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 can be applied 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. [3, 21, 31]. 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) [9, 12, 17].

Developing a tool for checking properties in execution traces, one faces the choice either to elaborate algorithms for a specific class of properties and to implement them in specialized tools or to reuse 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 developer of an analysis tool to rely 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 property checking in distributed systems based on traces collected during a system execution. Our approach relies on a partially ordered set (poset) of events, where the partial order is the traditional *happened-before* relation [5], and its corresponding 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 SUT using finite automata. Then, we prove that the composition of these automata is isomorphic to the lattice of the poset, which allows us to translate the problem of property testing in event traces to a typical model checker problem. Our framework for trace analysis treats both synchronous and asynchronous communications simultaneously.

From the theoretical perspective, our framework can be considered as a generalization of the framework proposed in [6], 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 as a partially ordered set of events, so existing techniques based on partial orders remain applicable. Notice also that [6] as well as [10] mostly concentrate on studying the hierarchy of various classes of traces, while we elaborate on modeling partial orders using automata for the purpose of trace analysis. Compared to [10], as well as other papers on the semantics and model checking of message sequence charts (MSCs) [2, 20], our framework includes local events and rendezvous that are ignored there. In [19] a global state graph (Büchi automaton) is built from an MSC, similar to [12] and [17] which consider the ideal lattice. The paper [17] describes an approach for trace analysis based on a method for 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 that is advocated in our paper allows us to consider more complex properties. Earlier findings are published in [14] and [30].

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 events, thus generalizing the framework of [6]. In Section 3, we detail our approach of modeling distributed processes by means of automata and show that the composition of automata and the automaton of the lattice of ideals are isomorphic. In Section 4, we present an implementation of our approach based on the commercial tool ObjectGEODE. We discuss the modeling of an event trace in SDL and the use of ObjectGEODE to verify properties of the SUT. Then, in Section 5, we apply our approach to an industrial case study. Finally, Section 6 concludes the paper and mentions directions for further work.

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 non-empty set E_i of events that are classified into *local* (*l*), *send* (*s*), *receive* (*r*), and *rendezvous* (*z*) events, and are the only events observable outside the system. The local events indicate some change in the local state of a process. Events of types *s* and *r* are generated when the corresponding processes exchange a message *m*, and a rendezvous event indicates that certain processes are involved in a rendezvous *v*. Let *M* and *R* denote the disjoint sets of messages and rendezvous, respectively. 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.

Assuming that each event is generated by a process P_i only once, the events constitute a sequence T_i whose length is $|E_i|$, called the *local trace* of P_i. P_i is sequential, so the events in E_i are totally ordered by a causal relation, $\prec_i$, defined by their occurrence in T_i. Each $\prec_i$ is an irreflexive total order. We define the mappings Γ_1, Γ_2, Γ_3 as follows:

- Γ_1 is defined from *M* into the set of send events, and $\Gamma_1(m) = s$, where *s* is the send of message *m*.
- Γ_2 is defined from *M* into the set of receive events, and $\Gamma_2(m) = r$, where *r* is the receive of message *m*.
- Γ_3 is defined from *R* into the set of rendezvous events, and $\Gamma_3(v) = z$, where *z* is the event with which a process P_i, such that $z \in E_i$, performs the rendezvous *v*.

Let $E = \bigcup E_i$ be the set of all the events generated in the system. In *E*, matching of receive with send events and identifying rendezvous events is defined by the relation Π that is the smallest subset of $E \times (M \cup R) \times E$ such that

- $(\Gamma_1(m), m, \Gamma_2(m)) \in \Pi$ for all $m \in M$ and
- $(\Gamma_3(v), v, \Gamma_3(v)) \in \Pi$ for all $v \in R$.

We consider in our framework only systems with point-to-point communications, which are formalized as follows.

Definition 1. Given a system of communicating processes P_1, …, P_n that execute the set of events *E*, exchange messages in the set *M*, and perform rendezvous in the set *R*, a relation $\Pi \subseteq E \times (M \cup R) \times E$ is a *point-to-point communication* relation if

1. the mappings Γ_1, Γ_2, and Γ_3 are bijections;
2. if $s, r \in E_i$ then $(s, m, r) \notin \Pi$ for all $m \in M$;
3. if $(z, v, z) \in \Pi$ then $|\{ i \mid z \in E_i \}| \geq 2$.

Conditions 1 and 2 state that all messages are unique and each message is transmitted between only two distinct processes. Similarly, conditions 1 and 3 ensure the uniqueness of rendezvous and restrict them to multi-party rendezvous (a process would not perform a rendezvous with itself). In the system of processes $P_1, \ldots, P_n$, causal dependencies between events from different processes are induced by the point-to-point communication relation.

Definition 2. The *causality* relation $\prec$ on the set of events E of the system $P_1, \ldots, P_n$ with the point-to-point communication relation Π is the smallest subset of $E \times E$ that satisfies the following rules.

1. If $a \prec_i b$ then $a \prec b$.
2. If $(s, m, r) \in \Pi$ then $s \prec r$.
3. If $a \prec b$ and $b \prec c$ then $a \prec c$.

The third condition ensures transitivity. Thus the causality relation is a transitive closure of a relation defined by the local orders and the precedence of a send event over the matching receive event. An event a is said to be a *predecessor* of another event b if $a \prec b$. Thus, a is a predecessor of b if and only if there exists a finite sequence $c_0, c_1, ..., c_k$ such that $a = c_0$ and $b = c_k$, and for every j, $1 \leq j \leq k$, either $c_{j-1} \prec_i c_j$ for some i or $(s, m, r) \in \Pi$ for some message m.

For an arbitrary collection, $(T_1, \ldots, T_n)$, of local traces of events generated by the processes of a distributed system to be causally consistent, we require that the causality relation on the set of events in a tuple of local traces be cycle-free.

Definition 3. A tuple $(T_1, \ldots, T_n)$ of local traces produced by a system $P_1, \ldots, P_n$ is said to be an *event trace* of the system if the causality relation $\prec$ on the set of events E is irreflexive.

