Verifying Trustworthiness Requirements in Distributed Systems with Formal Log-file Analysis

A. Ulrich[a], H. Hallal[b], A. Petrenko[b], S. Boroday[b]

[a] *Siemens AG, CT SE 1*
81730 Munich, Germany
andreas.ulrich@siemens.com

[b] *CRIM, 550 Sherbrooke West, Suite 100*
Montreal, H3A 1B9, Canada
{hallal, petrenko, boroday}@crim.ca

Abstract

The paper reports on an analysis technology based on the tracing approach to test trustworthy requirements of a distributed system. The system under test is instrumented such that it generates events at runtime to enable reasoning about the implementation of these requirements in a later step. Specifically, an event log collected during a system run is converted into a specification of the system. The (trustworthy) requirements of the system must be formally specified by an expert who has sufficient knowledge about the behavior of the system. The reengineered model of the system and the requirement descriptions are then processed by an off-the-shelf model checker. The model checker generates scenarios that visualize fulfillments or violations of the requirements. A complex example of a concurrent system serves as a case study.

1. Introduction

The challenge of deploying trustworthy distributed systems lies not only in their ever growing complexity, but also in requirements that change over time and that have to be met by the evolving system. New features added to the system must not interfere with existing ones or even prevent their fulfillment. Moreover, documentation of the system's design and its functions and performance usually degenerates over time. Thus, measures for system reengineering and support of automated regression testing become essential to maintain the system's trustworthy properties. These techniques are supported by tracing the internal communication of the system at runtime and subsequent analysis of system properties in the collected trace.

Tracing an object-oriented concurrent system consists of collecting events from object and thread creation and termination, method invocation and execution among concurrent objects, plus relevant local events from variable values in objects. Tracing must be included as an additional feature in the systems architectural design. Trace analysis converts a collected trace into a formal execution model that replicates the observed behavior of the system under test. This formal model is the starting point for model-checking of desired or undesired system properties.

The trace analysis approach is particularly a useful means for the analysis of system failures observed during integration or system testing and beyond. The analysis of failures requires the restoration of the exact system state, from where the failure originates. Concurrent systems make this restoration process especially hard to do. Moreover, nondeterminism due to concurrent communication messages may lead to system runs with unpredictable failure occurrences. In its ability to restore a systems behavior and to check for trustworthy properties exhaustively, trace analysis bears high expectations to support these fault diagnosis activities. Using trace analysis, fault scenarios in the trace are identified that guide the software developer to the origin of a system failure.

In this paper we state the requirements on a trace collected from the execution of a concurrent system under test to perform trace analysis. We show how a formal execution model can be constructed from such a trace. We present our approach to the analysis of system properties in a trace that is based on SDL, a formal description technique widely accepted in the telecommunication industries, and conventional model-checking. We develop a tool that automates the approach by interfacing trace collection techniques to a commercial SDL design and verification tool. A complex example of a concurrent system serves as a case study throughout the paper. An early version of this paper appeared as a technical report in [10]. Here, however, we report on our latest progress in the project, in particular, a case study performed with the developed tools.

The remainder of the paper is organized as follows. The next section gives an overview of related work. Section 3 describes our approach to analyze execution traces of concurrent systems. The phases trace generation, model reengineering, pattern design, and simulation are explained in detail. Section 4 presents an extensive case study, the analysis of traces from a distributed track control system, that

Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)
0-7695-1874-5/03 $17.00 © 2002 IEEE

demonstrates the feasibility of our approach. Finally, Section 5 concludes the paper.

2. Related Work

Different approaches exist for analyzing properties of distributed systems. A simple approach to perform the analysis would be to halt the execution of the system at some specified breakpoints that are defined as predicates of system's events [14]. However, the expressive power of the linked predicates is limited. In addition, the local state of each process needs to be known which is not always feasible, especially in the case of distributed systems.

Much work on trace-based analysis was done in the field of intrusion detection. The work in [12] shows such an approach that targets intrusion detection in network systems and models intrusion patterns using Colored Petri-Nets. The approach can be generalized to cover a wider range of properties, but it still lacks an implementation that shows its efficiency. In [5], flow graphs are used to represent potential communications between the processes of a distributed system. Properties, meanwhile, are represented using quantified regular expressions (QRE). This approach requires deep knowledge of tiny details in the processes of the tested system, and does not use execution traces to test the system's behavior. The approach in [16] describes the tool GrIDS to detect large scale intrusion attacks on network systems. The concept is to build activity graphs of the executions of the various processes in the system by monitoring them individually and to analyze them based on some reference rules to decide on an intrusion. The approach requires heavy means to protect the GrIDS modules themselves against attacks.

The papers [2] and [17] present an approach to visualize the collected traces. [7] and [11] describe an approach, centered on the concept of the lattice, to perform trace checking in distributed systems. Following this approach, a lattice is built, based on the monitored events and the relations between them, to represent the system under test. The lattice is then used in a model-checker to verify the behavior of the system against a desired property. In a recent work [13], a method of specifying abstraction hierarchies to define level-wise views of a distributed message-based system is outlined. This method utilizes event-pattern mappings and complex events to represent a system's behavior.

