TU

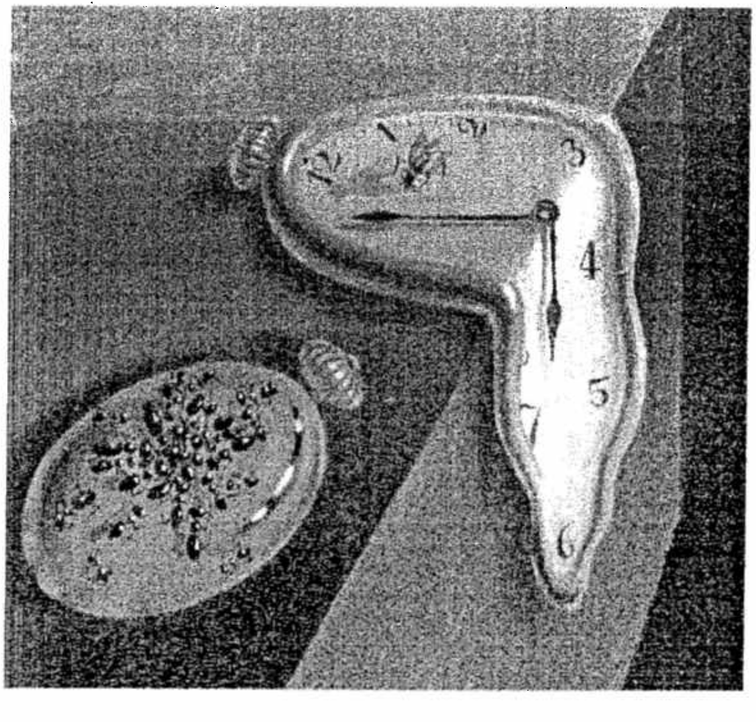
Institut für Automation
Abt. für Automatisierungssysteme

Technische
Universität
Wien

Projektbericht Nr. 183/1-41
Mai 1994

Analyzing Real−Time Systems Using TATs and the Specification Language Gold Mine

S. Stöckler



Ausschnitt aus: Salvador Dali, "Die Beständigkeit der Erinnerung"

Projekt **VTA**

Analyzing Real-Time Systems Using TATs and the Specification Language GOLD MINE

Transforming TATs Using GOLD MINE

Stefan Stöckler

Technical Report to Version 2.0 20-2-94

FWF Projekt Nr. P8390-PHY

Contents

List of Figures

Abstract

Monitoring concurrent systems by observing certain well defined events with respect to their occurrence order is an accepted way of investigating their behavior. To analyze real-time systems it is necessary to keep track of the occurrence times and possibly additional information (called attributes), which is extracted from the system at occurrence time. To accomplish this task, we introduced *Timed Attributed Event Traces (TATs)* as a means to describe the observed system behavior in a concise form ([Stö93]).

In this paper will introduce sematics and syntax of the specification language GOLD MINE which allows to define operations on TATs in a flexible but nevertheless simple way. Using GOLD MINE, a user can specify particular high-level viewpoints on the system activity appropriate for the analysis of a certain timing problem. This task is accomplished by filtering out meaningless events and by combining certein sets respectively sequences of events to form new events. The semantics of GOLD MINE is defined strictly formally by using the calculus of TATs introduced in this paper. (This work is part of the project **VTA** (*Versatile Timing Analyzer*[1])).

[1]Supported by the "Fonds zur Förderung der wissenschaftlichen Forschung" under grant P8390-PHY

Frequently Used Symbols

a, b, c, x, y, z	Event identifiers
E_a, E_c, Σ	Sets of event identifiers
$ts, ts', ts'', \ldots, ts_0, ts_1, \ldots$	Time slots of TATs
$\omega, \omega_0, \omega_1, \ldots$	Occurrences of events
$\mathcal{O}$	Set of all occurrences
$\tau, \tau', \tau'', \ldots, \tau_0, \tau_1, \ldots$	(Timed Attributed) Event Traces
$\alpha, \alpha', \alpha'', \ldots, \alpha_0, \alpha_1, \ldots$	Automata, state-transition systems
A_T	Set of automaton templates
A_I	Set of automaton instances
$s, s', s'', \ldots, s_0, s_1, \ldots$	States of state-transition systems
S	Set of states
S_{term}	Set of terminal states $S_{term} \subseteq S$
$t, t', t'', \ldots, t_0, t_1, \ldots$	Transitions of state-transition systems
T	Set of transitions
Cond	Set of conditional functions
Act	Set of actions
$m, m', m'', \ldots$	Memory structures of automata
$p, p', p'', \ldots$	Parameter arrays of automata
$o, o', o'', \ldots$	Output lists
$pri()$	Priority function
$major\_pri(), minor\_pri()$	Priority functions
cl	Clock variable of a CEG
$f, \mathcal{F}$	Functions defined by state-transition systems
$g, \mathcal{G}$	Functions defined by Compound Event Generators (CEG)

1 Introduction

Designing both hard and soft real-time systems is a responsible job and the intimate knowledge of the timing properties of existing real-time systems is essential (c.f. [Sch93]). To obtain the necessary information, which cannot be derived in an analytically way because a determination is not possible, the system in question has to be monitored.

Monitoring can be defined as extracting dynamic information concerning a (computational) process, as this process executes. This can be performed by identifying, observing, and (possibly) recording characteristic events in the system under study. Therefore, monitoring provides an indication what happened, thus serving as a prerequisite to ascertaining why it happened.

The analysis of the *timing behavior* of real-time systems can be performed in a three step approach:

1. Monitoring the system's behavior during normal operation by observing the events occurring (called *primitive events*). To allow a meaningful analysis attributes can be attached to each occurrence of such an event. This monitoring yields an event trace containing low level information characterizing the observed activity;

2. Transforming the representation of the observed behavior by constructing new events on the basis of the events already in the trace, i.e. change the level of abstraction from primitive events and raw data to a suitable point of view.

3. Analyze the event traces with a suitable tool, e.g., a data-flow systems. This includes the computation of moving averages, determine statistical measures like mean values, standard deviations, or distribution functions.

This approach is known as *Event Based Behavioral Abstraction (EBBA)* invented by Bates and Wileden ([BW83]) to debug concurrent systems. Applying this method produces information describing the system under study at various levels of abstraction each of them adequate for the analysis of certain characteristics.

To analyze real-time systems it is not sufficient to keep track only of the relative ordering of events but also their occurrence time and possibly additional information (called attributes), which is extracted from the system at occurrence time, has to be observed. To accomplish this task, we introduced *Timed Attributed Event Traces (TATs)* as a means to describe the behavior of the monitored system in a concise form ([Stö93]).

In this paper we will show, how Timed Attributed Event Traces can be used in combination with the ideas of the Event Based Behavioral Abstraction Approach to analyze timing properties of real-time systems. In order to integrate the two approaches, a calculus of TATs (to obtain higher levels of abstractions) has to be defined. This framework should meet the following requirements:

Flexible. There should be no (meaningful) operation on a TAT that cannot be specified.

Portable. The elements of the calculus should be independent of particular applications, hardware-, or software-platforms.

Reusable. Any specification of a transformation represents a measurement specification (e.g. interval duration) which should be reusable with other applications.

User friendly. Using the calculus of TATs should be easy to learn and easy to use.

We will introduce the specification language GOLD MINE which allows to define operations on TATs in a flexible but nevertheless simple way. Using GOLD MINE, a user can specify particular high-level viewpoints on the system activity appropriate for the analysis of a certain (timing) problem. In this paper we will show that GOLD MINE meets the requirements listed above. The paper is organized as follows:

In Section 2 we will investigate related models and their contribution to this paper. Section 3 discusses the requirements of an event-specification language as a consequence of Section 2. In Section 4 the basic concepts of our approach (state-transition systems with memory) are introduced. The notion of *Compound Event Generators*, which is the object-oriented extension of the system described in Section 4, is defined in Section 5. The specification language GOLD MINE, which is a realization of these concepts, and its applications are discussed in Section 7.

2 Related Work

Many papers on the topics *trace theory* and *monitoring and debugging of concurrent systems* have been published in recent years. They span a range from pure theoretical work to very specific application oriented publications. A good survey on debugging and monitoring systems can be found in [MH89]; some of the following trace-transforming systems are discussed there, also.

This section provides an overview of the publications on trace theory and system monitoring relevant to our work. We present the approaches and investigate how they fit to our requirements. Most of the space is dedicated to the *Event Based Behavioral Abstraction Approach* and the corresponding event definition languages which serve as base for our specification language GOLD MINE.

2.1 Trace Theory

Trace Theory emerged from the need of a formal description of concurrent systems. The basis of this theory, called the alphabet (A), is the set of events the system under study can engage in. A trace is a sequence of symbols of this alphabet each standing for a particular observed occurrence of an event. Not all words over the alphabet (denoted by A^*) are traces of a particular system.

C.A.R. Hoare introduced in [Hoa85] a formal system, called *Communicating Sequential Processes (CSP)*, which allows to describe the behavior of every single processes of a concurrent system by the events it may engage in. A system of concurrent processes cannot be described in terms of a single trace but by a set of possible traces. The descriptions of all participating processes together with formal rules of synchronization can be used to produce the corresponding traces of a particular system.

Trace Theory introduced by A. Mazurkiewicz (see [Maz86] or [AR88] for a survey) attacks the problem of transforming traces with respect to the characteristics of the underlying system. In this theory equivalence classes of traces are identified by using a equivalence relation induced by the *independence relation.* According to Mazurkiewicz a system is characterized by its alphabet A and an independence relation I. Like in the theory of CSPs the alphabet A is the set of all events the system can engage in. The independence relation I is a binary, symmetric relation and a pair (a, b) of events $a, b \in A$ is element of I if and only if they are independent in the system under study. The equivalence relation $\equiv_I$ is defined in the following way:

$$ab \equiv_I ba \quad \leftrightarrow \quad a, b \in A \text{ and } (a, b) \in I \tag{1}$$

and two traces $\tau_1, \tau_2 \in A^*$ are equivalent ($\tau_1 \equiv_I \tau_2$) if and only if there exists a finite sequence of traces

$$(t_1, t_2, \ldots, t_n), \qquad n \geq 0, \tag{2}$$

such that $\tau_1 = t_1$ and $t_n = \tau_2$ and for each i, $(1 < i \leq n)$ there are traces $u, v \in A^*$ and symbols $a, b \in A$ with $(a, b) \in I$, with

$$t_{i-1} = uabv, \quad ubav = t_i. \tag{3}$$

The quotient algebra $(A^*, \circ, \epsilon)/\equiv_I$ with the concatenation operation $\circ$, which may be commutative for some symbols, is a monoid.

Although, trace theory provides a formal and well understood system for investigating the behavior of concurrent processes, it is not quite realistic because of the lack of attributes. We think that in many cases the independence relation I (and as a result of this the dependence relation $D = A \times A \setminus I$) cannot be formulated without respect to certain attributes of events. For example, the sending of a message must happen before the receiving of the same message, but sending and receiving different messages is independent. Consider the following example:

Example 1
A producer task issues s-events every time it sends a message to a consumer task, which produces a r-event on the reception of each message. The following two traces may be observed while this system is executing:

$$\langle \text{s r s s r s r r} \rangle \qquad \text{and} \qquad \langle \text{s s s s r r r r} \rangle$$

The traces could be produced by different executions of the tasks given above using the same set of input data! The first trace may be interpreted as the result of running the two tasks in round robin fashion and the second trace may be caused by running the producer task at a higher priority than the consumer task.

Although these traces are obviously equivalent in their effect in the real world, they are not equivalent in trace theory because the pair (s, r) must be in the set D! This problem is imposed by the lack of attributes. Only those (s, r) pairs concerning the same message are causally related and therefore must not be swapped! □

Not only because of the lack of attributes and timing information in the model but also because their is no means of abstraction (creating new events on the basis of existing ones) these two theories are not suitable for analyzing real-time systems.

2.2 Event Based Behavioral Abstraction

A more practical approach is used by Bates and Wileden (*Event Based Behavioral Abstraction* [BW83], [Bat88]) to debug concurrent systems. They invented an *Event Definition Language (EDL)* to specify transformations on an existing event trace. The aim of their approach is to allow the user to look at a system from different viewpoints appropriate for analyzing a particular problem. To accomplish this task, already existing events are combined (clustered) or filtered to obtain suitable event traces. The trace-transformation operations are specified in EDL, which allows the user to formulate the clustering and filtering operations using an *is-clause*, a *with-clause* and a *cond-clause*.

In the *is-clause* the user specifies the timing dependencies of the events constituting the high-level event. This is done by using an event expression which combines events with the following operators ([BW83]):

catenation	$'$	$a\,'\,b$	a b-event must follow an a-event.
shuffle	$\hat{}$	$a\,\hat{}\,b$	both events a and b must occur, but the ordering is not relevant.
alternation	$\mid$	$a \mid b$	either an a- or a b-event must occur.
repetition	$*$	a^*	event a may occur zero, one, or more times.
repetition	$+$	a^+	event a may occur one or more times.

In [Bat88] a slightly different notation for event expressions (with the same meaning as the one above) is introduced.

In the *with-clause* the user specifies the attributes of the event being defined. The attributes bound to the instances of event expression (is-clause) constituents are used as operands in the assignment expressions which determine the values of the new attributes.

The *cond-clause* is used to define relational expressions over the attributes of event expression constituents to places constraints on them. Only events satisfying the restrictions in the cond-clause take effect in the event-expression. This realizes the above mentioned filtering mechanism.

In addition to the problems discussed by Bates and Wileden themselves, there are the following unanswered questions:

- When does a high-level event specified by an is-clause like $a\,'\,b^*$ occur? In general, every event expression ending with a repetition operater is not determined. Event expressions like a^* can occur never or even ever!

- It is not specified if the event sequence acb satisfies the EDL expression $a'(b \mid (c'\,d))$. In [Bat88] the notion of anonymous subexpressions, which are matched against the event trace simultanousely by the recognizer, is introduced. Therefore, the example expression can be divided into $(a\,'\,S_1)$ with $S_1 = (b \mid S_2)$ and $S_2 = (c\,'\,d)$. In this case the sequence acb matches S_1 and thereby $(a\,'\,S_1)$. With this definition there is no chance (if neccessary) to define that the sequence $(c\,'\,d)$ should be 'atomic'.

- Another problem arises with the access to attributes. If there are more instances of one particular event are required in an is-clause, they must be indexed to distinguish their attributes in the with-clause. It is not specified what happens if two (or more) events occur at (exactly) the same time, i.e. they have the same timestamp.

- Moreover, it is not defined, how the attributes of an event with an repetition operator can be accessed. The indexing operator works fine with single occurrences like $a[1]\,'\,b\,'\,a[2]$ but how can the attributes of the last (first) occurence of the event b in the expression $a\,'\,b^{+}\,'\,c$ be addressed in the cond-clause or the with-clause? There are neither syntactical rules to specify the desired attributes nor semantical definitions to solve this problem.