Since the causality relation is transitive by definition, an event trace is defined when the relation on E is a strict partial order. On the other hand, the reflexive closure of $\prec$, denoted $\preccurlyeq$, ($a \preccurlyeq b$ if and only if $a \prec b$ or $a = b$) is a partial order, and $(E, \preccurlyeq)$ is a poset.

An *ideal* P of poset $(E, \preccurlyeq)$ is a subset of E such that if $e \in P$ and $e' \preccurlyeq e$ then $e' \in P$. Let $I(E, \preccurlyeq)$ denote the set of all the ideals of poset $(E, \preccurlyeq)$. It is known that $I(E, \preccurlyeq)$ and a set inclusion relation form a lattice, which we denote by $(I(E, \preccurlyeq), \subseteq)$ and call it the *ideal* lattice (also called the lattice of consistent cuts or global states). Since in our case, the set of events is finite, the set E is the supremum and the empty set $\varnothing$ is the infimum of the lattice $(I(E, \preccurlyeq), \subseteq)$. The ideal lattice represents all the possible interleavings of events and provides a simple way to check properties of concurrent systems [17].

Here, the question arises how to build the ideal lattice of an event trace. Several methods have already been discussed, see, e.g., [17]. 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 [17] 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 [15], SMV [27], or ObjectGEODE [29], 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 can be modeled by a system of communicating automata and demonstrate that the system exactly characterizes the causality 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 denoted 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. An automaton is *deterministic* if $q \text{ –}a\rightarrow q'$ and $q \text{ –}a\rightarrow q''$ imply that $q' = q''$ for all $a \in \Sigma$ and $q \in Q$. A state q of an automaton is said to be *reachable* from state q' if there exists a sequence of transitions $q' \text{ –}a_1\rightarrow q_1 \text{ –}a_2\rightarrow q_2 \ldots \text{–}a_i\rightarrow q_i$, where $q_i = q$. We use q ***after*** σ to denote the state q_i reached from the state q after a finite sequence of transitions, $q \text{ –}a_1\rightarrow q_1 \text{ –}a_2\rightarrow q_2 \ldots \text{–}a_i\rightarrow q_i$, where $\sigma = a_1a_2\ldots a_i$. We assume that each state is reachable from itself with an empty word ε, i.e., q ***after*** $\varepsilon = q$. An automaton is *initially connected* if all its states are reachable from the initial state. In the following, we consider only automata that are deterministic and initially connected. We define the set ***events***(σ) to be the set of all events in a given word σ. We call $L_{acc}(q)$ the language accepted by an automaton initialized in state q. The language accepted by the automaton starting in the initial state is denoted $L_{acc}(A)$. An initially connected automaton A is *minimal* if all the final states of A are reachable from every state, and $L_{acc}(q) \neq L_{acc}(q')$ for each two states q and q', i.e., states are pairwise distinguishable.

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 $A_1 \parallel A_2$, 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 $q_1 \text{ -}a\rightarrow_1 q'_1$ and $a \notin E_2$ then $(q_1, q_2) \text{ -}a\rightarrow (q'_1, q_2)$.
- If $q_2 \text{ -}a\rightarrow_2 q'_2$ and $a \notin E_1$ then $(q_1, q_2) \text{ -}a\rightarrow (q_1, q'_2)$.
- If $q_1 \text{ -}a\rightarrow_1 q'_1$ and $q_2 \text{ -}a\rightarrow_2 q'_2$ then $(q_1, q_2) \text{ -}a\rightarrow (q'_1, q'_2)$.
- $Q_f = Q \cap (Q_{f1} \times Q_{f2})$.

The composition is associative; it can be applied to finitely many automata. The composition of n automata $C_1 \ldots C_n$ is an automaton $C = C_1 \parallel \ldots \parallel C_n$ over the alphabet $E = E_1 \cup \ldots \cup E_n$. Let $\omega \in E^*$, we use $\omega_{\downarrow Et}$ to denote the projection of ω onto the set E_t. The following properties of the composition automaton can directly be established from the definition.

Proposition 4. If $(q_1, \ldots, q_p)$ ***after*** ω is defined then $(q_1, \ldots, q_p)$ ***after*** $\omega = (q_1$ ***after*** $\omega_{\downarrow E1}, \ldots, q$ ***after*** $\omega_{\downarrow Ep})$. If $\omega \in L_{acc}(C)$ then $\omega_{\downarrow Ei} \in L_{acc}(C_i)$.

3.1 Ideal Lattice Automaton

An ideal lattice is usually visualized as a graph, called the covering digraph [17], 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.

Definition 5. Given an event trace $(T_1, \ldots, T_n)$ with the causality relation $\prec$, the *ideal lattice automaton*, denoted T, is a tuple $<E, I(E, \preccurlyeq), \varnothing, \rightarrow_T, \{E\}>$, where E is the set of events, $I(E, \preccurlyeq)$ is the set of states and $\varnothing$ is the initial state, $\rightarrow_T = \{(P, a, R) \mid P, R \in I(E, \preccurlyeq), R = P \cup \{a\}$, and $R \neq P\}$, $\{E\}$ is the set of final states.

Each word accepted by the ideal lattice automaton contains exactly one instance of an event in E and thus defines a total order that is a linearization of the causality relation $\prec$, i.e., the word is one possible interleaving of the events in the trace. The accepted language $L_{acc}(T)$ of the ideal automaton consists exactly of all linearizations of the strict partial order $\prec$ [17, 4]. In the following, we will not distinguish between a total order on E and the word it defines, thus $\sigma \in L_{acc}(T)$ also denotes a total order such that $\sigma \supseteq \prec$. The states of the ideal lattice automaton 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.

Proposition 6. The ideal lattice automaton is minimal.