Similar to the approach in [11], we base our model on the concept of the lattice. However, we suggest going further. We describe in SDL (Specification and Description Language) each component of the system under test (SUT) involved in the given trace as a state machine, and we let the SDL simulator in ObjectGEODE build the composite (global) machine of the system (an SDL state graph) that represents the same interleavings as the lattice. This is done at runtime when the model checker verifies the given property (pattern) [10]. We believe that our approach steps further in the direction of full automation of the process especially by means of a front-end tool that builds the SDL models from the collected trace and helps the user specify the properties of interest in the GOAL language.

3. Trace Analysis Detailed

3.1. Overview

Our approach to improve trustworthiness of concurrent systems culminates in the following problem statement. Given an executed trace collected by monitoring a concurrent system under observation and a set of properties of interest, verify whether the system's behavior represented by the trace exhibits the given properties. Such properties that we support with our approach are safety and reliability properties. Other properties that indicate, for example, performance aspects could also be supported with some adaptations to the reconstructed system model.

We base our work on two fundamental concepts that reflect concurrency in communicating systems: a partially ordered set (poset) of events, where the partial order is the traditional *happened-before* relation [3], and its corresponding lattice. In fact, these two concepts are at the heart of the main existing approaches to analyze traces of distributed systems. For example, the approach of [11] considers building the lattice of the poset of the collected events and performing the verification of a pattern on the lattice using existing model checking techniques. Similar to this approach, we consider using an existing tool, rather than relying on home developed algorithms and methods for model checking. In our trace analysis approach, we

- observe an execution trace of the concurrent system that contains a list of partially ordered events for the registration of threads and objects, communication and local events,
- build an SDL model of a system from the collected trace (the trace is completed, i.e., we do a post mortem analysis of a performed system run),
- design system properties of interest using the facilities of the ObjectGEODE tool suite,
- apply the ObjectGEODE simulator to perform the checking of a given property.

We decided to use SDL because it comes with a clear formal semantics and is supported by commercial tools. Moreover, SDL is a widely accepted, standardized formal description technique in the telecommunication industries.

An SDL model of the system under observation that stands behind a collected trace reflects the following as-

pects: structure, behavior, communication, and data. The structure of the system can be modeled using the hierarchy of system/block/process/procedure statements in SDL. In our case, it is sufficient to define a system with a single SDL block that is composed of several SDL processes. The overall behavior of the system is modeled by the joint behavior of a set of communicating processes in SDL. These processes correspond to entities of the real system whose behaviors are recorded in the trace; thus making each of the processes linear, as in most cases we cannot identify any two states of it, so no cycles can be deduced.

The representation of an SDL process can be obtained by projecting the collected trace into the set of events it executes: send, receive, and local events and subsequently inserting states between them, while representing local events as SDL tasks. The communication between processes is achieved via signals with optional signal parameters representing exchanged data and channels. Input signals to a process are stored in an individual queue before they are processed modelling unknown delays in real communication channels of the system.

The property to be checked (the pattern) is expressed as an observer [9] in the GOAL language. A GOAL observer implements in fact a finite automaton with accepting states [1]. GOAL is similar to SDL, but has some syntactic and semantic differences [1]. Observers are usually described in terms of entities associated to the SDL system, e.g. objects or signals, of the tested system. This makes communication signals directly accessible for observation while other model data, e.g., variables and states, are accessible to observers using "probes". The latter represent pointers to SDL entities. In addition, GOAL allows the declaration of two types of designated states: success states and error states. Entering success states (or error states) indicates that the system respects (or violates) the property expressed in the observer. The use of the success/error convention is completely up to the user and has no formal meaning.

Once the representation of the distributed system and the property is complete, the ObjectGEODE simulator and model checker can be employed to verify whether the system satisfies the property. This is achieved preferably in the so-called exhaustive simulation mode, when the tool performs all the possible executions of the system and builds a state graph of the SDL system.

The state graph represents all the possible interleavings of events in the collected trace that respect the *happened-before* relation. To perform model checking, the tool builds the synchronous product of the observer and the specification of the system. The ObjectGEODE simulator outputs a report of the number of errors and successes encountered, and stores the scenarios that lead to the observer error and success states. An error scenario is a textual description that indicates a system execution, which leads to the property violation. Thus it should be treated with care by the developer. Scenarios are visualized as message sequence charts (MSCs) to facilitate the task of depicting the system's history required to reach a certain error or success state.

The approach based on the use of the ObjectGEODE environment can be summarized in the workflow diagram shown in Figure 1. As it is clear from this figure, the implementation efforts are reduced to building just a front-end tool to ObjectGEODE containing three main blocks:

- System reengineering tool that builds an SDL specification from a collected trace;
- Pattern specification tool that supports the process of writing analysis patterns;
- A user interface module that allows the operator to control the whole process.