- Event definitions written in EDL are not portable because they are formulated using specific events existing (probably) only in the actual system.

- The implicit timestamping of instances looks rather mystical. It is allowed to read the timestamps to determine new attributes, but it is not mentioned if it is legal in EDL to assign a value to the timestamp attribute of a high-level event. The ability to determine the timestamp of a newly generated event instance would introduce tremendous problems (probably impossible to solve) to the event recognizer.

Although this list looks rather long, we think that this is an excellent approach and, therefore, we adopted it for our system.

The approach presented in [WH88], [HW88], and [HW90] is based on the one of Bates and Wileden. *Primitive events* produced by the monitored system and *global events* defined by the user are distinguished. Although only the relative timing of the events (according to Lamports *happend-before relation*, [Lam78]) is observed, primitive events are timestamped with a virtual occurrence-time. By definition, a global event inherits the occurrence-time of the last event satisfying it. The event specification language is augmented with a negation operater @ to indicate that an event should not occur in a sequence of other events.

In [Ros91] the *Task Sequencing Language (TSL)* is used to specify concurrent systems. TSL is an event description language like (and inspired by) EDL of Bates and Wileden. As an additional feature *properties (global monitor variables)* can be set or tested at the occurrence time of events. The problem of concurrent testing and setting of the same properties in different event descriptors is solved by the definition that all testings must be performed before new values are assigned. In TSL only the *happend-before relation* ([Lam78]) is relevant for the ordering of events, no timing information is available (at least not to the user). The language is designed to be integrated in Ada source code and during runtimte the monitor checks the real system behavior on the specification.

Data Path Expressions (DPEs) are used in [HK90] and [PHK91] to model the intended behavior of concurrent systems. In this approach there are two sequencing operators: one for sequences of events which are causaly dependent and another to express that one event preceeds the other one but both events are independent of each other. The model is designed to preserve only the relative ordering of events according to Lamports happend-before relation; no timing information can be accessed.

A similar approach is presented in [RRZ88]. A *composition language* for defining compound events using a Boolean predicate interrelating the state of several other events is provided. According to Rubin et al. this language gains the full power of *path rules*.

2.3 Event-Action Paradigm

Lumpp et al. view the act of monitoring as the association of (process level) events in the monitored system with actions to be taken in the monitoring system ([LCSM90] and [MLC90]). This imposes the (useful) distinction between active processes (in the target system) and reactive processes (in the monitoring system). The reactive processes perform one or more actions in response to an event occurrence.

The user specifies which events are to be observed, the actions to be taken upon the detection of events, and optionally a binding of events to actions. This binding may be changed dynamically. The montoring system also allows to define compound events using the sequencing operators *unordered*, *sequential*, *strictly sequential*, and *and*.

For the monitoring system of their instrumented testbed Bhatt et al. (in [BGR87]) followed the event-action paradigm, too. They also consider compound events as a suitable mechanism of abstraction, but this is not amplified.

The syntax introduced in [HK90] provides a construct to attach a statement block to every event. These statements are treated as one action which is to be exectuted on every occurence of the corresponing event.

From our point of view, the idea of a strict seperation of testing an event expression and executing actions on the occurrence of events is essential. This enables the user of a monitoring system to specify arbitrary changes of the monitor variables (representing the actual state) with *every single occurrence* of an event and, therefore, eliminating the problem of accessing attributes of events with a repetition operator mentioned above.

2.4 Relational and Logical Approaches

Snodgrass argues ([Sno88]) that a *historical database*, an extension of a conventional relational database, is an appropriate formalization of the information processed by a monitor of a complex system. In this approach the user is not dealing with data actually stored in a database, moreover, the user is presented the *conceptual view* that the dynamic behavior is available as a collection of historical relations.

A global clock in the system is assumed, so that every event can be timestamped on its occurrence. Events are viewed as relations in the database and can be divided in *event relations* and *interval relations*. For each relation Snodgrass distinguishes between implicit (time) and explicit (user defined) attributes. The mechanism of abstraction is to derive new relations from existing ones using Snodgrass' augmented version of the relational tupel calculus query language Quel called TQuel. TQuel provides a more comprehensive semantics by treating time as an integral part of the database.

In TQuel the occurrence time of an event/relation can be specified with the *valid at*-clause. This imposes problems to the monitor because the time of validity of an event can be defined to be the occurrence time of the first constituing event of a sequence of necessary events. The monitor can produce the compound event if the last event in the sequence has occurred and, therefore, must generate a new event with a timestamp of the 'past'. This implies that all other relations depending on this event have to be kept in the database for a long time (possibly forever). For such relations the database can no longer be conceptual.

Unlike a conventional database query which refers to stored data a TQuel qurey operates on dynamic data. It is possible to formulate and start a query at every moment in time while the monitor is running, but it is not defined how a (already started) query, which is no longer of interest, can be stopped.

In [GYK90] a monitoring system using *linear time temporal logic* is presented. The temporal operators $\Box$ (always), $\diamond$ (eventually), $\circ$ (next), and φ (until) are available to specify assertions which are checked by the *Temporal Assertion Checker* on the program history.

We think that both approaches do not meet the requirement of userfriendlyness because not all users of debuggers are familiar with the relational tupel calculus and even less operators of monitors are used to express their ideas of a system in terms of temporal logic.

2.5 Other Approaches

Lin and LeBlanc present in [LL89] an event based debugger designed for the *Clouds Oprating System* (any object/action system, resp.). They argue that the event seems to be the superior primitive to use and, therefore, no mechanism of abstraction is provided with their monitor. Debugging of concurrent systems is claimed to involve two phases: (1) observing the system from a general view to identify suspicious objects and (2) take a closer look at these objects.

A more versatile approach is taken by Joyce et al. ([JLSU87]) who developed a monitor for the message-based operating system *Jade*. Events are messages sent to monitoring processes called *channels*. *Controllers* (also part of the monitoring system) are responsible to obey the correct ordering of the observed events. And, last but not least, the event stream is passed to presentation processes called *consoles*. Compound events can be realized by implementing a suitable console and connect it to the monitor.

Hofmann et al. introduce in [HKLM87] an *Event Trace Description Language (EDL)* which allows to describe types and order of attributes of each individual event. This language provides neither means to specify higher-level events as combination of existing ones nor means to define analysis functions.

3 Requirements for Event-Specification Languages

In this section we try to characterize our ideas of a monitoring system and a corresponding language which provides methods for abstracting and analyzing the information extracted from the system under study. As stated in the introduction and in [Stö93] TATs are a suitable and comprehensive form of representing the observed system-behavior. For this reason we will introduce the specification language GOLD MINE as an appropriate means to define transformations on TATs with the aim to obtain TATs characterizing the monitored system at a higher-level of abstraction.

Apart from the fact that we consider TATs as a suuitable form of abstraction of an observed event stream and, therefore, as basis for our language, every specification language intended to realize the EBBA-Approach should meet the following requirements (excepted the last one). This list is an extension to the list given in Section 1.

Abstraction. Obviously, the language should provide an appropriate method to define higher level events as a means for obtaining a more abstract point of view. Moreover, the ability to construct multiple levels of abstraction at the same time is desirable. In order to realize these different points of view, it is neccessary to combine events iteratively, using both primitive events and already defined compound events as constituents.

Flexible. There should be no (meaningful) transformation from low level events to a higher level of abstraction that cannot be expressed. This includes the possibilities of computing new attributes as well as clustering event sequences to form a new event. EDL ([BW83]), e.g., does not meet this requirement because it is not possible to access the attributes of events with an repetition operator ($^+$ or *, see Section 2.2).

Userfriendly. The language should be easy to learn and easy to use. This can be achieved, for example, by augmenting an existing and well known programming language with the neccessary constructs.

Problem oriented There is no need for a new general purpose programming language; moreover, the language should be designed only for specifying event transformations following the EBBA-Approach and the event-action model (see below).

Portable. Like in any other (programming) language all dependencies of the specification language on the underlying system (monitored and monitoring) should be avoided. No re-coding should be neccessary if a measurement specification is used with different hardware/software platforms.

Reuseable. Trace transformations represent measurement specifications (e.g. interval duration) which should be reuseable with the same or other applications. This implies, that specifications must not be formulated with respect to application oriented information (for example, names of existing events). Most of the languages mentioned in the previous section violate this rule and are therefore not reuseable.

Event-action model. If an action can be specified for every constituent event in a compound event description, problems with the access to event attributes (like in Bates' EDL) can easily be avoided. The user specifies exactly when to store event attributes in monitor variables and when to use the values of this variables. The actions change the monitor state, which may be tested in conditional clauses of event specification.

Independency of the monitored system. Each transformation specification can be interpreted as a measurment description and for this reason, the specification language should not depend on the monitored system and its application. This includes all parts of the observed system: hardware, software, and operating system(s). There should be a strict seperation between the language for defining the *primitive events*, which must depend on the monitored system to a certain extent, and the language for specifying *high level events* (*compound events*).

Besides these general requirements, we demand the following two features concerning the application of the specification language.

Application oriented presentation. Although the language itself should be independent of the monitored system to keep it reuseable, it should be possible to assign (dynamically) user defined names to events. This makes it easier for the user to keep track of the different levels of abstractions and allows an application oriented presentations of the results using names the user is familiar with.

Interactive controlable. The system should be designed to allow measurements to be started and stopped during runtime, so that the user has the possibility to interact with the monitoring system. Activating and deactivating of transformations reconfigures the observing system without any interferrence with the observed system.

The last requirement is implied by the special issue of our application, but it seems to be important for other analysis problems, too.

Real-time capability. Because our aim is to analyze real-time systems, it is obvious, that timing information has to be attached to the event traces. This timing information has to be preserved and processed by the event transformations.

Note, that some of the entries listed above look contradictory. And in fact, designing an event specification language is a process of deciding which requirement is more (or less) important.

The language GOLD MINE was designed to specify transformation rules for TATs. In [Stö93] it was shown, that Lisp programs could be used to do this job, but we think, that Lisp is not a well known programming language and, therefore, violates the rule entitled 'userfriendly'. The fact that GOLD MINE only operates on TATs is due to the rule 'independence of the monitored system'; there is no way to define primitive events in GOLD MINE, but a TAT containing only primitive events is treated exactly in the same way as a TAT consisting of compound (and primitive) events.

4 Basic Concepts of GOLD MINE

When we decided to design a new event description language, which should combine the advantages of the EBB-Approach and the Event-Action Model, we choose *state-transition systems* as basic element. The special merits of these systems are their simple semantics, their flexibility, and that they are better known than query languages for relational databases or regular (event) expressions. In the following section we will introduce the special kind of state-transition systems we use.

4.1 State-Transition Systems

Before we start: Because of the fact that the state-transition system is not the only element in our approach and because we will introduce the different mechanisms step by step, some of the following explanations will leave open questions which, however, will be answered in subsequent sections.

Definition 1
A *state-transition system* or *automaton*[2] is defined by the tuple $\langle S, T, s_0, S_{term}, \text{Cond}, \text{Act}, E_a \rangle$ where

- S is the set of states,
- T is the set of transitions ($T = \{(s, e, c, a, s') \mid s, s' \in S, e \in E_a, c \in \text{Cond}, a \in \text{Act}\}$),
- $s_0 \in S$ is the start state,
- $S_{term} \subseteq S$ is a set of terminal states,
- Cond is a set of conditional functions (returning one of the Boolean values TRUE or FALSE),
- Act is a set of actions,
- E_a is the set of event identifiers the automaton will accept.

$\square$

[2]We will use the terms *state-transition system* and *automaton* as synonyms.

Definition 2
The semantics of these definitions are as follows (we refer to a transition $t \in T$ with $t = (s, e, c, a, s')$, which is shown in a simple graphical notation in Figure 1):

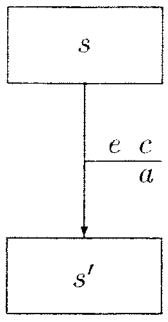


Figure 1: Simple automaton

1. A state-transition system accepts only events which have identifiers which are elements of the set E_a.
2. An automaton is always in a particular state $s \in S$.
3. A state-transition system changes from state s to state s' if and only if the event identifier of the input element (which is an event consisting of an identifier and an attribute list) is e, there is a transition $t \in T$ with $t = (s, e, c, a, s')$, and the conditional function $c \in \text{Cond}$ applied to the attributes of the input element evaluates to TRUE.
4. When the transition from s to s' $((s, e, c, a, s') \in T$ is performed, the action $a \in \text{Act}$ is executed (the attribute list of the event which caused the transition is passed as parameter to the action).
5. If the system changes to state s' and $s' \in S_{term}$, a new event is produced and inserted in the output list (which contains only newly produced occurrences).
6. If there is no transition from the actual state marked with the particular event identifier or if the conditional function named in the transition evaluates to FALSE, the input element is ignored.
7. If either steps 3 to 5 or step 6 are performed the automaton reads the next input element.

□

Obviously, we have to claim that the state-transition systems must be deterministic: this fact is expressed in Formula (4). Moreover, only finite automata are useful as specifications which can be (directly) implemented.

$$\begin{aligned} \forall (s,e,c',a',s'),(s,e.c'',a'',s'') \in T, is\_occurrence(\omega), first(\omega) = e. \\ c'(sec(\omega)) = \text{TRUE} \rightarrow c''(sec(\omega)) = \text{FALSE} \end{aligned} \tag{4}$$

The following example illustrates how a state-transition system may be used to specify that every time the sequence *abc* (of the three events a, b, c) is detected a new event x should be produced. Every time the a-event occurs the action a_a should be executed (analogous on the occurrence of b the action a_b, on c action a_c respectively). The state-transition system x is defined by the tuple

$\langle \{s_0, s_1, s_2, s_3\}, \{(s_0, a, \text{true}, a_a, s_1), (s_1, b, \text{true}, a_b, s_2), (s_2, c, \text{true}, a_c, s_3),$
$(s_3, a, \text{true}, a_a, s_1)\}, s_0, \{s_3\}, \{\text{true}\}, \{a_a, a_b, a_c\}, \{a, b, c\}\rangle$

(where 'true' is a constant function which always returns the value TRUE). Figure 2 shows a graphical representation of this automaton (the conditional function 'true' is omitted, the terminal state $s_3 \in S_{term}$ is emphasized by a bold-faced frame).