Proof. By construction, T is initially connected. Then, by definition of a lattice, an infinum and a supremum exist for every subset of the lattice. This implies that in the ideal lattice automaton there exists a sequence of transitions from every state to the final state that is the supremum E of the lattice. This means that the final state of the ideal lattice automaton is reachable from every state. We prove now that the states of T are pairwise distinguishable. Let I, J be two different states (ideals) of T. $L_{acc}(I)$ and $L_{acc}(J)$ are the sets of the linearizations of the posets $(E \setminus I, \preccurlyeq)$ and $(E \setminus J, \preccurlyeq)$, respectively. From $I \neq J$ it follows that $E \setminus I \neq E \setminus J$. Hence $L_{acc}(I) \neq L_{acc}(J)$. QED.

3.2 Composition of Automata

Given an event trace $(T_1, \ldots, T_n)$ with the causality relation $\prec$, we define $p = n + |M|$ communicating automata, where n automata model the processes $P_1, \ldots, P_n$ and accept the corresponding local traces $T_1, \ldots, T_n$, respectively, and $|M|$ automata that model message delays inherent to asynchronous communications. A delay automaton of message $m \in M$ is a three state automaton that accepts the word $s.r$ such that $\Gamma_1(m) = s$ and $\Gamma_2(m) = r$. A similar approach is taken in [10] where communications via individual message channels are used instead of message delay automata. In our work, we consider a more general case of traces of systems with mixed synchronous and asynchronous communications instead.

Definition 7. Given a process P_i with its local trace T_i, the *local trace* T_i automaton is the tuple $<E_i, I(E_i, \preccurlyeq), \varnothing, \rightarrow_{Pi}, \{E_i\}>$, where E_i is the set of events, $I(E_i, \preccurlyeq)$ is the set of states, $\varnothing$ is the initial state, $\rightarrow_{Pi} = \{(P, a, R) \mid P, R \in I(E_i, \preccurlyeq), R = P \cup \{a\}$, and $R \neq P\}$, and $\{E_i\}$ is the set of final states.

According to Definition 5, the local trace automaton corresponding to local trace T_i is the ideal lattice automaton of the local (total) order $\prec_i$. Thus, according to Proposition 6 a local trace automaton is minimal. The accepted language of such an automaton contains exactly one word that is the corresponding local trace.

Definition 8. Given a message $m_i \in M$, let $\Gamma_1(m_i) = s_i$, and $\Gamma_2(m_i) = r_i$, the *message* m *delay automaton* is the tuple $<E_{i+n}=\{s_i, r_i\}, \{\varnothing, \{s_i\}, \{s_i, r_i\}\}, \varnothing, \rightarrow_{mi}, \{\{s_i, r_i\}\}>$, where $\{s_i, r_i\}$ is the set of events, $\{\varnothing, \{s_i\}, \{s_i, r_i\}\}$ is the set of states, $\varnothing$ is the initial state, $\rightarrow_{mi} = \{(\{\varnothing\}, s_i, \{s_i\}), (\{s_i\}, r_i, \{s_i, r_i\})\}$, and $\{\{s_i, r_i\}\}$ is the set of final states.

Each message delay automaton is an ideal lattice automaton of a total order of two events, reflecting the precedence of a send event over the matching receive event and thus is minimal.

Let $C_1, \ldots, C_p$ be the set of (component) automata that model n communicating processes $P_1, \ldots, P_n$ and $|M|$ message delay automata, where $C_i = <\Sigma_i, Q_i, q_{0i}, \rightarrow_i, \{q_{fi}\}>$, such q_{0i} is the initial state of C_i that corresponds to an empty ideal, and q_{fi} is the final state of C_i. Consider the composition $C_1 \parallel \ldots \parallel C_p = C$. The set of events of C is the union of all Σ_i, where Σ_i is either E_i or $\{s_i, r_i\}$ for an appropriate $m_i \in M$. Hence the set E of all events of the given distributed system constitutes the set of events of C. The initial state of C is $q_0 = (q_{01}, \ldots, q_{0p})$. The set of states Q_C and the transition relation $\rightarrow$ of C are defined by the semantics of the composition operator $\parallel$. The set of final states of the composition automaton is $Q_f = Q_C \cap \{(q_{f1}, \ldots, q_{fp})\}$, which means that Q_f contains only one state or is empty when $(q_{f1}, \ldots, q_{fp})$ is not reachable in C. Thus, $C = <E, Q_C, q_0, \rightarrow, Q_f>$.

3.3 Isomorphism

Next we show that the composition of the local trace automata and message delay automata is isomorphic to the automaton of the ideal lattice based on the fact that two minimal automata that accept the same language are isomorphic. We already proved that the ideal automaton is minimal. We shall first demonstrate that the composition automaton is minimal and then that the two automata accept the same language. By definition of the composition operator, the composition automaton is initially connected. So, we need to show that the state $(q_{f1}, \ldots, q_{fp})$ is reachable from all states of the composition automaton and that all states are pair wise distinguishable.

Proposition 9. Given an event trace $(T_1, \ldots, T_n)$ with the causality relation $\prec$, the state $(q_{f1}, \ldots, q_{fp})$ is reachable from every state of the composition automaton C.

Proof. Assume that state $(q_{f1}, \ldots, q_{fp})$ is not reachable from some state in Q_C. Two cases are possible here: the state belongs to a cycle or there is a deadlock state with no outgoing transitions. There is no cycle in the composition automaton, since each component automaton is acyclic, see Proposition 4. Then let $q \in Q_C$ be a state with no outgoing transitions. The automaton C is initially connected, thus there exists a word $\sigma \in E^*$ such that $q = q_0$ ***after*** $\sigma = (q_{01}$ ***after*** $\sigma_{\downarrow E1}, \ldots, q_{0p}$ ***after*** $\sigma_{\downarrow Ep})$ and there exists a