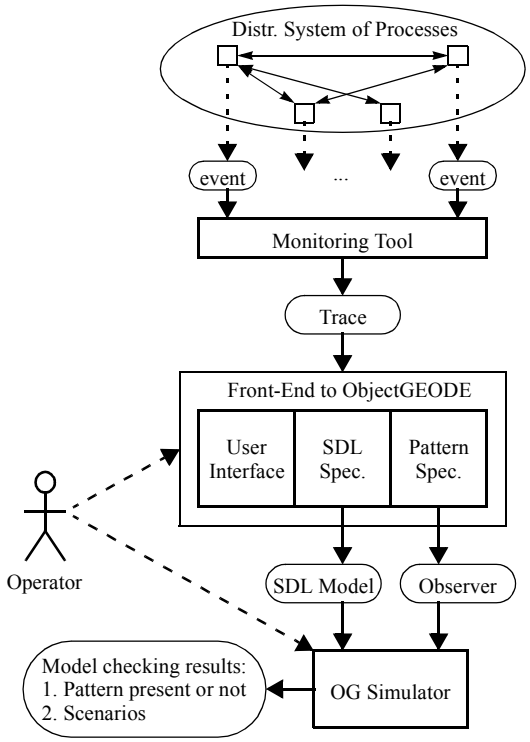

Figure 1. Workflow diagram of trace analysis.

3.2. Tracing of Concurrent Systems

Since a trace is input to our analysis approach, facilities in the system under observation must exist to create trace events at appropriate places in the source code of the system. Different methods to insert such probes exist. The most promising of them are, of course, those which work automatically. At Siemens, we explore different techniques for different platforms, such as Microsoft COM, Java RMI, CORBA and others, to do this job [8].

To be of use for trace analysis, each trace event that is a communication or local event must obey the following structure in order to assist model reengineering.

- Name and type of the event: Send, Receive, or Local
- ID of the issuing process,
- ID of the source process (for Receive),
- ID of the destination process (for Send),
- Message parameters of the event: a list of typed parameters that includes a message name and other message attributes (for Send and Receive events),
- Local parameters of the issuing process: a list of typed parameters reflecting its current state (for Local events).

Furthermore, Registration events occur that introduce newly created threads, processes or objects in a trace. These Registration events are required to determine the number of SDL processes in the model.

We assume that the local order of all events is preserved by their order of appearance in the trace. The problem of matching receive and send events is crucial to determine the partial order of events and eventually build an adequate SDL model from the recorded trace. Much depends on how the distributed system is instrumented and what exactly is monitored. We assume that in a pair of source and destination processes, each Receive event matches with a single Send event.

3.3. Model Reengineering

To be correct the SDL model has to represent the happened before relation in the trace. Three requirements must be met when constructing the model:

1. Each Send event happens before the corresponding Receive event.
2. The local order of events in each process is preserved.
3. Transitivity of the *happened-before* relation over all events in the model.

The first requirement is easily fulfilled in any SDL model since an SDL input signal can occur only after the corresponding output signal. This is defined by the semantics of SDL. Compliance to the third requirement entails that the matching of each Send and Receive event must be unique for all events in the trace. Here, the trace must contain sufficient information to obey this requirement. Requirement 2 is more difficult to satisfy. It requires that the order in any trace projection must coincide with that of a corresponding SDL process. To illustrate this problem we refer to a sample trace given in Figure 2.

Process $P1$ issues the Send event $S1$, which reflects sending the message $m1$ to $P2$. Upon receiving the message, process $P2$ issues the Receive event $R1$. Similar events are generated when $P3$ sends the message $m2$ to $P2$. Notice that events $S1$ and $S2$ are independent while $R1$ and $R2$ depend on the occurrence of $S1$ and $S2$, respectively. The trace indicates that $R1$ occurs before $R2$ regardless of the order between $S1$ and $S2$. In other words, it can be easily deduced from the trace that $S1 \le R1$, $S2 \le R2$, $R1 \le R2$, where $\le$ is the *happened-before* relation.

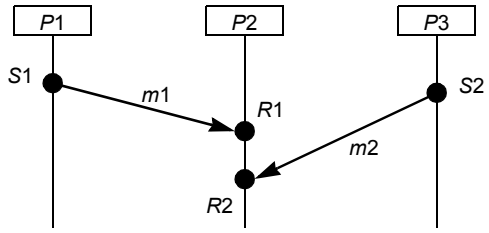

Figure 2. Sequence diagram of a sample trace.

A simple SDL process corresponding to $P1$ should just output a single output signal $m1$. The same applies to $P3$. $P2$ in turn should just receive two signals, namely, $m1$ followed by $m2$. When the ObjectGEODE simulator executes such an SDL system, it treats $S1$ and $S2$ as concurrent events, i.e., firing them in all possible orders ([$S1$, $S2$] and [$S2$, $S1$]). This means that the simulator reaches a global state where the input queue of $P2$ contains the signals $m2$ and $m1$ received in the reversed order, i.e., $m2$ precedes $m1$, contrary to what the trace prescribes. In this state, the SDL specification of $P2$ foresees the reception of $m1$ only; this process considers $m2$ as an implicit input and discards it. The resulting global state graph would then contain an execution that violates the local order perceived by $P2$.

It turns out that SDL offers a simple, elegant solution to this problem. To prevent a signal loss, SDL contains the SAVE construct that prohibits the process in its current state from consuming the declared signals from the queue. The saved signal is kept for future consumption. In our example, the use of SAVE in the state with input of signal $m1$ would prevent the SDL process from consuming $m2$ first and thus prevents the simulator from building executions that contradict to the given trace.

Therefore, we use the SAVE construct in each state of each SDL process to store all the unexpected signals that the ObjectGEODE simulator may try to put into the queue. Adding SAVE completes the reconstruction of the system model that can be obtained from the given trace. The resulting SDL model allows us to verify any system property that can be specified in the GOAL language.