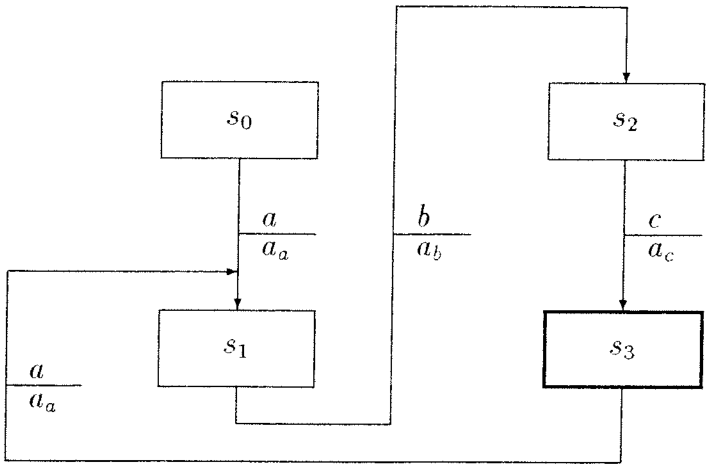


Figure 2: State-transition system accepting the event sequence abc

4.2 The Augmented State of State-Transistion Systems

Any state $s \in S$ of an automaton reflects the history of events already occurred, but says nothing about their attributes. In the example above, being in state s_2 means that the events a and b occurred and the state-transition system is waiting on a c-event now. There is no information about the attributes of the events occurred so far or, e.g., if the state-transition system is processing the first or the n-th sequence of the form *abc*.

For this reason we augment our system to refine the information on the current state of an automaton. Each state-transition system comprises an additional memory m where attributes, statistical data, or timing information can be stored. The actions ($a \in$ Act) executed on the occurrence of events are allowed to change the contents of this memory and the conditional functions ($c \in$ Cond) might test the memory as well as the attributes of the actual event.

Using the memory m to store relevant data at occurrence time enables information exchange without referring directly to the attributes of the events. Moreover, this method removes the problems induced by the repetition operator mentioned in Section 2.2. Every action $a \in$ Act may change the values stored in the memory and actions of conditional functions executed on subsequent occurrences (of the same or other events) can use this information.

We consider the memory m to be an array of variables, each of them capable of storing a value and each of them indicated by a unique identifier. We introduce the following notation to refer to the contents of a variable in the memory m: If id is an identifier than m/id refers to a specific variable in the memory. Using m/id in actions and conditional functions, values can be assigned to variables (using the assignment operator $:=$) or the memory contents may be tested (with the usual relations $=$, $\neq$, $\geq$, ...).

Moreover, we define actions as a sequence of operations on the memory m of an automaton. The fact that the execution of an action *act* changes the contents of the memory permanently, will be denoted by $[m'] = act\ (attrib\_list)[m]$ where $[m']$ is the memory after executing action *act* with *attrib_list* as parameter and the memory $[m]$.

Definition 3
In general, we will use the bracket-notation to denote that a function f changes a state-vector sv, which may consist of several parts.

$$f(param)[sv] = [sv']$$

means that applying f with the parameters *param* and with the actual values of $[sv]$ results in the changed state-vector $[sv']$. □

This notation simplifies the definition of recursive functions when some parameters are to be passed unchanged and others are determined by applying the function itself. To define that two functions f, g change the state-vector sv iteratively, we use the following notation:

Definition 4
We define now the sequencing operator $\star$ which enables us to apply functions f, g iteratively:

$$g(p_2) \star f(p_1)[sv] = g(p_2)[sv'] \qquad \text{with } [sv'] = f(p_1)[sv].$$

Note, that both functions must use the same type of state-vector. □

In the following example we will show the solutions to the problem explained in Section 2.2.

Example 2
Consider the events a, b, and c; to each of them an integer value attribute is attached. At first we like to find sequences of the form ab^+c with the condition that the attribute of the c-event should have the same value as the attribute of the *first* b-event. Figure 3 shows the graphical representation of the state-transition system defined by the tuple

$(\{s_0, s_1, s_2, s_3\}, \{(s_0, a, \text{true}, \text{null}, s_1), (s_1, b, \text{true}, a_b, s_2), (s_2, b, \text{true}, \text{null}, s_2), (s_2, c, c_c, \text{null}, s_3), (s_3, a, \text{true}, \text{null}, s_1)\}, s_0, \{s_3\}, \{\text{true}, c_c\}, \{\text{null}, a_b\}, \{a, b, c\})$.

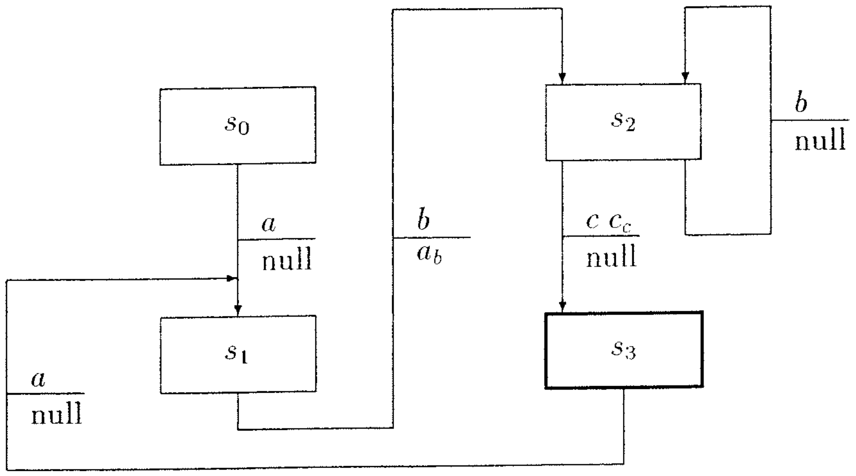


Figure 3: State-transition system accepting the event sequence ab^+c

The actions and conditional functions[3] are defined as follows (*null* is not really a function, but indicates that no action is executed on the transitions labeled with it):

$$a_b(b\_attrib\_list)[m] \overset{\text{def}}{=} m/b\_value := first(attrib\_list)$$

$$c_c(c\_attrib\_list)[m] = \begin{cases} \text{TRUE} & \text{if } m/b\_value = first(attrib\_list) \\ \text{FALSE} & \text{otherwise} \end{cases}$$

Simply replacing the *null*-function at the transition $(s_2, b, \text{true}, \text{null}, s_2)$ by the action a_b changes the effect of the automaton. With $(s_2, b, \text{true}, a_b, s_2)$ the first c-event following

[3]We use the symbol $\overset{\text{def}}{=}$ to denote that functions are procedures which alter the contents of the memory (denoted by the assignment operator $:=$) but do not return any value.

the sequence ab^+ must have the same attribute value as the *last* occurrence of the b-event. This fact is due to the (side-)effect of the action a_b which changes the memory every time it is executed. (Figure 4 shows the graphical notation of this state-transition system).

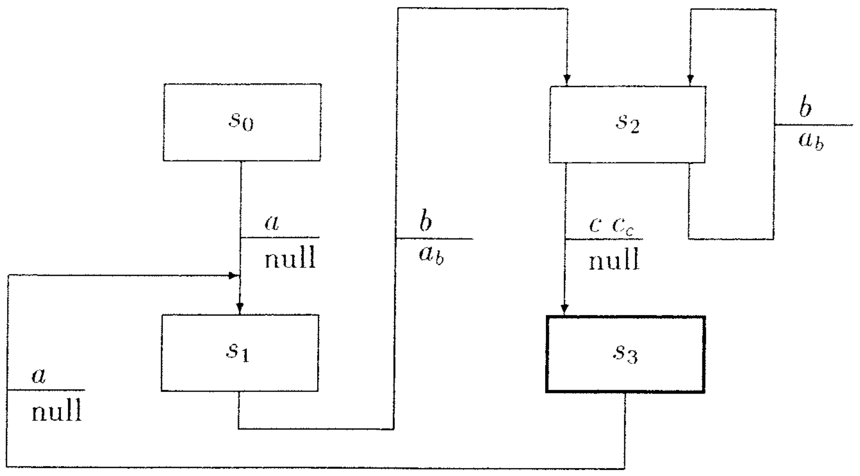


Figure 4: State-transition system accepting the event sequence ab^+c

□

To describe the semantics of an automaton $\alpha = \langle S_\alpha, T_\alpha, s_{0_\alpha}, S_{term_\alpha}, \mathrm{Cond}_\alpha, \mathrm{Act}_\alpha, E_{a_\alpha}\rangle$ (currently in the state $s \in S_\alpha$ and with the memory m) consuming one occurrence ω, we define a function $f_\alpha(\omega)[s,m,o]$ which may change the state, the memory, and the output o (which is a list of event occurrences produced by this state-transition system; we use the operator $\circ$ to insert new occurrences in this list)[4]:

$$f_\alpha(\omega)[s,m,o] = \begin{cases} [s', m', o], & \text{if } (s, \mathit{first}(\omega), \mathit{cond}, \mathit{act}, s') \in T_\alpha \\ & \wedge \mathit{cond}(\mathit{sec}(\omega))[m] = \text{TRUE} \\ & \wedge [m'] = \mathit{act}(\mathit{sec}(\omega))[m] \\ & \wedge s' \notin S_{term_\alpha} \\ [s', m', n \circ o], & \text{if } (s, \mathit{first}(\omega), \mathit{cond}, \mathit{act}, s') \in T_\alpha \\ & \wedge \mathit{cond}(\mathit{sec}(\omega))[m] = \text{TRUE} \\ & \wedge [m'] = \mathit{act}(\mathit{sec}(\omega))[m] \\ & \wedge s' \in S_{term_\alpha} \\ & \text{and } n \text{ is the newly produced occurrence.} \\ [s, m, o], & \text{otherwise.} \end{cases} \tag{5}$$

[4]if $\mathit{rest}(\mathit{list}) \neq \langle\rangle$ the function $\mathit{sec}(\mathit{list})$ is defined as $\mathit{first}(\mathit{rest}(\mathit{list}))$

With the definition of the $\star$–operator (see Definition 4) we can define the semantics of analyzing a time slot ts (a list of occurrences) with an automaton; we call this function $\mathcal{F}_\alpha$.

$$\mathcal{F}_\alpha(ts_i)[s,m,ts_o] = \begin{cases} \mathcal{F}_\alpha(rest(ts_i)) \star f_\alpha(first(ts_i))[s,m,ts_o], & \text{if } rest(ts_i) \neq () \\ f_\alpha(first(ts_i))[s,m,ts_o], \text{otherwise.} \end{cases} \quad (6)$$

Formula (6) shows that a state-transition system consumes a list of occurrences entry by entry. Each time an occurrence is processed, the automaton may change its state consisting of the automaton-state s, the memory m, and the output list o. The new state forms the basis for the next operation of the state-transition system, i.e. the next input element is analyzed.

4.3 Creating New Events

One of the questions left open by the preceding sections is, how identifier and attribute list of the newly generated event are determined. To provide a flexible but still simple way to generate different events with one automaton, we extend the notion of our state-transition systems with a labeling function $\mathcal{L}$: $S_{term} \mapsto E_c$ where E_c is the set of identifiers of the new events to be created.

Definition 5
Therefore, an automaton is defined by the tuple $\langle S, T, s_0, S_{term}, \text{Cond}, \text{Act}, E_a, E_c, \mathcal{L}\rangle$ where

- S, T, s_0, S_{term}, Cond, E_a, and Act are defined like above (see Section 4.1, Definition 1),
- E_c is the set of event identifiers which the automaton uses to produce new occurrences (with $E_a \cap E_c = \emptyset$),
- $\mathcal{L}$ is a labeling function ($\mathcal{L}$: $S_{term} \mapsto E_c$) which defines the identifier of the event to be created when the automaton reaches a state $s \in S_{term}$.

□

The semantics of the state-transition systems described in Section 4.1 has to be changed slightly, but before we do this we have to specify, how the attributes of the new occurrences are determined. As stated in [Stö93] we assume the number of attributes for every event to be fixed (and determined by the function $attrib\#()$). Hence, for each event

$e \in E_c$ an array of *attrib#*(e) variables can be reserved in the memory of the automaton. The actions can access the attributes with the identifiers *m/e.first*, *m/e.second*, ..., *m/e.n-th* (if $n =$ *attrib#*(e)). It is on behalf of the user to define actions assigning values to the attributes before a terminal state $s \in S_{term}$ is reached.

For convenience, we define that m/e refers to the whole attribute list in such a way that *is_attriblist*$(\langle m/e \rangle)$ holds. Step 5 of the semantics of state-transition system (see Section 4.1) must be redefined as follows:

Definition 6
Redefinition of step 5 of Definition 2.

5 If the system changes to a state $s' \in S_{term}$, then a new event is generated and inserted in the output list. The identifier of the new occurrence is determined by the labeling function $\mathcal{L}(s')$ and its attributes by accessing the corresponding value list in the memory $m/\mathcal{L}(s')$. Using the list notation (used in [Stö93]) the new occurrence is of the form $\langle \mathcal{L}(s') \, \langle m/\mathcal{L}(s') \rangle \rangle$.

□

In a more formal notation we can rewrite Formula (5) to define the semantics of the automaton $\alpha = \langle S_\alpha, T_\alpha, s_{0_\alpha}, S_{term_\alpha}, \mathrm{Cond}_\alpha, \mathrm{Act}_\alpha, E_{a_\alpha}, E_{c_\alpha}, \mathcal{L}_\alpha \rangle$:

$$f_\alpha(\omega)[s,m,o] = \begin{cases} [s',m',o], & \text{if } (s, \mathit{first}(\omega), \mathit{cond}, \mathit{act}, s') \in T_\alpha \\ & \wedge \mathit{cond}(\mathit{sec}(\omega))[m] = \text{TRUE} \\ & \wedge [m'] = \mathit{act}(\mathit{sec}(\omega))[m] \\ & \wedge s' \notin S_{term_\alpha} \\ [s',m',n \circ o], & \text{if } (s, \mathit{first}(\omega), \mathit{cond}, \mathit{act}, s') \in T_\alpha \\ & \wedge \mathit{cond}(\mathit{sec}(\omega))[m] = \text{TRUE} \\ & \wedge [m'] = \mathit{act}(\mathit{sec}(\omega))[m] \\ & \wedge s' \in S_{term_\alpha} \\ & \wedge n = \langle \mathcal{L}_\alpha(s') \, \langle m/\mathcal{L}_\alpha(s') \rangle \rangle \\ [s,m,o], & \text{otherwise.} \end{cases} \tag{7}$$

Note, that all actions $\in Act_\alpha$ and condition functions $\in Cond_\alpha$ use only a reduced state-vector: they are not able to access the state s or the output list o.

5 The Semantics of GOLD MINE

While the preceding section was concerned with the basic concepts of augmented state-transition systems which we use to analyze TATs, this section is dedicated to advanced topics like the notion of time, which is an integral element of our investigations. Furthermore, we will introduce the notion of *Compound Event Generators (CEGs)*, which stands for an object-oriented and very flexible method to define analyzing functions on TATs. In this section, the semantics of GOLD MINEwill be defined in a rigorous and formal way.