component automaton C_i such that q_{0i} ***after*** $\sigma_{\downarrow Ei} \neq q_{fi}$. Then ***events***$(\sigma_{\downarrow Ei}) \subset E_i$ and there exists an event $e \in E_i \setminus$ ***events***$(\sigma_{\downarrow Ei})$ such that all the predecessors of e (in E_i) are in q_{0i} ***after*** $\sigma_{\downarrow Ei}$. Hence e cannot be a local event, so there exists at least one component automaton C_j ($j \neq i$) such that $e \in C_i \cap C_j$. Moreover, state q_{0j} ***after*** $\sigma_{\downarrow Ej}$ has no transition on e, otherwise the composition automaton would have a transition from q. There are two cases possible: either $e \in q_{0j}$ ***after*** $\sigma_{\downarrow Ej}$ or $e \notin q_{0j}$ ***after*** $\sigma_{\downarrow Ej}$. In the first case, $e \in \sigma$ and, according to Proposition 4, $e \in \sigma_{\downarrow Ei}$ and thus $e \in q_{0i}$ ***after*** $\sigma_{\downarrow Ei}$. But, $e \in E_i \setminus$ ***events***$(\sigma_{\downarrow Ei})$, a contradiction. Consider now the second case, $e \notin q_{0j}$ ***after*** $\sigma_{\downarrow Ej}$, this implies that q_{0j} ***after*** $\sigma_{\downarrow Ej} \neq q_{fj}$. Then there should exist a transition from state q_{0j} ***after*** $\sigma_{\downarrow Ej}$ on another event, say e', that is a predecessor of e. Similar to the above considered cases, e' cannot be a local event and therefore there should exist another component automaton that shares this event and so on. Due to a finite number of events in the composition, an event must reoccur in this sequence. This means that a reoccurring event is a predecessor of itself. This is impossible due to irreflexivity of the causality relation $\prec$ on the event trace. So, $(q_{f1}, \ldots, q_{fp})$ is the final state reachable from all the states in Q_C the composition automaton for an event trace. QED.

Proposition 10. States of C are pairwise distinguishable.

Proof. Assume that there exist two states $q = (q_1, \ldots, q_i, \ldots, q_p)$ and $q' = (q'_1, \ldots, q'_i, \ldots, q'_p)$ that are not distinguishable, in other words, $L_{acc}(q) = L_{acc}(q')$. According to Proposition 4, for each i it holds that $L_{acc}(q_i) = L_{acc}(q'_i)$ since $L_{acc}(q_i)$ and $L_{acc}(q'_i)$ are both singletons. Each component automaton is minimal, thus, $q_i = q'_i$. This implies $q = q'$. QED.

Thus, the following statement holds.

Lemma 11. The composition automaton C is minimal.

Next, we show that the two automata, C and T, accept the same language, i.e., $L_{acc}(T) = L_{acc}(C)$.

Proposition 12. $L_{acc}(C) \subseteq L_{acc}(T)$.

Proof. Proposition 4 states that each projection of an arbitrary word ω, accepted by C, takes the corresponding component automata to the final state. Thus, all events of E are present in ω. In addition, any event occurs exactly once in ω, otherwise this would mean that events reoccur in component automata. Moreover, by construction, words accepted by local trace and message delay automata define the corresponding local orders and the send-receive precedence, respectively. Hence the word ω satisfies these total orders. Since the causality relation is just a transitive closure of local orders along with send-receive precedence, ω represents a linearization of the causality relation. Therefore, the ideal lattice automaton accepts ω. QED.

To prove that $L_{acc}(T) \subseteq L_{acc}(C)$, we need an auxiliary proposition, which establishes a relation between words of $L_{acc}(T)$ and states of the composition automaton C. Let $E_{\downarrow\sigma}$ denote $E \cap$ ***events***(σ) and let σ_k be a prefix of length k of a word $\sigma \in L_{acc}(T)$.

Proposition 13. For any k, $0 \leq k \leq |E|$, prefix σ_k of $\sigma \in L_{acc}(T)$, q_0 ***after*** $\sigma_k = (E_{1\downarrow\sigma k}, \ldots, E_{p\downarrow\sigma k})$.

Proof. Base of induction. For $k = 0$, σ_0 is an empty word, and q_0 ***after*** $\sigma_0 = q_0 = (q_{01}, \ldots, q_{0p}) = (q_{01}$ ***after*** $\varepsilon, \ldots, q_{0p}$ ***after*** $\varepsilon)$.

Inductive step. Let $\sigma_k = e_1\, e_2 \ldots e_k$ and $\sigma_{k+1} = e_1\, e_2 \ldots e_k\, e_{k+1}$, where $e_1, e_2, \ldots, e_k, e_{k+1} \in E$. Assume q_0 ***after*** $\sigma_k = (E_{1\downarrow\sigma k}, \ldots, E_{p\downarrow\sigma k})$, we need to show that for word σ_{k+1}, q_0

after $\sigma_{k+1} = (E_{1\downarrow\sigma k+1}, ..., E_{p\downarrow\sigma k+1})$. So, it is sufficient to prove that q_k ***after*** $e_{k+1} = (E_{1\downarrow\sigma k+1}, ..., E_{p\downarrow\sigma k+1})$. According to the properties of a partial order, $\sigma_{k+1} = \sigma_k e_{k+1}$ is a linearization of an ideal, so all the predecessors of e_{k+1} are in σ_k. Then $E_i\downarrow\sigma_k$ and $E_i\downarrow\sigma_{k+1}$ are different ideals of the total order $(E_i, \preccurlyeq)$ for any E_i that contains e_{k+1}. This means that the automaton C_i has a transition $(E_i\downarrow\sigma_k, e_{k+1}, E_i\downarrow\sigma_{k+1})$. Thus $E_i\downarrow\sigma_{k+1} = E_i\downarrow\sigma_k \cup \{e_{k+1}\}$; if $E_i\downarrow\sigma_{k+1} = E_i\downarrow\sigma_k$ then $e_{k+1} \notin E_i$. According to the definition of the composition operator, there exists a transition $(E_1\downarrow\sigma_k, ..., E_p\downarrow\sigma_k), e_{k+1}, (E_1\downarrow\sigma_{k+1}, ..., E_p\downarrow\sigma_{k+1})$. QED.

Proposition 14. $L_{acc}(T) \subseteq L_{acc}(C)$.

Proof. According to Proposition 13, each word σ accepted by T takes C to state $(E_{1\downarrow\sigma}, ..., E_{p\downarrow\sigma}) = (E_1, ..., E_p)$, which, according to the construction of C, is its final state. QED.