3.4. Pattern Design

To address the issue of pattern design for safety and reliability properties we turn to the properties that are believed to be mostly used in practical applications. The existing research in the field has explored a wide range of properties that can be sought in distributed systems. These properties can be classified into state-based and event-based types.

Event-based patterns allow detecting simple atomic or composite events in the behavior of the SUT. State-based patterns, on the other side, formulate assertions on state variables of the processes in the SUT. The work of Dwyer et al. to build a repository of specification patterns (similar to design patterns) [4], [15] shows an effort to cover both types of properties. In our project, we build a repository of typical and frequently used specification property templates in GOAL [10].

To make the specification of patterns portable, a mapping to several formalisms (LTL, CTL, GIL, QRE, and INCA Queries) has already been provided in [15]. There are two classes of patterns in the repository (Figure 3):

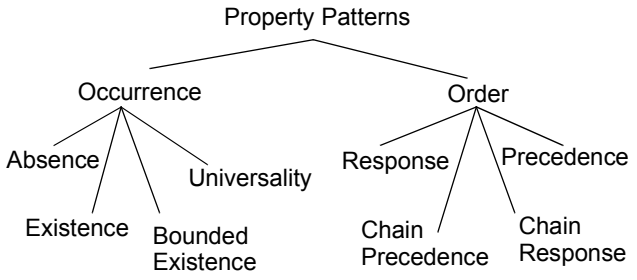

Figure 3. Classification of property patterns [15].

1. *Occurrence patterns*. They include patterns that express universality, existence, absence, and bounded existence of an event (or state) or sequence of events (or states).
2. *Order patterns*. These express relations between events (or states) or sequences of events (or states); they include precedence, response, chain precedence, and chain response.

Each pattern is defined over a scope that expresses the extent of the execution, over which the pattern must hold. Five basic kinds of scopes exist in the repository. A scope of a pattern can be:

- *Global*: the complete execution,
- *Before*: up to a given state/event in the execution,
- *After*: the part of the execution after a given state/event,
- *Between*: any part of the execution from one given state/event to another given state/event), or
- *After-until*: like between but the designated part of the execution continues even if the second state/event does not occur.

In the repository, each scope is determined by specifying state/event delimiters for the pattern: the scope consists of all states/events beginning with the starting state/event and up to but not including the ending state/event [15].

GOAL is an automata based language to specify properties that can be verified in SDL models. That is what makes GOAL capable of expressing a wide range of properties, basically all properties that are expressible by automata. In addition, the structure of GOAL is quite similar to that of SDL (see Section 3.2). This means that GOAL can be used to specify both state-based and event-based types of properties and any mixture of them. All this adds to the strengths of our approach for analyzing traces of distributed systems.

3.5. Simulation

The ObjectGEODE simulator supports three different simulation modes: random, interactive, and exhaustive. The *exhaustive mode* constructs the complete state graph of the SDL model and provides thus the functions of a typical model-checker. This model-checker can be parameterized to cope with the complexity of different SDL models. Several different state space exploration and reduction strategies are implemented.

We exploit the exhaustive simulation mode of ObjectGEODE to identify error scenarios in a trace. For this purpose, the simulator composes the synchronous product of the SDL model of trace and a property expressed as a GOAL observer. An *error* (or *success*) scenario is stored if the simulator reaches an *error* (or *success*) state of the observer. This scenario describes a path from the initial state of the SDL model, i.e. the starting state of the trace, to this particular error state. All events that occur in between are stored. An error scenario gives indications how the system under observation violates a given property. It therefore helps the developer of the system improve its quality in terms of safety, reliability, and robustness. Scenarios can be converted in ObjectGEODE to message sequence charts that offer a nice graphical representation to the user.

4. Case Study

4.1. A Distributed Track Control System

We use a real-life example to illustrate how the trace analysis approach can be applied to large size traces and concrete properties. We analyze the behavior of an implementation of a distributed track control system, which represents a complex example, designed according to [6] and verify an important system property based on traces only.

The system constitutes of a track and two or more vehicles. The track is divided into several blocks. Each block works independently and coordinates its activities with the neighbor blocks or the vehicle using synchronous message passing. The control of the overall system is decentralized, which implies high degree of concurrency. A vehicle can be in two blocks simultaneously, but communication is possible only with the block, in which the reference point of the vehicle is positioned. The move of the reference point from one block to the next is assumed to be atomic. Furthermore,

we assume that the vehicles travel in a unidirectional fashion over the various blocks, i.e., a vehicle is not allowed to change direction (Figure 4). A vehicle exchanges the following messages with a block:

- *VEnter*: The reference point of the vehicle has entered the current block.
- *VEnterComp*: The vehicle moved completely to the current block. The predecessor block is no longer occupied. It is assumed that blocks are larger than the vehicles.
- *VResReq*: The vehicle requests reservation of the successor block when it reaches the border of the current block.
- *VOccSucc*: The vehicle moved partly to the successor block.
- *VExit*: The reference point of the vehicle leaves the current block.