5.1 The Ordering of Events in a Time slot

Although, the ordering of event-occurrences in a time slot is not relevant when two time slots or two TATs are compared with each other, the ordering may be important when a TAT (a time slot, resp.) is analyzed. Consider, for example, a state-transition system which produces a new occurrence every time the sequence ab (with the events a, b) is detected. If this automaton is in the state of awaiting the next a-event and the time slot $\langle\langle \mathrm{b}\,\langle.\rangle\rangle\ \langle \mathrm{a}\,\langle.\rangle\rangle\rangle$ is investigated (the attributes are not relevant here), the ordering of the events affects the creation of a new event. With the given ordering and the assumed state of the automaton the b-event is ignored and the following a-event causes the state-transition system to change to the next state (and than wait for a b-event). But if the equivalent time slot $\langle\langle \mathrm{a}\,\langle.\rangle\rangle\ \langle \mathrm{b}\,\langle.\rangle\rangle\rangle$ is analyzed by the same automaton (in the same state, of course), the state-transition system will reach the terminal state indicating that a new event has to be inserted in the output list.

This simple example illustrates the necessity of a method which enables the user to specify how events in a time slot should be ordered, before this particular time slot is analyzed by a state-transition system. Moreover, it must be possible to define an ordering concerning multiple occurrences of the same event in one time slot (this may be the fact especially for compound events, but also for primitive ones; although, the latter case indicates that the granularity of the underlying digital clock may be to coarse).

For this purpose we introduce the *priority function*

$$pri\colon \mathcal{O} \mapsto I\!N_0 \times I\!N_0 \tag{8}$$

with $\mathcal{O}$ is the set of all occurrences

$$\mathcal{O} = \{ \langle id \langle attrib\_list \rangle \rangle \mid id \in \Sigma \;\wedge\; is\_attriblist(attrib\_list) \;\wedge\; length(attrib\_list) = attrib\#(id) \} \tag{9}$$

and N_0 is the set of all positive integers including 0. The function *pri* maps every occurrence to a pair of integers which serve as major and minor priority indicator. The major priority depends only on the event identifier and the minor priority is determined by the attribute list of the particular occurrence. Therefore, the function *pri* comprises two other functions, namely $major\_pri{:}\Sigma \mapsto N_0$ and $minor\_pri{:}R^n \mapsto N_0$ (with $n \in N_0$ is the number of attributes of the event considered and R is the set of all real numbers):

$$pri(\omega) = (major\_pri(first(\omega)), minor\_pri(rest(\omega))) \tag{10}$$

Definition 7
Using the priorities of occurrences the following relations can be defined (assuming that the occurrences $\omega_1, \omega_2 \in \mathcal{O}$ are assigned the priorities $pri(\omega_1) = (p_{1.maj}, p_{1.min})$ and $pri(\omega_2) = (p_{2.maj}, p_{2.min})$):

$$\begin{aligned} \omega_1 \prec \omega_2 \quad \leftrightarrow \quad & (p_{1.maj} < p_{2.maj}) \\ & \vee (p_{1.maj} = p_{2.maj} \;\wedge\; p_{1.min} < p_{2.min}) \end{aligned} \tag{11}$$

$$\begin{aligned} \omega_1 \preceq \omega_2 \quad \leftrightarrow \quad & (p_{1.maj} < p_{2.maj}) \\ & \vee (p_{1.maj} = p_{2.maj} \;\wedge\; p_{1.min} < p_{2.min}) \\ & \vee (p_{1.maj} = p_{2.maj} \;\wedge\; p_{1.min} = p_{2.min}) \end{aligned} \tag{12}$$

$$\omega_1 \simeq \omega_2 \quad \leftrightarrow \quad (p_{1.maj} = p_{2.maj}) \wedge (p_{1.min} = p_{2.min}) \tag{13}$$

□

We use the symbols $\prec$, $\preceq$, and $\simeq$ to distinguish relations defined by priorities from relations defined by the (in)equality of occurrences (e.g., occurrences $\omega_1 = \omega_2 \leftrightarrow \omega_1 = \langle id_1 \langle a\_list_1 \rangle \rangle \wedge \omega_2 = \langle id_2 \langle a\_list_2 \rangle \rangle \wedge id_1 = id_2 \wedge a\_list_1 = a\_list_2$).

To make priorities an efficient mechanism, we require for two occurrences $\omega_1, \omega_2 \in \mathcal{O}$ with $\omega_i = \langle id_i \langle . \rangle \rangle$ and $pri(\omega_i) = (p_{i.maj}, p_{i.min})$ $(i = 1, 2)$ that

$$\omega_1 \simeq \omega_2 \quad \leftrightarrow \quad (id_1 = id_2) \wedge (p_{1.min} = p_{2.min}) \tag{14}$$

and

$$id_1 \neq id_2 \quad \rightarrow \quad \omega_1 \not\simeq \omega_2. \tag{15}$$

Equations (14) and (15) say that two occurrences with the same priority must be occurrences of one specific event and that they cannot be distinguished by their attributes. Or in other words, if two occurrences differ in their identifiers, they must have different priorities.

Making use of this priority function we can define a *sort function* fulfilling the following requirements[5].

$$\begin{aligned} \forall is\_timeslot(ts).\ \ & is\_timeslot(sort(ts)) \\ & \wedge ts \equiv_{ts} sort(ts) \\ & \wedge is\_sorted(sort(ts)) \end{aligned} \tag{16}$$

$$\begin{aligned} \forall is\_timeslot(ts).\ \ & is\_sorted(ts) \leftrightarrow ((length(ts) < 2) \\ & \vee((length(ts) \geq 2) \rightarrow (\forall \omega \ \mathbf{in}\ rest(ts).\ \omega \preceq first(ts) \wedge is\_sorted(rest(ts))))) \end{aligned} \tag{17}$$

The sort function rearranges the occurrences in a time slot in such a way that the occurrences with higher priorities are located before those with lower priorities. This allows to order the occurrences in a time slot before it is analyzed by a state-transition system. With this method any ambiguity is eliminated and the designer of a state-transition system can rely on a canonical ordering of the occurrences in time slots.

5.2 Compound Event Generators

We will now introduce another level of abstraction which combines state-transition systems with the priority property described above. We call this system, which serves as shell for automata, a *Compound Event Generator (CEG)*. The basic idea of a CEG is to define a function which keeps track of the time slots in a TAT and uses a state-transition system to analyze the (sorted) time slots. In other words, a CEG is like a shell which houses an automaton.

Definition 8
A *Compound Event Generator* is defined by the tuple $\langle E_a, E_c, pri, \alpha\rangle$ where

- E_a is the set of accepted events,
- E_c is the set of events which are to be created,
- $pri{:}E_a \mapsto N_0 \times N_0$ is a priority function according to the definition in Section 5.1, and

[5]The predicates and relations used are defined in [Stö93]

- $\alpha = \langle S, T, s_0, S_{term}, C, A, E_a, E_c, \mathcal{L}\rangle$ is an automaton defining the functions $f_\alpha()$ and $\mathcal{F}_\alpha()$ according to Formulae (7) and (6).

$\square$

Each CEG $\gamma = \langle E_a, E_c, pri, \alpha\rangle$ defines the functions g_γ and $\mathcal{G}_\gamma{:}\mathcal{TAT} \mapsto \mathcal{TAT}$ in the following way (note, that g_γ uses another type of state-vectors than $f_\alpha()$ and $\mathcal{F}_\alpha()$:

$$\mathcal{G}_\gamma(\tau_i) = \tau_o, \quad \begin{array}{l} \text{where } \tau_o \text{ is part of the state-vector} \\ [s', m', \tau_o] = \mathrm{g}_\gamma(\tau_i)[s_0, m, \langle\rangle], \\ \text{with } s_0 \text{ is the start state of } \alpha \end{array} \tag{18}$$

$$\mathrm{g}_\gamma(\tau_i)[s, m, \tau_o] = \begin{cases} [s, m, \tau_o], & \text{if } \tau_i = \langle\rangle \\ \\ \mathrm{g}_\gamma(rest(\tau_i))[s', m', \tau_o {}^\frown \langle ts\rangle] & \text{otherwise} \\ \text{with } [s', m', ts] = \mathcal{F}_\alpha(sort(first(\tau_i)))[s, m, \langle\rangle] & \end{cases} \tag{19}$$

As can be seen from the definition in Formula (19), the function g_γ uses the function $\mathcal{F}_\alpha$ to analyze each (sorted) time slot of a TAT. Formula (18) defines, that the state-transistion system starts in its start-state and an empty TAT as output list; after applying g_γ recursively the newly created TAT is stored in this output list.

The function $\mathcal{F}_\alpha$ returns a list of newly created event occurrences if the state-transition system reaches terminal states while the input list of occurrences is scanned. This output list ts is inserted as time slot in the new TAT (denoted by $\tau_o {}^\frown \langle ts\rangle$ in the state-vector of the CEG). Therefore, the TAT to be analyzed and the newly created TAT are *synchronized* (see [Stö93]). As side-effect, the function $\mathcal{F}_\alpha$ changes the state of the automaton (to s' and m') which must be passed to the analysis of the next time slots $(\mathrm{g}_\gamma(rest(\tau))[s', m', \tau {}^\frown \langle ts\rangle])$.

Note, that a CEG allows the creation of new occurrences only at the recognition time of the last occurrence required to reach a terminal state $s \in S_{term}$ of the automaton. This is due to the fact that new occurrences are inserted into a time slot with the same index as the time slot currently under investigation. Thereby, ante- or post-dating is prohibited explicitly.

Example 3
Using the state-transition system shown in Figure 2 (page 16) and the definition of CEG-systems introduced above, it is not possible to create a new event (say x) after the recognition of a sequence abc but in the time slot corresponding to that of the occurrence of the a-event.

$$\begin{array}{ccccccc} \langle \ldots \langle\rangle & \langle\langle \mathrm{a}\,\langle.\rangle\rangle\rangle & \langle\rangle \ldots \langle\rangle & \langle\langle \mathrm{b}\,\langle.\rangle\rangle\rangle & \langle\rangle \ldots \langle\rangle & \langle\langle \mathrm{c}\,\langle.\rangle\rangle\rangle & \langle\rangle \ldots\rangle \\ & \not\downarrow & & & & & \\ \langle \ldots \langle\rangle & \langle\langle \mathrm{x}\,\langle.\rangle\rangle\rangle & \langle\rangle \ldots \langle\rangle & \langle\rangle & \langle\rangle \ldots \langle\rangle & \langle\rangle & \langle\rangle \ldots\rangle \end{array}$$

Only at the time of detection of the c-event the x-occurrence can be inserted into the output time slot (as shown in the following illustration).

$$\begin{array}{ccccccc}
\langle\ldots\langle\rangle & \langle\langle \mathrm{a}\,\langle.\rangle\rangle\rangle & \langle\rangle\ldots\langle\rangle & \langle\langle \mathrm{b}\,\langle.\rangle\rangle\rangle & \langle\rangle\ldots\langle\rangle & \langle\langle \mathrm{c}\,\langle.\rangle\rangle\rangle & \langle\rangle\ldots\rangle \\
 & \not\downarrow & & \not\downarrow & & \downarrow & \\
\langle\ldots\langle\rangle & \langle\rangle & \langle\rangle\ldots\langle\rangle & \langle\rangle & \langle\rangle\ldots\langle\rangle & \langle\langle \mathrm{x}\,\langle.\rangle\rangle\rangle & \langle\rangle\ldots\rangle
\end{array}$$

□

5.3 CEGs and Time

As mentioned in the previous section, the only time slot into which a new occurrence can be inserted is the one corresponding to the time slot containing the occurrence which forced the automaton to change to a terminal state. But, by now, CEG-systems are not sufficient enough to measure durations or simply to remember the time when the first occurrence of a certain sequence has been detected.

It is impossible to define actions which update a variable in the memory of a state-transition system to realize a logical clock because actions are only executed when the corresponding events occur and the transition-conditions are satisfied. Therefore, an action cannot determine how many time slots have been passed since the last update of the clock-variable. Because of this fact, we need to introduce another mechanism which keeps track of the actual time and makes it available to conditional functions and actions.

For this purpose, we supply some functions with a second parameter *cl* (*clock*), which represents the actual time and is passed to actions and conditions. The clock advances always after a time slot has been analyzed by the state-transition system. Again, the CEG serves as shell for the automaton because the CEG system is in charge of advancing the clock. This leads to a redefinition of the Formulae (18) and (19); a CEG $\gamma = \langle E_a, E_c, pri, \alpha\rangle$ defines the following functions g_γ and $\mathcal{G}_{CEG}$:

$$\mathcal{G}_\gamma(\tau_i) = \tau_o, \quad \begin{array}{l}\text{where } \tau_o \text{ is part of the state-vector} \\ [s', m', \tau_o] = \mathrm{g}_\gamma(\tau_i, 0)[s_0, m, \langle\rangle], \\ \text{with } s_0 \text{ is the start state of } \alpha\end{array} \tag{20}$$

$$\mathrm{g}_\gamma(\tau_i, cl)[s, m, \tau_o] = \begin{cases} [s, m, \tau_o] & \text{if } \tau_i = \langle\rangle \\ \mathrm{g}_\gamma(rest(\tau_i), cl + \Delta t)[s', m', \tau_o{}^\frown\langle ts\rangle] & \text{otherwise} \\ \text{with } [s', m', ts] = \mathcal{F}_\alpha(sort(first(\tau_i)), cl)[s, m, \langle\rangle] & \end{cases} \tag{21}$$

In Formula (20) the initial value of the clock is set to zero. Formula (21) shows that for each iterations the clock's value is incremented by Δt which should be equal to the granularity of the underlying digital clock (if $\Delta t = 1$ then the clock just reflects the index of the actual time slot—starting with 0).

Obviously, we have to redefine the behavior of state-transition systems to introduce the clock value *cl* (Formulae (6) and (7) are to be replaced by (22) and (23)); each automaton $\alpha = \langle S_\alpha, T_\alpha, s_0, S_{term_\alpha}, C_\alpha, A_\alpha, E_{a_\alpha}, E_{c_\alpha}, \mathcal{L}_\alpha \rangle$ defines:

$$\mathcal{F}_\alpha(ts, cl)[s,m,o] = \begin{cases} \mathcal{F}_\alpha(rest(ts), cl) \star f_\alpha(first(ts), cl)[s,m,o] & \text{if } rest(ts) \neq \langle\rangle \\ f_\alpha(first(ts), cl)[s,m,o], & \text{otherwise} \end{cases} \tag{22}$$

$$f_\alpha(\omega, cl)[s,m,o] = \begin{cases} [s', m', o], & \text{if } (s, first(\omega), cond, act, s') \in T_\alpha \\ & \wedge cond(sec(\omega), cl)[m] = \text{TRUE} \\ & \wedge [m'] = act(sec(\omega), cl)[m] \\ & \wedge s' \notin S_{term_\alpha} \\ [s', m', n \circ o], & \text{if } (s, first(\omega), cond, act, s') \in T_\alpha \\ & \wedge cond(sec(\omega), cl)[m] = \text{TRUE} \\ & \wedge [m'] = act(sec(\omega), cl)[m] \\ & \wedge s' \in S_{term_\alpha} \\ & \wedge n = \langle \mathcal{L}_\alpha(s') \langle m/\mathcal{L}_\alpha(s') \rangle \rangle \\ [s, m, o], & \text{otherwise.} \end{cases} \tag{23}$$

Maintaining a clock and making its value available to the actions introduces the possibility of saving the occurrence time of certain events in the memory m. This time-value can be retrieved by other actions (at a later time) to compute, e.g., the duration between two occurrences. Having the chance to compare the actual time (the value of *cl*) with time values stored in the memory opens up a new dimension for enabling or disabling state transitions.