Lemma 15. $L_{acc}(T) = L_{acc}(C)$.

Lemma 11 and 15 imply the following.

Theorem 16. The composition automaton C and the ideal lattice automaton T are isomorphic.

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

4. Using SDL Tools in Trace Analysis

By following the approach to trace analysis outlined in the previous sections, we translate the problem of trace analysis into a typical model checking problem. So, given a logfile of events, implementation of the approach is reduced to the following tasks:

1. Extracting an event trace from a logfile produced by a monitoring tool.
2. Choosing an appropriate model checking tool and describing a system of local trace and message delay automata in the specification language of this model checker.

4.1 Event Trace Extraction

Monitoring a distributed system yields a *logfile* that comprises a not necessarily ordered collection of events indicating the actions executed. To verify whether the logfile describes an event trace, we have to analyze each event individually and its relation to others. We assume that every event is recorded in a certain structure, the *event record*, that contains information sufficient to identify the type of the event, the process that generated it, and to determine properties of the causality relation. Moreover, it must be possible to verify whether all events satisfy the point-to-point communication relation. In our framework, we assume that every event record in the logfile has the following fields:

- Type of event: Communication (Send, Receive, or Rendezvous), Local
- Name of the issuing process
- Local ordinal number
- Partner for communication events

- Message / rendezvous parameters
- Local parameters of the issuing process (variables, …).

In a logfile, each event is uniquely identified by its complete event record structure. Local ordinal numbers determine the local order of each process in the system. As explained in Sect. 2, matching of receive with send events and identifying rendezvous events is defined by the point-to-point communication relation. So communication events should be verified for the existence of their matching events. The final check should make sure that the causal relation between the events in the logfile is a strict partial order.

4.2 Choosing a Model Checker

A system of communicating automata can be extracted from an event trace and described in the language of a chosen model checker fully automatically. Therefore, the choice is dictated by the richness of the language used for property specification and not by the system to be analyzed. User friendliness is another important selection criterion. Indeed, to be accepted by practitioners, any integrated environment for trace analysis should hardly require test engineers to master, for example, a temporal logic.

From this viewpoint, ObjectGEODE (OG for short) of Telelogic [29] meets the above requirements. OG supports modeling systems in SDL extended with synchronous communication and performing specification simulation and property checking. Moreover, OG provides the SDL-like language GOAL (GEODE Object Automata Language) to specify properties [1, 29]. Once an SDL specification is built from a logfile, the OG simulator performs the verification of specified properties. To deploy OG, we need to translate the automata model into an SDL system, addressing the aspects of structure, behavior, communication, and data.

The structure of an SDL system is modeled using a hierarchy of *system* / *block* / *process* / *procedure* statements. In our case, it is sufficient to define a system with a single block that is composed of several processes. The obvious approach is to map each local trace and message delay automaton into a designated SDL process. The communication between processes, which is point-to-point and represents multi-party rendezvous, is then achieved via signals that represent data exchanged in SDL channels. To do so, the representation of an automaton can be obtained by projecting the collected trace onto the set of events of this automaton: send, receive, rendezvous, and local events. Each send event of the automaton translates to an OUTPUT statement in SDL with the signal name showing the name of the message. Similarly, each receive event is mapped to an INPUT statement, and a local event is translated to TASK statements. As for rendezvous events, SDL does not support synchronous communications, but OG extends standard SDL by allowing the definition of rendezvous channels, over which signals are exchanged without any delay. However, this extension allows only two-party rendezvous to be directly modeled, thus we restrict our implementation of the approach to cover only this type of rendezvous. Subsequently, states are inserted between the events. The overall behavior of the system is modeled by the joint behavior, represented as a global state graph, of all communicating processes in SDL.

Such a straightforward solution requires the definition of as many SDL processes as there are processes and messages in the given event trace. This leads to a large system

specification and could represent a significant obstacle to the model checker in handling the model during syntax and semantic checking and simulation. A better solution consists in replacing message delay automata by individual channels, where a separate channel with associated individual input queue is used to carry every message. These input queues play the role of the message delay automata. An advantage of this approach is the drastic reduction of the number of processes defined in the model. However, individual channels are not supported in standard SDL.

The need to further reduce the size of the resulting SDL specification motivates the search for another solution to the problem of translating message delay automata to SDL. We decided to use the standard SDL input queue. However, to breach FIFO order of consumption and prevent discarding messages during simulation, we use an SDL feature, the asterisk SAVE construct, which prohibits a process in its current state from discarding those signals which could not be consumed by an explicitly defined transition, as suggested in [25] and [22, 18]. This prevents signal loss when signals arrive at the input queue of a process out of the expected order. The saved signals are kept for future consumption, which means that messages are delivered finally in the order as stored in the collected event trace.

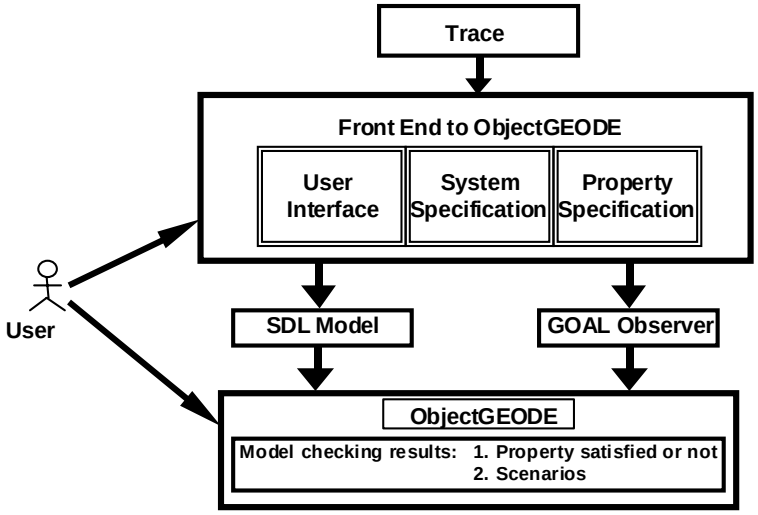

Fig. 1. The workflow of the approach.