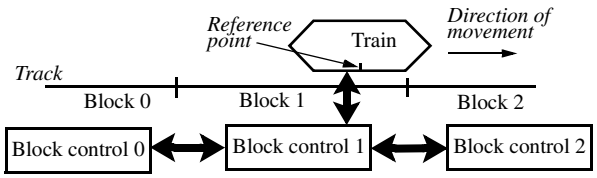

Figure 4. Block diagram of the track control system.

Blocks communicate with each other by exchanging a number of further messages. Those messages, however, are not of interest in this paper and need not be disclosed.

In detail, the system consists of two or more vehicles *V* and three or more blocks *Blk*0, *Blk*1, and so on, that are circularly connected. Each block is composed of the three components *IntCtl*, *PreCtl*, and *SucCtl*. The component *IntCtl* is responsible for the internal control of a block. It must ensure the principal safety property that no more than one vehicle may occupy a block at any time (mutual exclusion). Component *PreCtl* is responsible for predecessor control, i.e., the control of the vehicle's position and the release of the predecessor block after the vehicle moved completely to the current block. Finally, *SucCtl* is responsible for successor control that organizes the reservation of the next block once a vehicle moved to the border of the current block.

The complete specification of the track control system is given in [6]. It was verified to fulfil the mutual exclusion property stated above. We use this verified design as the starting point for our experiments, where we analyze the system property in traces recorded from a system execution.

The implementation of the track control system is done in Java. Each vehicle and each component of a block is implemented as an extended Java *Thread* object. A system comprising of three blocks and two vehicles, for example, spans therefore 3*3+2 = 11 threads. Communication among the thread objects is done via message calls. Figure 5 shows the GUI of the implementation of the track control system.

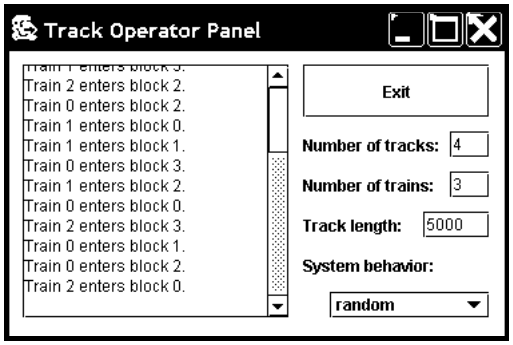

Figure 5. GUI of the track control implementation.

The original design satisfies the safety property [6], so we mutate the implementation in such a way that the property no longer holds. This mutated implementation is used for trace analysis, whose results are presented below.

4.2. Tracing of the Track Control System

To produce an execution trace of the track control system, i.e., thread creation, termination, and communication, the implementation is augmented by instrumenting the code at appropriate places. This could be done, for example, by tracing message call invocations between the Java threads that implement the behavior of vehicles or block components. A trace of events is recorded in XML format (Figure 6). An event in XML contains all information relevant for trace analysis (see Section 3.2). Note however that we use a correlation ID to identify matching send and receive events instead of providing a sender and receiver address of a message since this information is not readily available in a distributed object-oriented system.

Test runs of the mutated implementation are done for a different number of vehicles and blocks (two or three vehicles and three to five blocks) for about 60 seconds producing traces of between about 2800 and 4300 events each (see Table 1 on last page).

4.3. Model Reengineering

We build an SDL model of the system from an observed trace. For this, we use a home-developed front-end tool. We choose to build a model that relies on synchronous communication between the components since this is also the assumption taken in the system specification. In such a model, all the communications between components are instantaneous. The block diagram generated automatically from a trace of configuration #1 (three blocks and two vehicles, see Table 1) is depicted in Figure 8 (on page after next).

The SDL model includes 11 processes communicating over bi-directional channels. The dashed rectangles surround the processes that represent the controls of each

```
<event operation="Send" timestamp="2872687210179" type="Communication">
      <parameters correlation-id="9" message-id="1" thread-id="Thread[Thread-11,6,main]"/>
      <message>
            <parameter name="name" type="string" value="vEnter"/>
      </message>
</event>
<event operation="Receive" timestamp="2872687577202" type="Communication">
      <parameters correlation-id="9" message-id="1" thread-id="Thread[Thread-5,6,main]"/>
      <message>
            <parameter name="name" type="string" value="vEnter"/>
      </message>
</event>
...
```

Figure 6. Send and receive events from a recorded trace in XML notation.

block, *Blk0* for the first block, *Blk1* for the second block, and *Blk2* for the third block. *V0* and *V1* denote the processes that model the behavior of the vehicles in the system.

Each SDL process is created, by our tool, from a linear projection of the trace reflecting a component of the original system. As an example, the initial part of an automatically generated SDL process describing a vehicle is given in Figure 7. It starts in state *state0* by sending the message *VResReq* to block *Blk0*. In state *state4*, the variable assignments correspond to local events, generated by the instrumented system. The SDL process in the figure contains altogether 566 states. It terminates in state *finals*. Note also the SAVE construct (the star in a parallelogram) at the beginning of the SDL process definition.

4.4. Pattern Design

The required safety property that the reference points of several vehicles must not be located in one block, can be formulated as a GOAL observer. This property can be expressed as an event-based observer or a state-based one.