The following example illustrates how the clock *cl* can be used in actions.

Example 4
Measuring the length of intervals is one of the most important application when analyzing real-time systems. We will now specify a CEG γ_1 which inserts an occurrence into its output TAT every time an interval starting with an a-event and ending with a z-event is detected. The new event (x) has the duration of the interval as attribute. The CEG is defined by $\gamma_1 = \langle \{a, z\}, \{x\}, pri_{\gamma_1}, \alpha_{\gamma_1} \rangle$ where

$$pri_{\gamma_1}(\omega) = \begin{cases} (1,0), & \text{if } first(\omega) = a \\ (0,0), & \text{otherwise} \end{cases}$$

and the automaton $\alpha_{\gamma_1} = \langle\{s_0, s_1, s_2\}, T, s_0, \{s_2\}, \{\mathit{true}\}, \{\mathit{start}, \mathit{term}\}, \{a, z\}, \{x\}, \mathcal{L}\rangle$ with the transitions T defined in Figure 5, and the actions *start* and *term*:

$$\mathit{start}(\mathit{attrib\_list})[m, cl] \stackrel{\text{def}}{=} m/st := cl$$

$$\mathit{term}(\mathit{attrib\_list})[m, cl] \stackrel{\text{def}}{=} m/x.\mathit{first} := cl - m/st$$

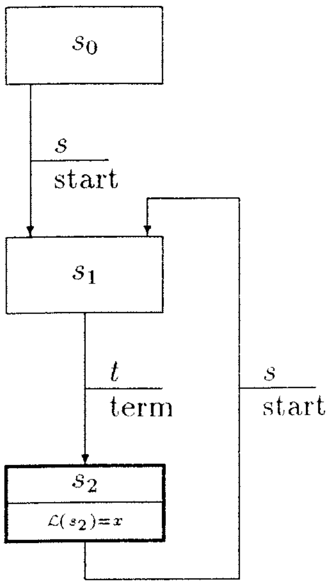


Figure 5: State-transition system recognizing the interval $s - t$

As can be seen from the definition above, the action *start* saves the actual clock value, which is the occurrence time of start event a, in the memory location m/st. The action *term* computes the duration of the interval using the occurrence time of the termination event z (clock value cl) and the start time saved in the memory (m/st). The duration is saved in the memory cell $m/x.\mathit{first}$ which is by definition the attribute for occurrences of the event x. □

Analogous, the clock cl can be used in conditional functions to express time depending conditions. Consider, for example, a conditional function which returns only TRUE when the corresponding event has been detected within (or following) a certain interval after the preceding occurrence.

The CEG system introduced so far is a very flexible mechanism to analyze TATs and we think that the basic structure, namely the state-transition system, is well-known and easy to understand. But with the definition of CEGs given so far, it is not possible to solve all analysis problems. For example, to recognize *coending intervals* (determined by two particular events) requires multiple occurrences of the event to be created (see

Figure 6), but (by definition) reaching an terminal state is associated with inserting just one new occurrence into the output list. Therefore, the notion of a CEG will be extended in the next sections.

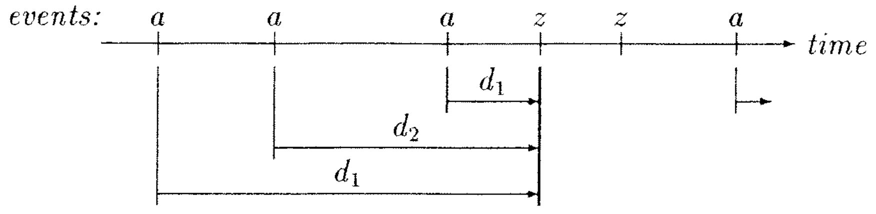


Figure 6: Example of coending intervals *determined by the events s – t*

5.4 Templates and Instances of Automata

Facing the analysis problems explained above, we decided to follow the object-oriented paradigm with automata.

Definition 9
A CEG will no longer be the shell for a single automaton but of a variable set of state-transition systems. A CEG is specified by a set of *templates of automata*

$A_T = \{\alpha \mid \alpha = \langle S_\alpha, T_\alpha, s_0, S_{term_\alpha}, C_\alpha, A_\alpha, E_{a_\alpha}, E_{c_\alpha}, \mathcal{L}_\alpha\rangle$ defines a state-transition system according to Definition 5 (see page 20$\}$).

A template of an automaton is just the *specification of the behavior* of a state-transition system. In addition to this, there is another list to be included in the definition of a CEG, namely the list of instances A_{I_0}, which determines which templates initially contribute to the behavior of the CEG. Therefore, a CEG is defined by a tuple $\langle E_a, E_c, pri, A_T, A_{I_0}\rangle$. Obviously, the following statements hold

$$E_a = \bigcup_{\alpha\in A_T} E_{a_\alpha} \tag{24}$$

$$E_c = \bigcup_{\alpha\in A_T} E_{c_\alpha} \tag{25}$$

□

Whereas an template is just the formal specification of a state-transition system, an instance of an automaton is a concrete 'incarnation' of such a specification which is in a certain state ($[s, m]$). In the following forumlae we define the predicates *is_state*, *is_instance*, and *is_instlist* which test wether a expression is a valid state of an automaton, an instance, or a list of instances. Therefore, the predicate $\mathit{is\_instlist}(A_{I_0})$ evaluates to TRUE.

$$\forall \alpha \in A_T.\ \mathit{is\_state}(x, \alpha) \leftrightarrow (x = [s, m] \wedge s \in S_\alpha) \tag{26}$$

$$\forall \mathit{is\_list}(x).\ \mathit{is\_instance}(x) \leftrightarrow (\mathit{first}(x) \in A_T \wedge \mathit{is\_state}(\mathit{sec}(x), \mathit{first}(x))) \tag{27}$$

$$\begin{aligned} &\forall \mathit{is\_list}(x).\quad x = \langle\rangle \rightarrow \mathit{is\_instlist}(x) \\ &\forall \mathit{is\_list}(x).\quad \mathit{is\_instlist}(x) \leftrightarrow (\mathit{is\_instance}(\mathit{first}(x)) \wedge \mathit{is\_instlist}(\mathit{rest}(x))) \end{aligned} \tag{28}$$

Now, the function of a CEG is no longer determined by only one automaton, but by all state-transition systems in the list A_I ($\mathit{is\_instlist}(A_I)$), which is initially equivalent to the list A_{I_0}. During operation a CEG may change the list of instances A_I by adding new instances. Every newly created instance of a state-transition system is in its initial state and has an empty memory. Therefore, the list A_I is extended in the following way:

$$A'_I = A_I \frown \langle\langle \alpha, [s_0, m]\rangle\rangle \qquad \text{with} \quad \begin{aligned} &\alpha \in A_T \\ &\wedge\ s_0 \text{ is the start state of } \alpha \\ &\wedge\ m \text{ is an unused portion of memory} \end{aligned}$$

The dynamic creation of new state-transition systems and the distinction between templates and instances affects the definition of the functions f_α, $\mathcal{F}_\alpha$, g_{CEG}, and $\mathcal{G}_{CEG}$ (Formulae (20), (21), (22), and (23)). But we will postpone the redfinition of thos formulae until we have introduced the mechanism of parameter passing in the next section.

5.5 Passing Parameters to Instances

To enhance the benefits of the object-oriented approach presented in the previous section, we introduce a mechanism to pass parameters to each instance at creation time. Without this means, the profits of dynamically creating automata would be rather small. Consider, for example, the problem illustrated in Figure 7. It is conceivable that one is interested in the duration of intervals which are determined by a starting event a and a terminating event z and the additional requirement that the values of certain attributes must match.

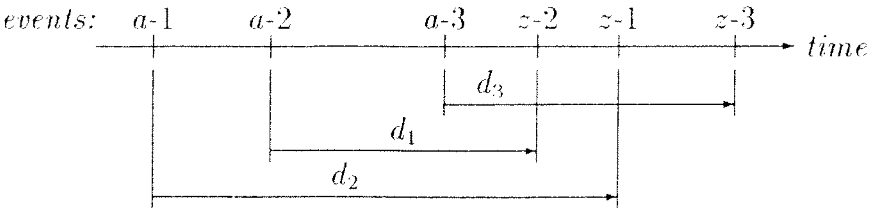


Figure 7: Measuring intervals determined by events and attributes

To solve this problem, it is not sufficient to instantiate a particular state-transition system every time an a-event occurs. Each new automaton waiting for a z-event rather must know which value the attribute of the occurrence should match. Hence, we have to introduce a possibility to pass parameters to each instance.

For this reason we augment the state of a state-transition system ($[s, m]$) to $[s, m, p]$, where p is an array of parameters, which are supplied at creation time of each automaton. Similar to reading variables in the memory m, each parameter can be read or written by specifying p/id (with id is an identifier).

In order to determine the parameters of an instance which is to be created dynamically, we have to change the labelling function $\mathcal{L}$ ($\mathbb{P}$ is the set of all possible parameter types; obviously, this set depends on the implementation).

$$\mathcal{L} : S_{term} \mapsto \{\textbf{create}\} \times E_c \cup \{\textbf{inst}\} \times A_T \times \mathbb{P}^n.$$

The first kind of labels defines that the particular event $e \in E_c$ occurs when this state is reached. Labels with the keyword **inst** indicate that an instance of the named automaton has to be created and the parameters given have to be passed. If the instance which will be created accepts no parameters, the number n is equal to 0 which indicates that the label is of the form (**inst**, (α)). Parameters in the label can be supplied as constants or as identifiers referring to memory locations of the actual instance. The latter allows to determine the parameters dynamically during the analysis procedure.

The functions ($\mathcal{L}_t$:$S_{term} \mapsto \{\textbf{create, inst}\}$ and $\mathcal{L}_c$:$S_{term} \mapsto E_c \cup A_T \times \mathbb{P}^n$) introduced in Definition 10 enable a selective access to the type t and the contents c of a label.

Definition 10
For every state $s \in S_{term}$ with the labelling (t, c), where $t \in \{\textbf{create, inst}\}$ and $c \in E_c \cup A_T \times \mathbb{P}^n$ (with $n \geq 0$) we define

$$\mathcal{L}_t(s) = t, \tag{29}$$

$$\mathcal{L}_c(s) = c. \tag{30}$$

□

In analogy to the parameter passing mechanism for automata, we allow CEGs to be parameterized, also. These parameters can be passed to the instances in the list A_{I_0}.

We are now ready to redefine the functions $\mathcal{G}_\gamma$, g_γ, $\mathcal{F}_\alpha$, and f_α, thereby replacing Formulae (20), (21), (22), and (23). We start with semantics of a CEG $\gamma = \langle E_a, E_c, pri, A_T, A_{I_0}\rangle$: the effect of applying the function $\mathcal{G}_{CEG}: \mathcal{TAT} \mapsto \mathcal{TAT}$ defined by the tuple given above can be specified using the function g_γ:

$$\mathcal{G}_\gamma(\tau_i, p_\gamma) = \tau_o, \text{ where } \tau_o \text{ is part of the state-vector } [A_I, \tau_o] = \mathrm{g}_\gamma(\tau_i, 0)[A_{I_0}, \langle\rangle]. \tag{31}$$

Each element in the list A_{I_0} has the form $\langle \alpha, [s, m, p_\alpha]\rangle$, where α is an automaton template and $[s, m, p_\alpha]$ is the initial state of this automaton instance. The values of parameters *palpha* may be specified by using constants or by referring to parameters p_γ of the CEG (using identifiers).

Formula (32) shows the definition of the function g_γ, which recurs into the TAT τ_i and collects the new time slots in τ_o.

$$\mathrm{g}_\gamma(\tau_i, cl)[A_I, \tau_o] = \begin{cases} [A_I, \tau_o] & \text{if } \tau_i = \langle\rangle \\ \mathrm{g}_\gamma(\mathit{rest}(\tau_i), cl + \Delta t)[A'_I, \tau_o {}^\frown \langle ts'\rangle], & \text{otherwise} \end{cases} \tag{32}$$

The (not necessarily) changed list A'_I and the time slot ts' are determined by the function $\mathcal{H}_I$ (which will be defined subsequently) in the following way:

$$[A'_I, ts'] = \mathcal{H}_I(\mathit{sort}(\mathit{first}(\tau_i)), cl)[A_I, \langle\rangle].$$

The function $\mathcal{H}_I$ is just used to split each (sorted) time slot into its constituent occurrences: Formula (33) shows its definition.

$$\mathcal{H}_I(ts_i, cl)[A_I, ts_o] = \begin{cases} [A_I, ts_o]. & \text{if } ts_i = \langle\rangle \\ \mathcal{H}_I(\mathit{rest}(ts_i), cl)[A''_I, ts'' \circ ts_o], & \text{otherwise.} \end{cases} \tag{33}$$

Again, the list A''_I and the time slot ts'' are determined by applying another (lower level) function, namely h_I, which has the purpose to compute the contribution of every instance I which is an element of the list A_I:

$$[A''_I, ts''] = h_I(\mathit{first}(ts_i), A_I, cl)[\langle\rangle, \langle\rangle].$$

The function h_I accepts a single occurrence, a list of automaton instances, and the clock-value as parameters and uses a state-vector with another list of instances which has the purpose to save the changes (of the state-vector) of each instance and to accumulate newly generated instances. While the function h_I iterates through the elements of the list of automaton instances given as parameter, the newly created occurrences are collected in the second element of the state-vector. This leads to the following definition of the function h_I:

$$h_I(\omega, A_I, cl)[A_{I_\sigma}, ts] = \begin{cases} [A_{I_\sigma}, ts], & \text{if } A_I = \langle\rangle \\ \\ h_I(\omega, rest(A_I), cl)[A'''_{I_\sigma}, ts'''], & \text{otherwise} \end{cases} \tag{34}$$

where $A'''_{I_\Sigma} = rest(A_{I_\Sigma})^\frown\langle\langle\alpha, [s', m', p']\rangle\rangle^\frown i$ if $first(A_I) = \langle\alpha, [s, m, p]\rangle$ and $ts''' = ts^\frown o$ if $[s', m', p', o, i] istheresultofcomputing f_\alpha(\omega, cl)[s, m, p, \langle\rangle, \langle\rangle]$.