4.3 Workflow of the Approach Using SDL and OG

The approach based on OG is summarized in the workflow shown in Figure 1. The implementation efforts are reduced to building just a front-end to OG featuring:

1. A system specification tool TRAYSIS that builds an SDL specification from the collected trace.
2. A property specification tool called Property Manager that eases the process of writing properties for trace verification.

4.3.1 The System Specification Tool

We present here the Trace Analysis (TRAYSIS) tool, Figure 2, which currently accepts logfiles with either synchronous or asynchronous communications. Its main task is to automate the process of producing an SDL specification from a logfile of an event trace specified in XML syntax. The tool offers the following main features :

- XML logfile treatment and model construction. The tool checks the logfile as discussed in Section 4.1. In case of missing communication events, the tool informs the user and converts the unmatched communication events into local events of the processes. The tool also checks for flaws in the logfile that would render the generated SDL model syntactically incorrect, e.g. the presence of SDL keywords in the logfile and the use of some illegal characters. Once a logfile is successfully checked, the tool generates the SDL model of the system. Causality cycles are then detected by OG during simulation of the generated SDL model as they cause deadlock prior to the termination of the processes.
- Customized model generation. This feature enables the user to cope with large logfiles. The tool offers three types of filtering**:** process filter, signal filter, and a filter that extracts a segment from the initial logfile.
- Statistics about the SDL model. The tool offers listings of processes, variables, and signals occurring in the model to help the user better understand the generated model.

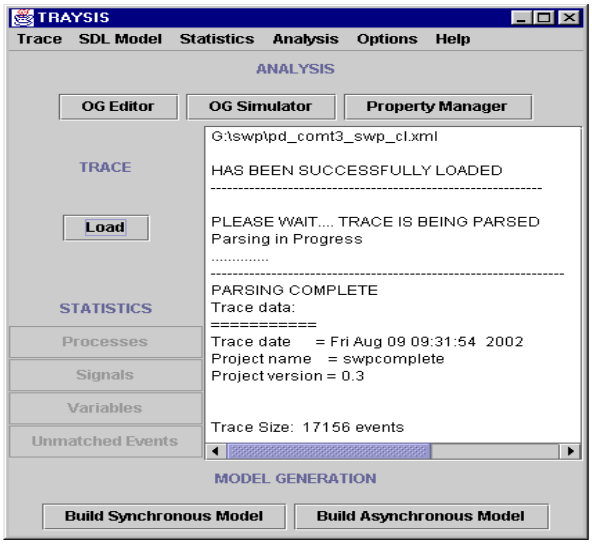

Fig. 2. The TRAYSIS tool.

4.3.2 Pattern Specification

There is a wide range of properties that can be sought in distributed systems. These properties are classified into two types: state-based and event-based properties. Event-based properties are expressed in terms of atomic or composite events of the distributed system, while state-based properties formulate assertions on state variables of processes of the system. The work of Dwyer et al. to build a repository of typical

and frequently used specification patterns (similar to design patterns) [8], [24] is an effort to consider both types of properties. For our project, we rely on the repository from [24] to build a library [11] of specification property templates in GOAL in which we also consider mixed properties that are expressed in terms of both events and state predicates. However, we needed to modify the original patterns, with the addition of a termination predicate, to count for the fact that we deal only with finite traces in our post mortem analysis.

GOAL is an automata-based language to specify properties of SDL models. The structure of GOAL, which is similar to SDL [1], allows one to specify any event-based or state-based properties as well as mixed properties as long as they are expressible by automata [13]. A property presentation in GOAL is called an *observer* that implements an extended finite automaton with accepting states. Observers are usually described in terms of entities (objects, signals, variables etc.) of the system. In addition, GOAL allows the declaration of two types of designated states: *success* and *error* states. Entering a success state (error state) indicates that the system respects (violates) the property expressed in the observer. However, the use of the success/error convention is completely up to the user and has no formal meaning. In our observers, reachability of error states indicates that the property is not satisfied, and absence of error states indicates that the property is satisfied (independently of interleavings). Presence of success indicates that there are possible interleavings on which the property is satisfied. During simulation, OG builds the synchronous product of the observer and the specification of the system. The output includes a report on the number of error and success states encountered along with scenarios that lead to these observer states. Scenarios can be replayed as counter or supporting examples later.

The Property Manager is a tool that supports the process of property specification for trace analysis in OG by offering the user a library of predefined patterns [11]. Every pattern of the library is a parameterized template of a GOAL observer. The tool helps the user customize the observer to specific settings of the observed event trace. A screenshot of the Property Manager tool is shown in Figure 3.

5. Case Study

We applied the trace analysis approach in an industrial case study. The task of a development team at Siemens' Power Transmission and Distribution Division was to implement a *sliding window protocol* (SWP) mechanism on top of the factory automation protocol suite PROFIBUS. This extension of PROFIBUS is required to support communication among distributed power control devices within regional power grids. A protocol analyzer observes the exchange of PROFIBUS messages within the communication network and produces an execution trace of the behavior of the system.

The SWP is a well-anticipated means of flow control for a reliable data transfer taking place between two communication partners and is well studied [28]. The protocol in our case study offers a symmetrical service for both communication partners, i.e., partners, server and client, can send data, which is flow-controlled. The protocol uses the two messages *TVoid* to send data to a partner and *TWindowAck* to send an acknowledgement back. Both messages carry the two parameters *Win_Send* and *Win_Quit* indicating the sequence numbers of the latest message sent and the

latest acknowledgement received, respectively. The sliding window protocol requires that no more than a certain, pre-defined number of messages (the maximum window size) be sent in a row before an acknowledgement is received. A single acknowledgement can confirm the receipt of up to *mws* unacknowledged messages simultaneously, where *mws* is the maximum window size.