For the event-based observer, we formulate the property in terms of the relevant events as follows: No block should receive two successive *VEnter* events from two vehicles without receiving a *VExit* in between. Actually, this property can be seen as an instantiation of the absence pattern. Figure 9 shows, in GOAL language, the *event-based observer*, which can be interpreted as an extended finite state machine (this simplified version assumes the existence of only two vehicles in the track). In state *s*0 the observer waits for the occurrence of the first *VEnter* message. It then stores the received message in a variable. In state *s*1, the state reached next, the observer waits for a second occurrence of *VEnter*. If it comes from a sender process (a vehicle) different to the sender of the first message and if it is received by the same process (a block), the observer enters the *err* state that is declared explicitly as an error state. This indicates that the required property is violated.

Figure 10 shows the *state-based observer* that signals a property violation when the values of the variables indicating the block numbers occupied by the vehicles coincide. The variables *blockNo* are contained inside the vehicle im-

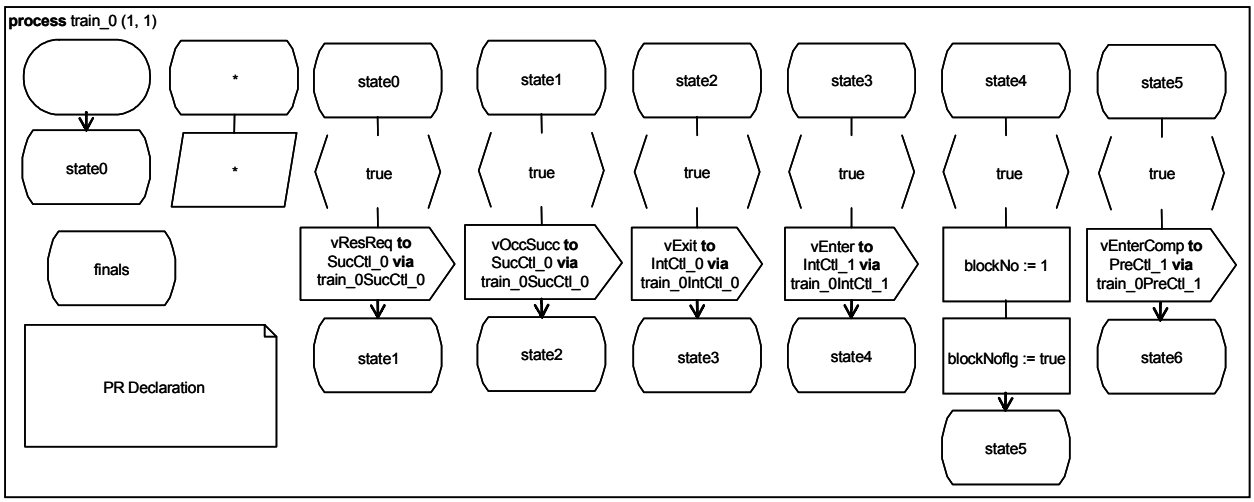

Figure 7. Initial part of a reconstructed vehicle SDL process.

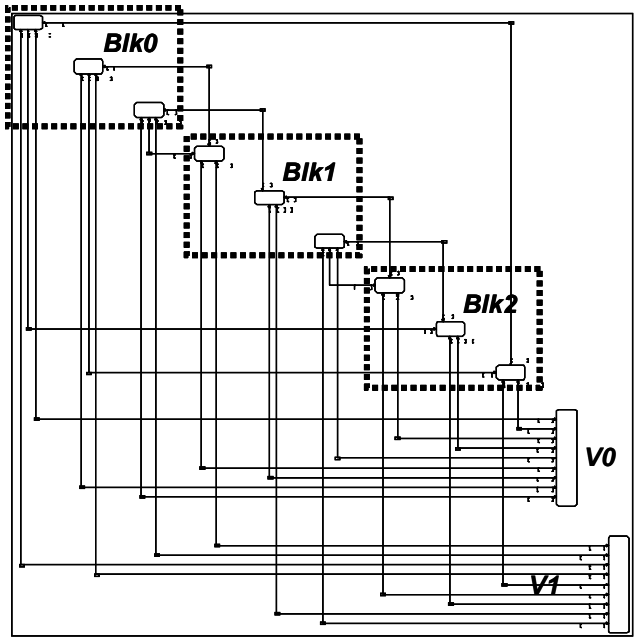

Figure 8. Block diagram of configuration #1.

plementations and are reported in the trace by corresponding local events. The variables *blockNoflg* are inserted during the model reengineering process (cf. Figure 7). Both observers can be used to detect property violations in the trace as it is discussed in the following Section.

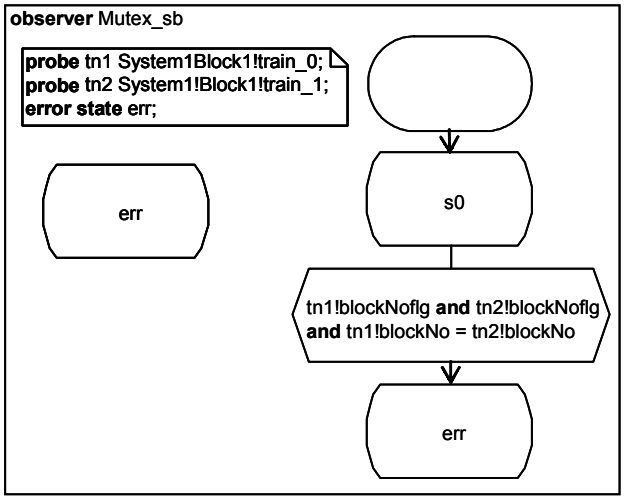

Figure 10. State-based observer of the mutual exclusion property.