In the definition given above the function f_α which is the function defined by the automaton α is used to determine the effect of the occurrence ω. Applying this function can result in the creation of a new occurrence o or a new automata instance i or just in a simple change of state $[s', m', p']$ of the current instance ($first(A_I)$). We have to redefine Formula (23) f_α to satisfy the new needs caused by introducing multiple instances and parameter passing:

$$f_\alpha(\omega, cl)[s, m, p, o, i] = \begin{cases} [s', m', p', o, i], & \text{if } (s, first(\omega), cond, act, s') \in T_\alpha \\ & \wedge cond(sec(\omega), cl)[m, p] = \text{TRUE} \\ & \wedge s' \notin S_{term_\alpha} \\ & \wedge [m', p'] = act(sec(\omega), cl)[m, p] \\ \\ [s', m', p', n \circ o, i], & \text{if } (s, first(\omega), cond, act, s') \in T_\alpha \\ & \wedge cond(sec(\omega), cl)[m, p] = \text{TRUE} \\ & \wedge s' \in S_{term_\alpha} \\ & \wedge \mathcal{L}_t(s') = \textbf{create} \\ & \wedge n = \langle \mathcal{L}_c(s') \langle m / \mathcal{L}_c(s') \rangle\rangle \\ & \wedge [m', p'] = act(sec(\omega), cl)[m, p] \\ \\ [s', m', p', o, i^\frown I], & \text{if } (s, first(\omega), cond, act, s') \in T_\alpha \\ & \wedge cond(sec(\omega), cl)[m, p] = \text{TRUE} \\ & \wedge s' \in S_{term_\alpha} \\ & \wedge \mathcal{L}_t(s') = \textbf{inst} \\ & \wedge \mathcal{L}_c(s') = (\alpha_{new}, p_{new}) \\ & \wedge I = \langle \alpha_{new}, [s_{0_{new}}, m_{new}, p_{new}] \\ & \wedge [m', p'] = act(sec(\omega), cl)[m, p] \\ \\ [s, m, p, o, i], & \text{otherwise.} \end{cases} \tag{35}$$

Formula (35) concludes the definition of the semantics of a CEG. It shows that every single occurrence affects every instance to either generate a new instance or to create a new occurrence or just to change the state or to do nothing. The function $\mathcal{F}_\alpha$ is no longer needed because it was replaced by the functions $\mathcal{H}_I$ and h_I.

Having defined the semantics of state-transition systems and Compound Event Generators (CEGs) in a rigorous and formal way, all the problems mentioned in Section 2 (e.g. "When does an event defined by $a'b^*$ occur?") could be eliminated. Section 7 will present the syntax of GOLD MINE which is a specification language exactly implementing this semantics. In the following section we will discuss some characteristics of the approach presented above in general and especially in connection with its practical use.

6 Characteristics of GOLD MINE

This section provides a discussion of the characteristics of Compound Event Generators and a few examples to illustrate the capability of this approach.

In the previous section we showed that the combination of the event-based behavioral abstraction approach and the event-action paradigm forms a powerful basis for analyzing distributed real-time systems. Describing event sequences in terms of deterministic finite state-transition sytems and adding the opportunity to instantiate such automata dynamically limits the number of necessary basic elements in the approach with no restriction on the capabilities.

6.1 State-Transtion Sytems — The Basic Elements

In Section 2 we discussed some event definition languages (like EDL by Bates and Wileden) and their advantages and disadvantages. The reasons for taking state-transition systems as basic notion to describe event sequences are:

- they are a well known and powerful means to describe sequences;
- it is easy to augment the notion of a traditional automaton with the necessary elements like additional conditions, actions, or labeling functions;
- they are simple to understand and, therefore, easy to use.

Another—very important—advantage of our augmented state-transition system is the ability to remember facts of the past by using the state-vector, which consists of a state every automaton is in, parameters, and a memory which serves as data store for dynamically acquired information. Taking a closer look at the semantics of a state-tranistion system defined in Formula (35), it becomes obvious that the memory is necessary to remember characteristics of already analyzed occurrences and time slots for the use in subsequent actions and/or conditional functions.

Each state-transition system is just a partial system which makes no sense without the notion of the surrounding CEG. But on the other hand, each CEG contains one or more automaton instances and the behavior of a specific CEG is defined by the sum of these objects. Each instantiation alters the behaviour of the CEG because a special aspect is observed by the new automaton. At every point in time, the list of instances of a CEG represents the history of the TAT analyzed so far. Both the number of automaton instances and their individual states reflect the current state of analysis.

Because of the fact that state-transition systems do not maintain their own clocks, it is possible to instantiate automata at any random time during the analysis phase. As can be seen from Formula (35), which defines the semantics of an state-transition system α, each automaton instance analyzes simple occurrences and relies on the clock value passed as parameter. Each state-transition system is in charge of monitoring particular characteristics of the "rest of the TAT" and to create new occurrences and/or instances if distinguished properties are recognized. It is legal to instanitiate automata at random points in time and to monitor just a "rest of the TAT" because the properties to be monitored are only of interest after the occurrence of some other characteristics (remember that $\forall is\_tat(\tau).\tau \neq \langle\rangle \rightarrow is\_tat(rest(\tau))$; moreover, $\forall is\_timeslot(ts), is\_tat(\tau).ts \neq \langle\rangle \rightarrow is\_tat(rest(ts) \circ \tau)$).

It is important to note that new instances are to be created whenever different characteristics have to be observed concurrently. For example, for some kinds of intervals one (or more) automaton instance(s) observe(s) the occurrences of *opening events* and other state-transition systems wait for the corresponding *closing events*. The possibility to dynamically instantiate new automata simplyfies the realisation of a wide range of measurements, respectively, actually enables to formulate them using augmented state-transition systems as basic elements.

6.2 CEGs in Practical Use

As shown in the previous section, Compound Event Generators are an appropriate means to transform and, thereby, analyze Timed Attributed Event Traces. A CEG is a system consisting of

- a set of automaton templates,
- a list of instances of automata,
- two distinct sets of events (accepted and generated),
- and a priority function ordering the set of accepted events.

Which automata are to be applied on the input TAT is determined by the set of instances. Initially, at least one instance consisting of the automaton template and its initial state-vector (including the parameters) has to be in this set. And, as can be seen from Formulae (33) and (34), each occurrence of every time slot of the input TAT is analyzed by every instance. This analysis may result in the generation of new occurrences or the instantiation of new state-transition systems, which will start to contribute to the overall effect of the CEG when the next occurrence is investigated.

Note, that this is a very important point. It is necessary that newly generated instances can take effect in that time slot which the occurrence causing the instantiation is in. But on the other hand, it would not be correct to analyze this particular occurrence with the new instances which are direct products of it.

Because all occurrences of a time slot are sorted according to the priority function of the CEG, appending newly created instances to the list of automaton instances so that they will take effect with the next occurrence in the current time slot (if there is one left) is a deterministic and correct solution. We will show this mechanism with the next example.

Example 5

Consider the following TAT (we introduce indices for time slots to make it easier to refer to a particular one):

$$\langle\langle\langle \mathrm{a}\,\langle\cdot\rangle\rangle\rangle_0\,\langle\ \rangle_1\,\langle\ \rangle_2\,\langle\langle \mathrm{a}\,\langle\cdot\rangle\rangle\rangle_3\,\langle\ \rangle_4\,\langle\langle \mathrm{z}\,\langle\cdot\rangle\rangle\rangle_5\,\langle\ \rangle_6\,\langle\langle \mathrm{a}\,\langle\cdot\rangle\rangle\ \langle \mathrm{z}\,\langle\cdot\rangle\rangle\rangle_7\,\langle\langle \mathrm{z}\,\langle\cdot\rangle\rangle\rangle_8 \ldots\rangle$$

If the event a marks the entry into a certain (recursive) procedure and the event z the corresponding return to the calling function, we could be interested in the duration of each procedure call. But, as can be seen from Figure 8, we assume that the granularity of the underlying clock is to coarse to obtain correct values.

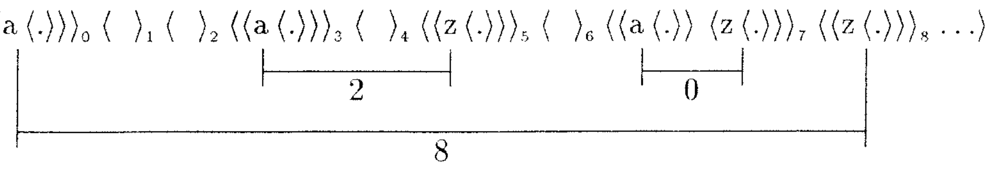

⟨⟨⟨a ⟨·⟩⟩⟩₀ ⟨ ⟩₁ ⟨ ⟩₂ ⟨⟨a ⟨·⟩⟩⟩₃ ⟨ ⟩₄ ⟨⟨z ⟨·⟩⟩⟩₅ ⟨ ⟩₆ ⟨⟨a ⟨·⟩⟩ ⟨z ⟨·⟩⟩⟩₇ ⟨⟨z ⟨·⟩⟩⟩₈ ...⟩
2
0
8

Figure 8: Measuring the duration of stacked intervals

It is very important that a new event with the duration 0 as parameter is created while analyzing time slot number 7. The two reasons are:

1. Missing one of the constituing events (a, z) will cause erroneous results for the rest of the measurement.
2. The user must be informed about the unsatisfactory granularity of the clock.

This problem can be solved by using the following CEG γ_x and by applying the semantics defined in the previous section.

$$\gamma_x = \langle \{a, z\}, \{x\}, pri_x, \{\alpha_1, \alpha_2\}, \langle\langle \alpha_1, [s_{1_0}, m_1, p_1]\rangle\rangle\ \rangle$$

We will show the (more illustrative) graphical notations of the two state-transition systems α_1, α_2 in Figure 9 rather than define them formally:

The states s_{1_0} and s_{2_0} are the start states of the corresponding automata. The labels of the two terminal states s_{1_1} and s_{2_1} are:

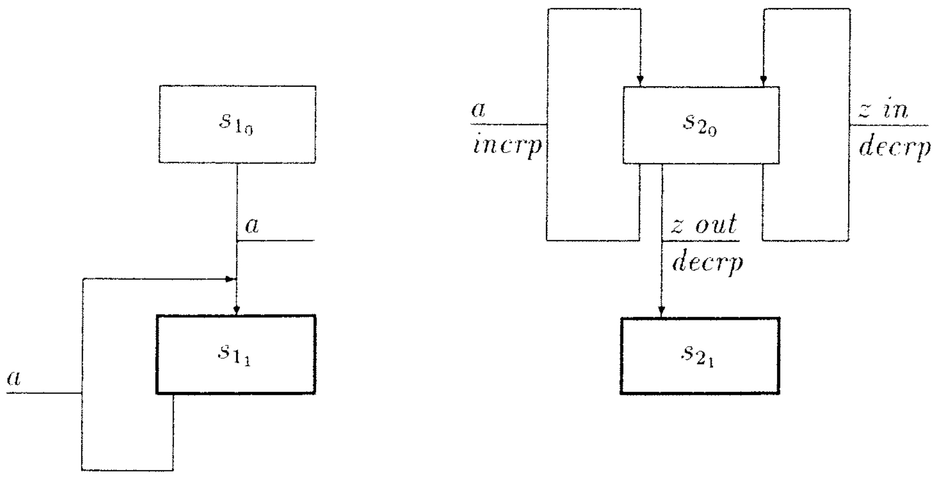


Figure 9: Automaton templates recognizing stacked intervals

$$\mathcal{L}(s_{1_1}) = \textbf{inst}\ (\alpha_2, (1, cl)) \qquad \text{and} \qquad \mathcal{L}(s_{2_1}) = \textbf{create}\ (x)$$

Note, that we use the notation $m/x.1$ and $p/1$ to refer to the memory location number 1 respectively the parameter number 1. This notation seems to be convinient because we do not have to deal with names for memory locations or parameters when defining the functions.

The condition functions *in* and *out* used in the graphical specification of the automaton templates are defined as follows (note that this functions test parameter $p/1$ which is set by the actions *incrp* and *decrp* which are also defined below):

$$in = \begin{cases} \text{TRUE} & \text{if } p/1 > 1 \\ \text{FALSE} & \text{otherwise} \end{cases} \qquad out = \begin{cases} \text{TRUE} & \text{if } p/1 \leq 1 \\ \text{FALSE} & \text{otherwise} \end{cases}$$

$$incrp \overset{\text{def}}{=} \quad p/1 := p/1 + 1 \qquad decrp \overset{\text{def}}{=} \begin{bmatrix} p/1 & := & p/1 - 1 \\ m/x.1 & := & cl - p/2 \end{bmatrix}$$

The priority function pri_x is defined as follows:

$$pri_x(occ) = \begin{cases} (1,0), & \text{if } \mathit{first}(occ) = a \\ (0,0), & \text{otherwise} \end{cases}$$

This CEG comprises two state-transition systems which realize the required behavior. Automaton α_1, which is instantiated once as initial state-transition system, creates on every occurrence of event a an instance of system α_2.

All instances of automaton α_2 use two parameters supplied by the creating system α_1:

p/1 reflects the level of occurrence (1 stands for the innermost level).

p/2 is the start time of the procedure; this value is necessary to compute the duration of each procedure call (interval).

All instances of the state-transition system α_2 assign the computed duration time to the memory location $m/x.1$, which is the only attribute of the occurrences of the event x.

Let us now take a closer look at the activities of this CEG when it analyzes a given TAT τ_{in}. The following table shows the input TAT τ_{in}, the state of the CEG (columns 'cl'and 'A_I (list of instances)', and the output TAT τ_{out} as they change (step by step).

In the list A_I we will show memory locations and parameters as list surrounded by paranthesis. For this reason, the parameters of instances of automaton α_2 are displayed in the form $(p/1, p/2)$. In the memory of α_2-instances only the location $(m/x.1)$ is used. If the contents of $m/x.1$ is not determined (because it was not set by an action), the memory will be displayed as (.).