In our case study, we verified various properties of the behavior of the network of communicating power control devices. These properties include the check whether the maximum window size is always respected and the following distributed properties. The first property requires that the total number of unacknowledged messages in the whole network do not exceed a certain threshold at anytime. The requirement is specifically of interest in systems where concurrency control is required. The second property states that the total number of messages in transit does not exceed a certain limit. These properties help the developers of the system evaluating the efficiency and reliability of the communication media and checking whether their occurrence contributes to undesired behavior. It is clear that these properties cannot be checked locally at the server side since the server is unaware of any messages in transit.

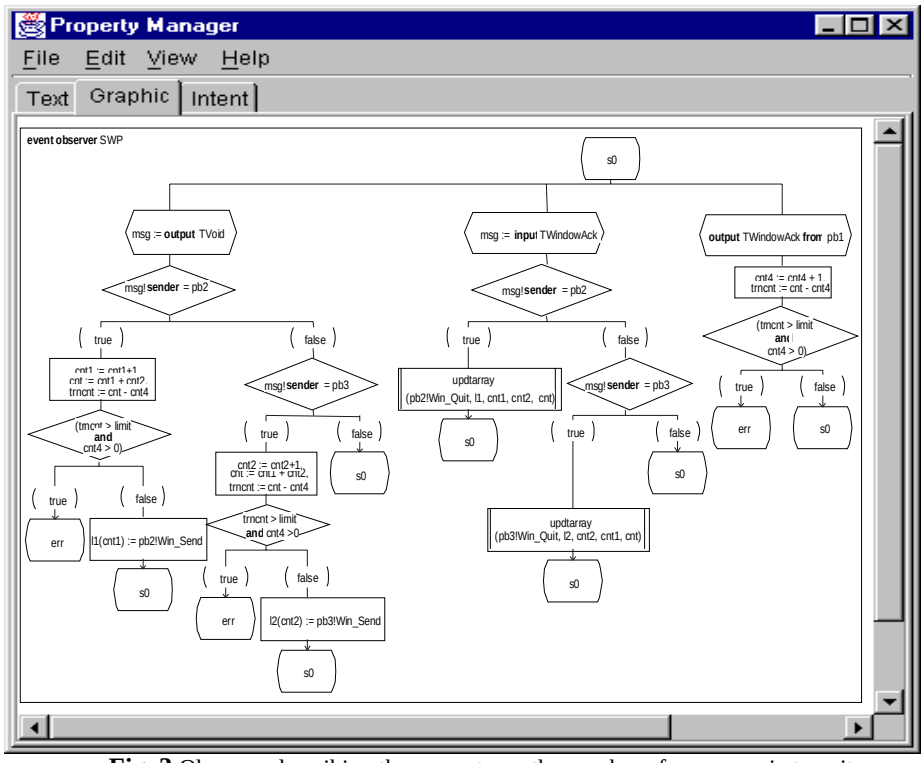

Fig. 3 Observer describing the property on the number of messages in transit.

We formalized the last property in GOAL using the “global response” property template taken from our template library. Due to lack of space we present a simplified version of the observer in Figure 3. The observer uses three probes *pb*1, *pb*2, and *pb*3 to get access to variables that are otherwise internal to the SDL system. The SDL

processes *a*2, *a*40, and *a*36 are the communication partners that were identified from the recorded trace, where *a*2 is the server and the remaining processes act as clients. Their behaviors are modeled based on the trace information and the model construction rules for asynchronous communication as outlined before. The observer waits for the occurrence of an output message at any of the client processes (left branch) and stores the message number of the sent message (*Win_Send*) in an array to keep track of unacknowledged messages. Every time a message is sent by a client, the observer checks whether the total number of messages in transit exceeds a certain limit (2 messages) that is specified by the variable *limit*. In this case the observer reaches the error state. The number of messages in transit in the system is the difference between the number of messages sent by the clients and the number of acknowledgement messages issued by the server (detected in the right branch). However, the observer does not signal a violation before the first acknowledgment is detected which indicates that the server is alive. Once the observer detects that an acknowledgement message is received by any of the clients (the middle branch), it compares the acknowledgement number (*Win_Quit*) with the message numbers stored in the array of this client and resets the array and the counters appropriately by calling the *updtarray* procedure.

During verification in OG, the observer is triggered by the occurrences of *TVoid* and *TWindowAck* messages. If the error state *err* is reached, the verification process terminates. Then, an error scenario is created that depicts the history of the system behavior. This error scenario can be visualized as an MSC. The visualization helps test engineers and developers understand better the cause of the property violation. In our case study, the verification of a trace containing about 1000 messages exchanged between the involved processes was done relatively fast since the resulting state space remained rather small (20800 states). The maximum number of messages in transit detected in the observed network helped assessing the bandwidth of the network.

6. Conclusion

We presented a general formal framework for trace analysis of distributed systems that applies to a logfile of events collected during execution of the system. We formally defined an event trace to consist of message send and receive events, rendezvous events, and local events; partially ordered by an irreflexive causality relation. We modeled an event trace by a system of communicating automata that correspond to the processes observed in the SUT and demonstrated that the system exactly characterizes the causality relation of the event trace. We proved that the composition of the automata is isomorphic to the ideal lattice automaton defined by the event trace.

We used SDL and the general-purpose model checker ObjectGEODE to implement our approach. We implemented the front-end tool TRAYSIS to carry out the translation from a logfile to an SDL specification of the SUT. We also built a library of property patterns in GOAL and interfaced it with the Property Manager tool to customize patterns to specific settings. Last but not least, we illustrated the trace analysis approach on an industrial case study on the verification of sliding window protocol implementations.

Our tools allow us to deal with relatively large traces (we processed traces of about twenty thousand events). However, more work is required to cope with even larger

traces from industrial applications. Our case study indicates that further development of user-friendly property specification tools and pattern libraries based on the automata approach is also needed.