4.5. Simulation

We simulate the reengineered system model with each observer in ObjectGEODE to verify whether the trace exhibits any violation of the property, i.e., whether it is possible to reach an error state of the observer on a path starting in the initial state of the system. Such a path is represented in an error scenario generated by ObjectGEODE. The simulation

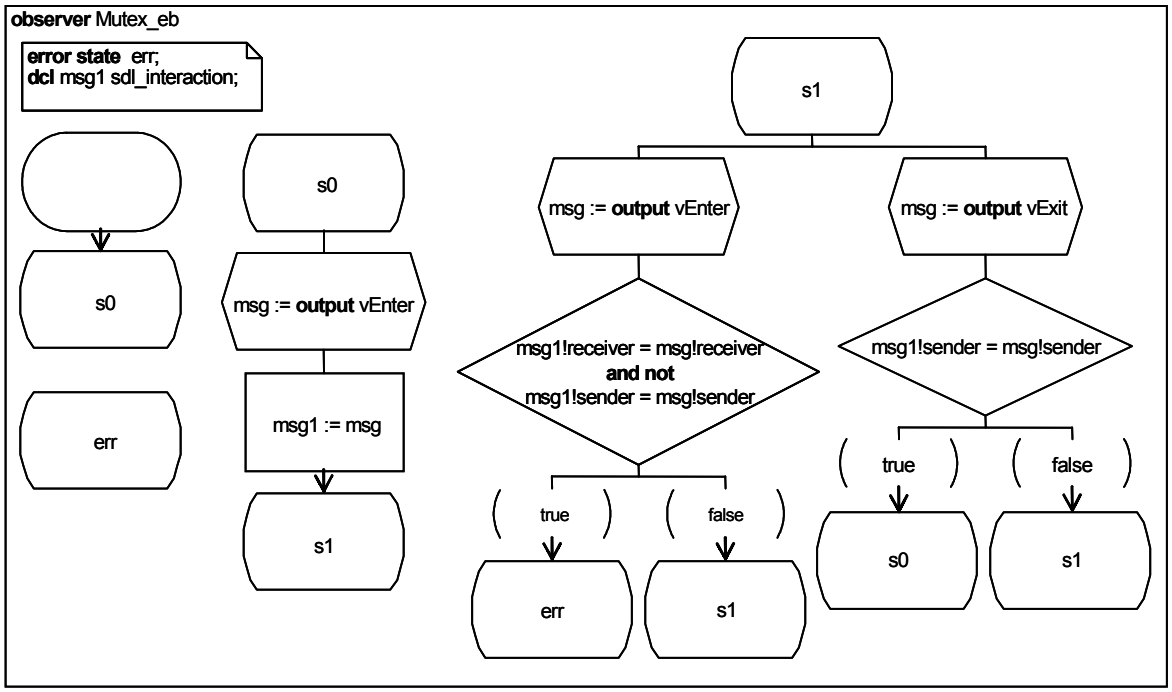

Figure 9. Event-based observer of the mutual exclusion property.

of an erroneous system can be terminated either as soon as a first error scenario is generated or when no new scenario could be obtained (complete simulation). The results of the complete simulation for different system configurations and observers are shown in Table 1. To generate a first error scenario the simulation takes only few states to explore, e.g., in configuration #5, only 82 states are checked with the event observer. Note that both observers lead to different number of error scenarios. The reason is merely technical. The event-based observer handles the events *VEnter* of the vehicles, which precede the local events exploited by the state-based observer in the local order of the corresponding processes. Clearly this leads to more interleavings being allowed in case of the state-based pattern, and subsequently more states to explore, transitions to fire, and scenarios to record. The experimental results indicate that the approach scales well to the length of a trace, however, certain filtering options have to be implemented in the front-end tool to alleviate the limitation of the simulator.

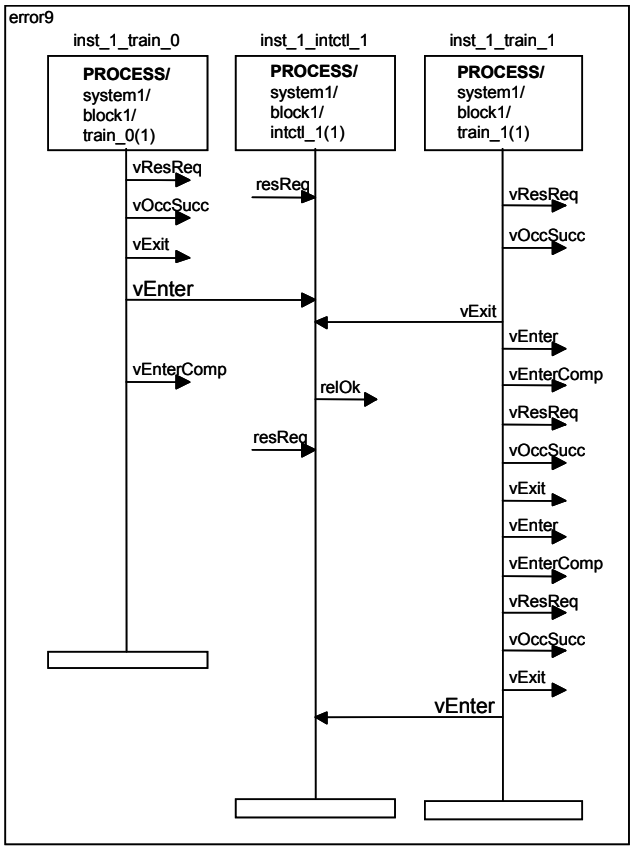

Figure 11. An error scenario found in configuration #1.