τ_{in}	cl	A_I (list of instances)	τ_{out}
$\langle$		$\langle\langle\alpha_1,[s_{1_0},(),()]\rangle\rangle$	$\langle$
$\langle\langle a\langle.\rangle\rangle\rangle$	0	$\langle\langle\alpha_1,[s_{1_1},(),()]\rangle$ $\langle\alpha_2,[s_{2_0},(.),(1,0)]\rangle\rangle$	$\langle\,\rangle$
$\langle\,\rangle$	1	$\langle\langle\alpha_1,[s_{1_1},(),()]\rangle$ $\langle\alpha_2,[s_{2_0},(.),(1,0)]\rangle\rangle$	$\langle\,\rangle$
$\langle\,\rangle$	2	$\langle\langle\alpha_1,[s_{1_1},(),()]\rangle$ $\langle\alpha_2,[s_{2_0},(.),(1,0)]\rangle\rangle$	$\langle\,\rangle$
$\langle\langle a\langle.\rangle\rangle\rangle$	3	$\langle\langle\alpha_1,[s_{1_1},(),()]\rangle$ $\langle\alpha_2,[s_{2_0},(.),(2,0)]\rangle$ $\langle\alpha_2,[s_{2_0},(.),(1,3)]\rangle\rangle$	$\langle\,\rangle$
$\langle\,\rangle$	4	$\langle\langle\alpha_1,[s_{1_1},(),()]\rangle$ $\langle\alpha_2,[s_{2_0},(.),(2,0)]\rangle$ $\langle\alpha_2,[s_{2_0},(.),(1,3)]\rangle\rangle$	$\langle\,\rangle$
$\langle\langle z\langle.\rangle\rangle\rangle$	5	$\langle\langle\alpha_1,[s_{1_1},(),()]\rangle$ $\langle\alpha_2,[s_{2_0},(5),(1,0)]\rangle$ $\langle\alpha_2,[s_{2_1},(2),(0,3)]\rangle\rangle$	$\langle\langle x\langle 2\rangle\rangle\rangle$
$\langle\,\rangle$	6	$\langle\langle\alpha_1,[s_{1_1},(),()]\rangle$ $\langle\alpha_2,[s_{2_0},(5),(1,0)]\rangle$ $\langle\alpha_2,[s_{2_1},(2),(0,3)]\rangle\rangle$	$\langle\,\rangle$
$\langle\langle a\langle.\rangle\rangle$	7	$\langle\langle\alpha_1,[s_{1_1},(),()]\rangle$ $\langle\alpha_2,[s_{2_0},(5),(2,0)]\rangle$ $\langle\alpha_2,[s_{2_1},(2),(0,3)]\rangle$ $\langle\alpha_2,[s_{2_0},(.),(1,7)]\rangle\rangle$	$\langle$
$\langle z\langle.\rangle\rangle\rangle$	7	$\langle\langle\alpha_1,[s_{1_1},(),()]\rangle$ $\langle\alpha_2,[s_{2_0},(7),(1,0)]\rangle$ $\langle\alpha_2,[s_{2_1},(2),(0,3)]\rangle$ $\langle\alpha_2,[s_{2_1},(0),(0,7)]\rangle\rangle$	$\langle x\langle 0\rangle\rangle\rangle$
$\langle\langle z\langle.\rangle\rangle\rangle$	8	$\langle\langle\alpha_1,[s_{1_1},(),()]\rangle$ $\langle\alpha_2,[s_{2_1},(8),(0,0)]\rangle$ $\langle\alpha_2,[s_{2_1},(2),(0,3)]\rangle$ $\langle\alpha_2,[s_{2_1},(0),(0,7)]\rangle\rangle$	$\langle x\langle 8\rangle\rangle\rangle$
$\cdots\rangle$	$\cdots$		$\cdots\rangle$

This example shows that *stacked intervals* can be found easily with a CEG consisting of two simple state-transition systems. The first one (α_1) generates an instance of the second kind (α_2) every time a start event occurs. Each instance of the second kind keeps track of the level of recursion and creats one occurrence of the compound event when the end of the corresponding interval is detected. □

As can be seen from Example 5, the notion of a CEG is based on the idea of independend instances of automata. These instances might work in parallel because there is no way for one automaton to influence the course of events or even to change the memory in another instance. Each automaton instance in confronted with the sequence of occurrences and may create a new instance or a new occurrence in the output TAT of its CEG, but there is no communication between state-transition systems.

6.3 Towards an Event Recognizer

With the notion of a CEG we are able to create TATs on the base of an input TAT. This new TATs are synchronized with the original one and. therefore, can be merged (using the operator +, see [Stö93]) and different CEGs which use compound events as well as primitive events as input elements may analyze these TATs again. Exploiting this property, we can construct an *Event Recognizer* which has the capability to create TATs providing events at different levels of abstractions. Such an event recognizer implements the event-based behavioral abstraction approach mentioned in Section 2.

If CEGs are iteratively applied in the way bescribed above, one must pay attention to the precedence relation imposed by the events which are consumed and produced by the different CEGs. Consider, for example, a TAT τ containing occurrences of the events a, b, and c and the two CEGs γ_1 and γ_2 where

- CEG γ_1 consumes the events a and b and produces an occurence of the event x always when a b-event is preceded by one or more a-events.
- CEG γ_2 consumes the events c and x and produces an occurrence of the evnt y on every occurrence of either c or x.

Obviously, the CEG γ_1 has to be applied first to the original TAT τ:

$$\tau_{new} = g_{\gamma_2}(g_{\gamma_1}(\tau) + \tau) + \tau$$

The new TAT τ_{new} contains occurrences of all events a, b, c, x, and y.

The fact that characteristics of the already analyzed occurrences is represented by the state of the automata in a CEG and the restriction that newly generated occurrences can be inserted in actual time slots only (post- or ante-dating is prohibited) are essential for the implementation of an *on-line event recognizer*. These two properties imply that every time slot which has been analyzed by all automaton instances of all CEGs can be discarded. The information of already analyzed time slots is no longer of interest because there is no way of referring to occurrences and their attributes in these time slots.

This leads to a simple and effective implementation of the event recognizer. Every time slot is investigated by all CEGs which keep track of the history by changing the state-vectors of their automaton instances and possibly the number of automaton instances itself.

7 The syntax of GOLD MINE

In this chapter we will focus on the object-oriented event specification language GOLD MINE, which was designed to implement the semantics introduced in the previous sections. GOLD MINE is intended to be

- independent of the **VTA**-system, easy to be ported to other hard- or software platforms,
- easy to learn and simple to apply, and
- a problem-oriented, i.e. not a general purpose programming language.

For this reasons GOLD MINE realizes only those parts of the problem domain which deal with the definition of CEGs or state-transition systems. The actions and conditional functions are to be expressed in a general purpose programming language, such as C, C++, or Ada.

The following sections introduce the object-oriented concepts and the syntaxt of GOLD MINE. To discribe the semantics of the various syntactic constructs, we will refer to formulae of previous chapters. The syntax will be introduced in an extended Bakkus normal form using the following meta-symbols and font faces:

standard	Non-terminal symbols.
bold face	Terminal symbols.
'{' and '{'	The terminal symbols { and }
[*brackets*]	Options surrounded by brackets may be omitted or choosen once.
{ *braces* }	Options surrounded by braces may be omitted or choosen a random number of times.
::=	Substitution.
|	Seperates options, one of them has to be choosen
•	Marks the End of a substitution.

The non-terminal symbol *Identifier* is not specified because of its intuitive usage in GOLD MINE (it is a sequence of characters and some other symbols like in any general purpose programming language). A summary of reserved words (the terminal symbols), which must not be used as identifiers of CEGs, automata, variables. or types can be found in Appendix B. Appendix A presents a survey of all productions introduced in this chapter.

A correct GOLD MINE definition has the form:

gold_mine	::=	{type_section ceg_definition} •

We will take a closer look at the *type_section* in the following section, whereas the *ceg_definition* will be presented in detail in Section 7.2.

7.1 The Host-Language Concept

As mentioned above, GOLD MINE was designed to be an event specification language realizing the semantics introduced in Section 5. Therefore, our main stress was to keep the definition of CEGs and state-transition systems as simple as possible. But on the other hand, there is a need for traditional programming language constructs like *if-then-eles* or *loop* statements to implement the actions and conditional functions. As a result of these considerations we decided to combine elements of our specification language with the merits of a traditional programming language which we call the *host-language* and which can be choosen freely (as already mentioned). Hence, a complete CEG specification comprises

- a GOLD MINE module,
- a host-language module, and
- a type-definition module.

The type-definition module serves as interface between the two other parts of the specification:

In practice, the integration of the host-language module and the GOLD MINE module is realized by simply calling the specified functions from within the GOLD MINE part in a host language independent way, using parameters and return values to exchange values. Since GOLD MINE is strongly typed, it is necessary to have a common type system on both sides of the interface to guarantee consistent use of values. Note that simply using the typing concepts of a particular host-language in the GOLD MINE part would have been contradictory to the goal of independency of a specific programming language.

Our solution to this problem is the use of GOLD MINE specific data types which are mapped to the corresponding host-language types using a translation table which is contained in the type-definition module. Therefore, changing the host-language does not

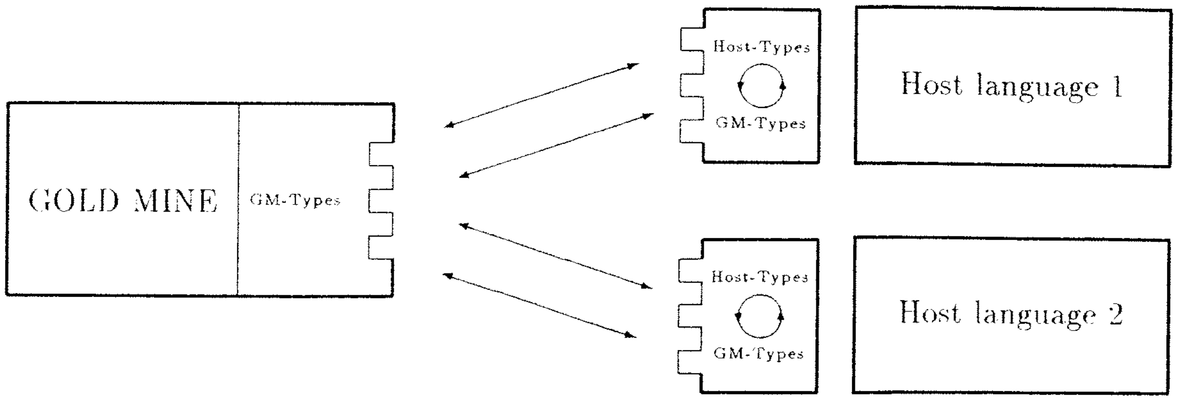


Figure 10: GOLD MINE *typing concept*

affect the GOLD MINE specific part of a CEG definition as long as the type-definition module is adapted to the new situation (see Figure 10).

A correct *type_section* has the form:

type_section	::=	{type_statement} •
type_statement	::=	**type** gm_type1 [**derived from** gm_type2]= type_def_body; •
type_def_body	::=	simple_type | event_type •
simple_type	::=	(host_language_type) •
event_type	::=	'{'host_language_variable_definition'}' •
gm_type1	::=	Identifier •
gm_type2	::=	Identifier •

As can be seen from the definitions above, the type-definition module just maps GOLD MINE-types to types of the actual host-language (host-types). Note that GOLD MINE distinguishes between simple types (host_language_types) and so-called event types (host_language_variable_definition). This is due to the fact that an event can have more than one attribute. In this case the correct host-language construct is a record type which comprises all attributes[6]. These record types (grouped by braces {}) are associated with a GOLD MINE-event type.

The inheritance mechanism (introduced by the keywords **derived from**) is discussed in Section 7.4.

To reduce the efforts to be undertaken in implementing the GOLD MINE-Compiler, we decided to realize it as precompiler with the host-language as target-language. Because of this fact, compiling a GOLD MINE source is a two step process:

1. Precompile GOLD MINE code to host-loanguage statements.
2. Compile these parts together with the actions and conditional functions (already defined in the host-language) obtaining executable object code.

At the moment, a GOLD MINE (pre)compiler for the host-language C++ is implemented as part of the VTA-project (see [Zei93]).

The advantage of this approch is twofold:

- The user of GOLD MINE needs not learn a new language to define procedures and functions.
- Using a widely distributed programming language as host-language makes it easy to port GOLD MINE to other (hardware) platforms and, as additional effect, the optimization features of the host-language compiler can be directly exploited.

A disadvantage of this approach may be the management of the various parts of a CEG. Therefore, a graphical user interface was developed and implemented using the operating system UNIX and X-Windows (see [Hol94]). The purpose of this user interface is to simplify the specification of state-transition systems and to support the user in the management of the different files (which may be reused, also).

7.2 Specification of a CEG

As defined in Section 5, CEGs are elements specifying how event traces are to be investigated. According to this definiton a CEG is a tuple with the elements $\langle E_a, E_c, pri, A_T, A_{I_0}\rangle$. Every element in this tuple has its counterpart in the GOLD MINE-language, but the syntax of GOLD MINE distinguishes between the

interface elements:

- the set of acceped events E_a,

[6]Actually, for internal reasons the GOLD MINE-compiler adds some other components to all event types but not to simple types.

- the set of events to be created E_c, and the

implementation elements:

- the priority function *pri*,
- the set of automata templates A_T, and
- the list of initial automata instances A_{I_0}.

For this reason the specification is split up into the two parts *interface* and *implementation*.

ceg_definition	::=	**ceg** ceg_name1 [(generic_params)] [**derived from** ceg_name2] [interface] [implementation] **end ceg** . ●
ceg_name1	::=	Identifier ●
ceg_name2	::=	Identifier ●
generic_params	::=	variable_list {; variable_list} ●
variable_list	::=	gm_type variable_name {, variable_name} ●

In the sense of the object-oriented language GOLD MINE a CEG is a class (the meaning of the keyword **derived from** will be discussed in Section 7.4) which may be instantiated one or many times. Therefore, the definition of the interface and the CEG and its automata bases on formal events which are mapped to real world events in the trace when a new object is created (this process is called *binding*). Using the *generic parameters* (generic_params) every instance of a CEG class can be initialized in a particular way.

The definition above shows that only GOLD MINE-types (which are mapped to host-language types in the type-definition module) are used to define the types of the generic parameters. If the host language is changed, only the mapping file has to be adapted but the CEG specification is still valid.

INTERFACE SECTION: The interface section defines the type of events a particular CEG will accept and create. From this point of view, a CEG can be seen as black box with well defined interface and function (Formula (31)) which is defined by the implementation part. The following defintions show the syntax of the interface part of a CEG:

interface	::=	[**accepts** (accepted_events);] [**generates** (generated_events);] ●

```
accepted_events     ::=  variable_list {; variable_list}  •
```

```
generated_events    ::=  variable_list {; variable_list}  •
```

Although the two sets (E_a and E_c) are not really optional. the former definition is correct because a CEG class may be derived from another one without changing these sets.

Despite the fact that the defintions above request only general varaiable_lists. the GOLD MINE-types used have to be event types, of course.