References

1. B. Algayres, Y. Lejeune, E. Hugonnet, "GOAL: Observing SDL behaviors with GEODE", *in SDL'95 with MSC in CASE (ed. R. Braek, A. Sarma), Proc. of the 7th SDL Forum*, Oslo, Norway, Sept. 1995, Elsevier Science Publishers B. V. (North Holland), pp. 359-372.
2. R. Alur and M. Yannakakis. "Model Checking of Message Sequence Charts". *In CONCUR'99: Concurrency Theory, Tenth International Conference*, LNCS 1664, pages 114--129, 1999.
3. J. P. Black, M. H. Coffin, D. J. Taylor, T. Kunz, A. A. Basten, "Linking Specifications, Abstraction, and Debugging", *CCNG Technical Report E-232, Computer Communications and Network Group*, University of Waterloo, Nov. 1993.
4. R. Bonnet and M. Pouzet. "Linear Extensions of Ordered Sets". *In I. Rival, editor, Ordered Sets*, pages 125-170, D. Reidel Publishing Company, 1982.
5. K. Chandy, L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computing Systems* 3(1), pp. 63-75, Feb. 1985.
6. B. Charron-Bost, F. Mattern, and G. Tel. "Synchronous, Asynchronous, and Causally Ordered Communications". *Distributed Computin*g, 1995.
7. F. Dietrich, X. Logean, S. Koppenhoefer, J.-P. Hubaux, "Testing Temporal Logic Properties in Distributed Systems", *In Proc. of the 11th International Workshop on Testing of Communicating Systems*, Tomsk, Russia, Aug. 1998.
8. M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in Property Specifications for Finite-state Verification", *In Proc. 21st International Conference on Software Engineering*, May 1999.
9. M. Dwyer, L. Clarke, "Data Flow Analysis for Verifying Properties of Concurrent Programs", *In Proc. of ACM SIGSOFT*'94, New Orleans, LA, USA, 1994.
10. A. Engels, S. Mauw, and M.A. Reniers. "A Hierarchy of Communication Models for Message Sequence Charts". *In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, FORTE/PSTV'97*, pages 75-90, Osaka, Japan, Nov. 1997. Chapman & Hall. To appear in Science of Computer Programming, 2002.
11. Q. Fan. "Formalizing Properties for Distributed System Testing", *Master Thesis in Preparation*.
12. E. Fromentin, M. Raynal, V. Garg, and A. Tomlinson, "On the Fly Testing of Regular Patterns in Distributed Computations", *Internal Publication* # 817, IRISA, Rennes, France, 1994.
13. R. Groz, "Unrestricted Verification of Protocol Properties on a Simulation Using an Observer Approach", *Protocol Specification, Testing and Verification*, VI, Montréal, Canada, North-Holland, 1986, pp. 255-266.
14. H. Hallal, A. Petrenko, A. Ulrich, S. Boroday, "Using SDL Tools to Test Properties of Distributed Systems", *In proc. of Workshop on Formal Approaches to Testing of Software (FATES) in affiliation with CONCUR'01*; Aalborg, Denmark; BRICS Technical Re-port NS-01-4, Aug. 2001.
15. G.J. Holzmann. "The Model Checker SPIN". *IEEE Transactions on Software Engineering*, 23(5):279-295, 1997.
16. B. E. Jackl, "Event-Predicate Detection in the Debugging of Distributed Applications", *Master's Thesis*. Department of Computer Science, University of Waterloo, 1996.
17. C. Jard, T. Jeron, G. V. Jourdan, and J. X. Rampon, "A General Approach to Trace-checking in Distributed Computing Systems", *In Proc. IEEE Int. Conf. on Distributed Computing Systems*, Poznan, Poland, Jun. 1994.
18. KLOCwork. Corporate website, http://www.KLOCwork.com/.

19. P. B. Ladkin and Stefan Leue. "Interpreting Message Flow Graphs". *Formal Aspects of Computing 7(5*), p. 473 - 509, Sept./Oct. 1995.
20. Leue and P.B. Ladkin. "Implementing and Verifying MSC Specifications Using Promela/Xspin". *In J.-C. Grégoire, G. Holzmann and D. Peled (eds.), Procs of the DIMACS Workshop SPIN96, the 2nd Intl Workshop on the SPIN Verification System. DIMACS Series Volume 32, American Mathematical Society*, Providence, R.I., 1997
21. D. Luckham and B. Frasca, "Complex Event Processing in Distributed Systems", *Stanford University Technical Report* CSL-TR-98-754, Mar. 1998, 28 pages.
22. N. Mansurov, D. Zhukov, "Automatic Synthesis of SDL models in Use Case Methodology". *In R. Dssouli, G. v. Bochmann, and Y. Lahav (eds.), SDL'99: The Next Millenium, Proc. of the ninth SDL Forum*, Montreal, Québec, Canada, Jun. 21 - 25, 1999.
23. B. Miller, J. Choi, "Breakpoints and Halting in Distributed Programs", *In Proc. of the 8th IEEE Int. Conf. on Distributed Computing Systems*, San Jose, Jul. 1988.
24. Pattern Specification System, http://www.cis.ksu.edu/santos/spec-patterns.
25. G. Robert, F. Khendek and P. Grogono, "Deriving an SDL Specification with a Given Architecture from a Set of MSCs", *In A. Cavalli and A. Sarma (eds.), SDL'97: Time for Testing - SDL, MSC and Trends, Proc. of the eight SDL Forum*, Evry, France, Sept. 22 - 26, 1997.
26. R. L. Smith, G. S. Avrunin, L. Clarke, and L.J. Osterweil. "An Approach Supporting Property Elucidation". *In Proc. of the 24th International Conference on Software Engineering*, Orlando, FL, May 2002, pages 11-21.
27. The SMV System, http://www-2.cs.cmu.edu/~modelcheck/smv.html.
28. A. Tanenbaum. "Computer Networks". *Prentice Hall*, 1996.
29. Telelogic, "ObjectGEODE SDL Simulator Reference Manual".
30. A. Ulrich, H. Hallal, A. Petrenko, S. Boroday, "Verifying Trustworthiness Requirements in Distributed Systems with Formal Log-file Analysis". *In Proc. of the thirty-sixth Hawaii International Conference on System Sciences (HICSS-36)*, 2003.
31. P. A. S. Ward, "A Framework Algorithm for Dynamic Centralized Dimension-Bounded Timestamps", *In Proc. of CASCON* 2000, Mississauga, Ontario, Canada.