ObjectGEODE allows further to visualize the simulation results as message sequence charts (MSCs). Moreover, we can filter out any irrelevant processes from the generated MSC that do not participate in a particular error scenario. This adds to the clarity of visualization and the ease of interpretation. An MSC that depicts an error scenario found involves two vehicles that want to access *Blk0* simultaneously. It is shown in Figure 11. This visualization of the error allows one to trace a system failure back to its origin.

5. Conclusions

We proposed an approach to property verification in execution traces from distributed systems to improve their trustworthiness. The approach takes into account the usual lack of formal design specifications in practice. Therefore it attempts to remodel the system behavior from a trace that contains executed events of threads/object registration, send and receive events of messages and local events. The approach is implemented on basis of SDL, a well-established formal description technique and a commercial off-the-shelf model checker that reduces the cost of the development of proprietary tools.

To apply our approach to trace analysis in a project, a user does not necessarily need to be involved in the SDL technology behind it. The input is just an appropriate trace of recorded events preferable in XML format. Properties of interest can be designed together with an expert of the ObjectGEODE tool set. This process is assisted by our library of predefined basic property templates in GOAL. Last but not least, the generated scenarios are represented in a graphical notation that is easy to understand by any software developer. The approach is currently evaluated at Siemens to apply it in other industrial applications.

6. 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, September, 1995, Elsevier Science Publishers B. V. (North Holland), pp. 359-372.

[2] 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, November 1993.

[3] K. M. Chandy, L. Lamport, “Distributed Snapshots: Determining Global States of Distributed Systems”, ACM Transactions on Computing Systems 3(1), pp. 63-75, February 1985.

[4] 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.

[5] M. Dwyer, L. Clarke, "Data Flow Analysis for Verifying Properties of Concurrent Programs", In Proc. of ACM SIGSOFT'94, New Orleans, LA, USA, 1994.

[6] S. Fischer, A. Scholz, D. Taubner, "Verification in process algebra of the distributed control of track vehicles – a case study"; 4th International Conference on Computer-aided Verification (CAV'92); Montreal, Canada; 1992; pp. 192–205.

[7] 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.

[8] K. Grabenweger, E. Reyzl, H. Sauer, "Trace-based testing of middleware": 5th Conference on Quality Engineering in Software Technology; Nürnberg, Germany; 2002.

[9] 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.

[10] H. Hallal, A. Petrenko, A. Ulrich, S. Boroday, "Using SDL Tools to Test Properties of Distributed Systems", Workshop on Formal Approaches to Testing of Software (FATES) in affiliation with CONCUR 2001; Aalborg, Denmark; BRICS Technical Report NS-01-4, August 2001.

[11] 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, June 1994.

[12] S. Kumar, E. Spafford, "An Application of Pattern Matching in Intrusion Detection", Technical Report 94-013, Purdue University, Department of Computer Sciences, March 1994.

[13] D. C. Luckham and B. Frasca, "Complex Event Processing in Distributed Systems", Stanford University Technical Report CSL-TR-98-754, March 1998, 28 pages.

[14] B. Miller, J. Choi, "Breakpoints and Halting in Distributed Programs", In Proc. of the 8th IEEE Int. Conf. on Distributed Computing Systems, San Jose, July 1988.

[15] Pattern Specification System, http://www.cis.ksu.edu/santos/spec-patterns/.

[16] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoaglan. K. Levitt, C. Wee, R. Yip, D. Zerkle, "The Design of GrIDS: A Graph-Based Intrusion Detection System", Technical Report, Department of Computer Science, University of California at Davis, January 1999.

[17] P. A. S. Ward, "A Framework Algorithm for Dynamic Centralized Dimension-Bounded Timestamps", In Proc. of CASCON 2000, Mississauga.

Table 1. Simulation results of different system configurations.

Configuration	**#1**	**#2**	**#3**	**#4**	**#5**
# Blocks	3	4	4	5	5
# Trains	2	2	3	2	3
# Threads	11	14	15	17	18
# Events	3,490	3,341	4,332	2,804	3,860
Event-based Observer					
# States	5,479	44,353	130,029	356,499	1,044,335
# Transitions	27,304	285,055	891,479	2,819,251	8,709,147
Duration	0 mn 4 sec	0 mn 34 sec	2 mn 6 sec	4 mn 50 sec	19 mn 40 sec
# Errors	9	27	65	81	195
State-based Observer					
# States	13,676	45,610	193,746	447,228	1,318,988
# Transitions	44,485	288,446	1,084,639	3,131,122	9,724,617
Duration	0 mn 7 sec	0 mn 35 sec	2 mn 52 sec	5 mn 39 sec	22 mn 14 sec
# Errors	564	75	1959	2772	5988