IMPLEMENTATION: The implementation of a CEG comprises the priority function. the definition of automaton templates, the initialition of the automaton instance list, and a section describing the functions (defined in the host-language) imported into the CEG.

```
implementation      ::=  [prototype_section]
                         [priority_section]
                         {automaton}
                         [ceg_initialize]  •
```

The prototype section contains a list of declarations of host-language functions imported into the CEG. These functions may be used as actions, conditional functions, or priority functions (as we will see later) or in the constructor of the CEG object.

To keep GOLD MINE simple and independent of any host-language, function calls are defined by the keyword **call** followed by the name of the function. For this reason any function is allowed to accept no or just one parameter (which may be a record type in the host-language, of course).

```
prototype_section   ::=  prototype
                             {prototype_list};
                             {prototype_list};  •
```

```
prototype_list      ::=  gm_type methode_name (gm_type)  •
```

```
method_name         ::=  Identifier  •
```

In the priority section the priorities of the events which are accepted by the CEG are defined. Formula (10) shows that the priority function is split into two parts which define a major and a minor priority. In GOLD MINE the major priority is simply expressed by listing all events defined in the *accepts-statement* seperated by > signs. This list assigns the highest priority to the left most event and the lowest priority to the right most event.

priority_section	::=	**priority** event_name [:priority_method] {> event_name[:priority_method]} •
priority_method	::=	Identifier •

As mentioned in Section 5, the minor priority function defines the ordering of occurrences of a certain event based on the attributes of this event. The priority methods, which are optional, map the values of attributes of occurrences to a (floating point) number which is interpreted as priority by GOLD MINE (higher return values are assumed to reflect higher priority). A priority method is an identifier of a function declared in the prototype section. Any priority method must accept an event type and must return a floating point value (predefined GOLD MINE-type PRIORITY).

Because the *automaton section* will be discussed in the following Section 7.3, the next defintion shows the initialization part of a CEG specification. The GOLD MINE-compiler as it is implemented now does not accept a list of of initial automaton instances but only a single instance. This restriction will be eliminated in one of the next releases of the GOLD MINE-compiler.

ceg_initialize	::=	**initialize instantiate** auto_name[**(params)**]; •
auto_name	::=	Identifier •

7.3 Automata

The fact that state-transition systems are defined as templates, which may be instantiated many times but always are contained in a surrounding CEG, suggests that automata are realized as nested classes.

Nested classes are defined in the context of another class. Moreover, they are hidden from the outside like other implemenation details. As a consequence of this, objects of nested classes can only be instantiated and acessed in the context of the surrounding class. Furthermore, deriving new classes from nested ones is only possible if the superclass is contained in the same surrounding class or in one of the superclasses of the surrounding class.

Because of the fact that state-transition systems accept and create events according to their definition, the specification of automata is split into the parts *interface* and *implementation* like a CEG.

automaton ::= **auto** auto_name1 [(params)] [**derived from** auto_name2]
    [interface]
    [auto_implement]
end auto •

Whereas the interface section defines the sets E_{a_α} and E_{c_α} (see Section 7.2), in the implementation section all other elements of the tuple $\alpha = \langle S_\alpha, T_\alpha, s_0, S_{term_\alpha}, C_\alpha, A_\alpha, E_{a_\alpha}, E_{c_\alpha}, \mathcal{L}_\alpha \rangle$ which defines an automaton α are specified.

---

auto_implement ::= [variable_section]
[prototype_section]
[state_section]
[directive_section]
[auto_initialize]
[cleanup_section] •

---

variable_section ::= **variable**
    variable_list;
    {variable_list;} •

Again, a prototype section must be used to import the host-language functions used as actions and conditions (sets A_α and C_α). Local variables defining the memory m of the state-vector may be declared in the variable section.

The sets S_α and S_{term_α} as well as the labeling function $\mathcal{L}_\alpha$ are defined together in the state section (see below).

---

state_section ::= **state**
    state_name [label_clause];
    {state_name [label_clause];} •

---

state_name ::= Identifier •

---

label_clause ::= **is start**
| **raise** event_name {, event_name}
| **instantiate** auto_name [(params)] •

The state section lists all states of the set S_α in random order seperated by semicolons. Additionally, all states of the set S_{term_α} and the start state are marked with one of the following keywords:

is start marks the start state s_{0_α}.

raise is the GOLD MINE synonym for the label **create**, which marks a state of S_{term_α}. If such a state is reached, the specified events are inserted to the output TAT.

instantiate is equivalent to the label **inst**. Every time such a state $\in S_{term_\alpha}$ is reached, the specified automaton is instantiated.

The transitions comprising a start state, an event, a target state, a condition, and an action are defined in a simple and intuitive way in the directive section of an automaton specification.

directive_section	::=	**directive** transition_clause; {transition_clause;} •
transition_clause	::=	state_name + event_name -> state_name [**cond** condition_method] [**call** transition_method]; •
condition_method	::=	Identifier •
transition_method	::=	Identifier •

All methods (functions) which are called in the directive section (i.e. all identifiers following the keywords **cond** or **call**) must be declared with the appropriate types (return types: GOLD MINE-types BOOL for conditional functions and VOID for actions) in the prototype section.

To initialize the local memory m at creation-time of every automaton instance, a particular method specified in the initialization section can be called. Moreover, to clean up dynamically created data structures another dedicated function may be called when the automaton instance is destroyed.

auto_initialize	::=	**initialize call** method_name; •
cleanup_section	::=	**cleanup call** method_name; •

7.4 Object Oriented Mechanisms

As already mentioned, CEGs are classes and, therefore, new CEGs may be derived from other ones. As usual in the object oriented paradigm, new classes inherit all aspects from their superclass (also called parent class), but it is possible to add new elements to the following parts of a CEG:

- the list of generic parameters,
- the list of accepted events,
- the list of created events,
- the priority rules, and
- the set of automaton templates.

Except for automaton templates (which are discussed in detail below) no redefinition of elements of the superclass is allowed. Therefore, all additional elements must have unique and new identifiers.

If the list of accepted events is extended and no new priority statement is specified, the GOLD MINE-compiler assumes the priorities of the new events (in the order of their appearence) to be lower as the priority of the events defined in the superclass. For more details see [Zei93].

Because of the fact that automaton templates are (nested) classes, they may be derived from superclasses, too. But with the current implementation of the GOLD MINE-compiler the following restrictions have to be observed:

- Automaton templates can only be derived inside a (derived) CEG. It is possible to derive an automaton template from
  - a superclass inside the same CEG or
  - a superclass inside a CEG which is a superclass of the CEG housing the considered template.
- As a direct consequence of this, it is not possible to inherit of an automaton template or to import one from a CEG which is not an ancestor of the actual CEG.

The following elements of a superclass may be extended in a derived class:

- generic parameters,
- accepted events,
- generated events,
- set of states, and
- set of transitions.

In addition to this extensions the following elements may be redefined also:

- the labeling function in the state section and

- the transitions in the directive section.

This section discussed only a part of the possibilities of the object oriented mechanisms of GOLD MINE. A complete introduction to GOLD MINE and its features may be found in [Zei93]. Moreover. [Zei93] also discusses some details of the implementation of the GOLD MINE-compiler and illustratres the power of GOLD MINEwith various examples.

A GOLD MINE Syntax – Summary

gold_mine	::=	{type_section ceg_definition} •
type_section	::=	{type_statement} •
type_statement	::=	**type** gm_type1 [**derived from** gm_type2] = type_def_body; •
type_def_body	::= |	simple_type event_type •
simple_type	::=	(host_language_type) •
event_type	::=	{host_language_variable_definition} •
gm_type1	::=	Identifier •
gm_type2	::=	Identifier •
ceg_definition	::=	**ceg** ceg_name1 [(generic_params)] [**derived from** ceg_name2] [interface] [implementation] **end ceg** . •
ceg_name1	::=	Identifier •
ceg_name2	::=	Identifier •
generic_params	::=	variable_list {; variable_list} •
variable_list	::=	gm_type variable_name {, variable_name} •
interface	::=	[**accepts** (accepted_events);] [**generates** (generated_events);] •

accepted_events	::=	variable_list {; variable_list} •
generated_events	::=	variable_list {; variable_list} •
implementation	::=	[prototype_section] [priority_section] {automaton} [ceg_initialize] •
prototype_section	::=	**prototype** prototype_list; {prototype_list;} •
prototype_list	::=	gm_type methode_name (gm_type) •
method_name	::=	Identifier •
priority_section	::=	**priority** event_name [:priority_method] {> event_name[:priority_method]} •
priority_method	::=	Identifier •
ceg_initialize	::=	**initialize instantiate** auto_name[**(params)**]; •
auto_name	::=	Identifier •
automaton	::=	**auto** auto_name1 [(params)] [**derived from** auto_name2] [interface] [auto_implement] **end auto** •
auto_implement	::=	[variable_section] [prototype_section] [state_section] [directive_section] [auto_initialize] [cleanup_section] •

variable_section ::= **variable**
    variable_list;
    {variable_list;} •

---

state_section ::= **state**
    state_name [label_clause];
    {state_name [label_clause];} •

---

state_name ::= Identifier •

---

label_clause ::= **is start**
    | **raise** event_name {, event_name}
    | **instantiate** auto_name [(params)] •

---

directive_section ::= **directive**
    transition_clause;
    {transition_clause;} •

---

transition_clause ::= state_name + event_name -> state_name
    [**cond** condition_method]
    [**call** transition_method]; •

---

condition_method ::= Identifier •

---

transition_method ::= Identifier •

---

auto_initialize ::= **initialize call** method_name; •

---

cleanup_section ::= **cleanup call** method_name; •

B Reserved Words

The following table contains a summary of all reserved words of the specification language GOLD MINE.

accepts	auto	call
ceg	cleanup	cond
derived	directive	end
from	generates	initialize
instantiate	is	priority
prototype	raise	start
state	type	variable

References

[AR88] I. Aalbersberg and G. Rozenberg. **Theory of traces**. Theoretical Computer Science, 60:1–82, 1988.

[Bat88] P. C. Bates. **Debugging heterogenous distributed systems using event-based models of behavior**. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 11–22, May 1988.

[BGR87] D. Bhatt, A. Ghonami, and R. Ramanujan. **An instrumented testbed for real-time distributed systems development**. In *Proceedings of the Real-Time Systems Symposium*, pp. 241–250. Computer Society Press, December 1987.

[BW83] P. C. Bates and J. C. Wielden. **High-level debugging of distributed systems: The behavioral abstraction approach**. The Journal of System and Software, 3:255–264, 1983.

[GYK90] G. S. Goldszmidt, S. Yemini, and S. Katz. **High-level language debugging for concurrent programs**. ACM Transactions on Computer Systems, 8(4):311–336, November 1990.

[HK90] W. Hseush and G. E. Kaiser. **Modeling concurrency in parallel debugging**. In *Proceedings of the 2^{nd} ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pp. 11–20. ACM Press, March 1990.

[HKLM87] R. Hofman, R. Klar, N. Luttenberger, and B. Mohr. **Zählmonitor 4: Ein Monitoringsystem für das Hardware- und Hybrid-Monitoring von Multiprozessor- und Multicomputer-Systemen**. In U. Herzog and M. Paterock, editors, *Proceedings der 4. GI/ITG-Fachtagung Messung, Modellierung und Bewertung von Rechnersystemen*, pp. 79–99. Springer Verlag, Berlin, September 1987.

[Hoa85] C. A. R. Hoare. **Communicating Sequential Processes**. Prentice Hall International, 1985.

[Hol94] J. Holzer. **Entwicklung einer graphischen Oberfläche zur Erstellung von Compound Event Generatoren**. Master's thesis, University of Technology, Vienna, Departement for Automation, February 1994. (Diplomarbeit, german).

[HW88] D. Haban and W. Weigel. **Global events and global breakpoints in distributed systems.** In B. D. Shriver, editor, *Proceedings of the 21st Annual Hawaii International Conference on System Science (Vol. 2)*, pp. 166–175, January 1988.

[HW90] D. Haban and D. Wybranietz. **A hybrid monitor for behavior and performance analysis of distributed systems.** IEEE Transactions on Software Engineering, 16(2):197–211, February 1990.

[JLSU87] J. Joyce, G. Lomow, K. Slind, and B. Unger. **Monitoring distributed systems.** ACM Transactions on Computer Systems, 5(2):121–150, May 1987.

[Lam78] L. Lamport. **Time, clocks, and the ordering of events in a distributed system.** Communication of the ACM, 21(7):558–565, July 1978.

[LCSM90] J. E. J. Lumpp, T. L. Casavant, H. J. Siegel, and D. C. Marinescu. **Specification and identification of events for debugging and performance monitoring of distributed multiprocessor systems.** In *Proceeding of the 10th International Conference on Distributed Systems*, pp. 476–483, June 1990.

[LL89] C.-C. Lin and R. J. LeBlanc. **Event-based debugging of object/action programs.** In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Systems*, pp. 23–34, January 1989.

[Maz86] A. Mazurkiewicz. **Trace theory.** In *Petri Nets: Applications and Relationships to Other Models of Concurrency (LNCS 255)*, pp. 279–324. Springer-Verlag, Berlin, September 1986.

[MH89] C. E. McDowell and D. P. Helmbold. **Debugging concurrent systems.** ACM Computing Surveys, 21(4):593–622, December 1989.

[MLC90] D. C. Marinescu, J. E. J. Lumpp, and T. L. Casavant. **Models for monitoring and debugging tools for parallel and distributed software.** Journal of Parallel and Distributed Computing, 9:171–184, 1990.

[PHK91] M. K. Ponamgi, W. Hseush, and G. E. Kaiser. **Debugging multithreaded programs with mpd.** IEEE Software, pp. 37–43, May 1991.

[Ros91] D. S. Rosenblum. **Specifying concurrent systems with TSL.** IEEE Software, pp. 52–61, May 1991.

[RRZ88] R. V. Rubin, L. Rudolph, and D. Zernik. **Debugging parallel programs in parallel.** In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[Sch93] U. Schmid. **Monitoring distributed real-time systems.** accepted manuscipt, 1993.

[Sno88] R. T. Snodgrass. **A relational approach to monitoring complexe systems.** ACM Transactions on Computer Science, 6(2):157–196, May 1988.

[Stö93] S. Stöckler. **Timed attributed event traces.** Technical Report 183/1-35, Institut für Automation, April 1993.

[WH88] D. Wybranietz and D. Haban. **Monitoring and performance measuring distributed systems during operation.** In *Performance Evaluation Review, Proceedings of the 1988 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pp. 197–206. ACM Press, May 1988.

[Zei93] R. Zeitlberger. GOLD MINE **Implementierung einer objekorientierten Spezifikationssprache für Event-basiertes Monitoring.** Master's thesis, University of Technology, Vienna, Departement for Automation, December 1993. (Diplomarbeit, german).