Semantics of Reactive Components in Event-Driven Workflow Execution

Dimitrios Tombros Andreas Geppert Klaus R. Dittrich

Institut für Informatik, Universität Zürich
Winterthurerstr. 190, CH-8057 Zürich, Switzerland
{tombros,geppert,dittrich}@ifi.unizh.ch

Abstract: The exact semantics of workflows and involved processing entities is an open yet urgent problem. This paper considers the semantics and correctness of event-driven workflow execution. The basis for the formalization in our approach is provided by an event history which records all events that have occurred during the execution of workflows. Workflows are executed by reactive components which operate on top of that history. Based on the history it is possible to determine the semantics of these reactive components (and consequently, the semantics of workflows) as well as to check whether their observable behavior is correct.
Keywords: workflow management, ECA-rules, distributed systems

1 Introduction and Motivation

Workflow management systems (WfMS) [8] are cooperative environments in which multiple distributed processing entities cooperate in order to accomplish tasks (e.g., process an insurance claim). A *workflow specification* defines the required activities and their execution dependencies, data flows between the activities, and further information such as the assignment of resources to activities. WfMSs have to provide the functionality to define and execute workflows in a distributed heterogeneous environment. However, while many research prototypes and products have been developed only few provide a formal foundation for the specification and execution of distributed workflows.

Our approach to the specification and execution of workflows is multi-leveled [14]. We separate the high-level (graphical) specification of workflow types and abstract workflow execution characteristics from an intermediate-level executable representation described with the BROKER/SERVICE MODEL (B/SM). The resulting system is then directly implemented with the help of a distributed execution framework (event engine - *EvE*) providing facilities for process event management and history logging, communication of participating workflow enactors etc. [7]. The B/SM provides a formal framework in which precise semantics of the workflow enactors and workflow execution can be described. This presents a series of advantages:

- the modeler can understand what a workflow enactor (*broker)* exactly does,
- the set of workflow executions that correctly model the workflow specification can be precisely defined,
- the correctness of the WfMS-implementation (transformations) can be verified,
- post-mortem workflow analysis is possible (e.g., for bottleneck recognition), and
- the impact of modifications of specifications during workflow execution (workflow evolution) can be precisely described.

In Proc. 9th Int'l Conf. on Advanced Information Systems Engineering, Barcelona, Spain, June 1997.

This paper is structured as follows: the next section contains an overview of our approach to workflow management. In section 3 we formally describe the notion of *event history* on which correctness of distributed workflow execution is based. Based on these definitions we describe the semantics of the workflow enactors (*brokers*) and their functionality in section 4. We then define the semantics of workflow execution. In section 5 we consider related work in formal semantics for WfMSs. We conclude the paper in section 6 by summarizing the contributions of our approach and future work.

2 Overview

2.1 Workflows

A *workflow type* is the enactable specification of a (business or design) process. The workflow specification defines the process steps. Execution dependencies between the steps are also defined. There are ordering, temporal, and output-data dependencies. A step can either be an elementary activity or in turn a workflow. In the first case, the activity is performed by a person or a software system. Both are henceforth summarized under the term *processing entity* (PE). The workflow specification assigns responsible PEs to activities. In the second case, a step is a sub-workflow and thus has in turn a complex structure and defines steps, constraints, and responsible PEs. A *workflow instance* is a concrete workflow executed according to the definition of its type. A WfMS is a software system that supports workflow specification and enactment.

In Fig. 1 we present as an example the processing of a health insurance claim (HIC). The workflow is initiated once the HIC mail arrives at a local insurance agency.

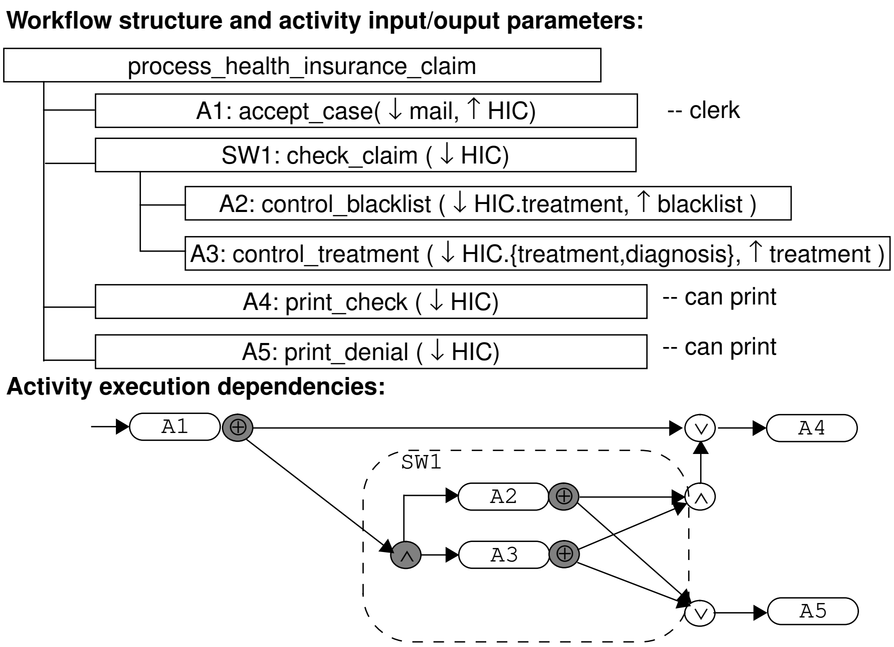

Fig. 1. The health insurance claim example workflow

A clerk working at the local agency creates a file containing the diagnosis, treatments, costs, and the insurance number (activity accept_case). If the claimed amount is below 300 Swiss Francs the HIC is directly accepted and paid (activity print_check). Otherwise, the HIC has to be processed at the central company clearing center. There a sub-workflow is started, which consists of activities that may be performed in parallel. One is to check whether some of the treatments are contained in a blacklist, in which case their coverage will be denied (activity control_blacklist). An activity of type control_treatment controls whether the treatment actually suits the diagnosis. If any of these controls fails, an entry is made in the customer record, a notification of the rejection is printed at the local agency (print_denial). Otherwise, a payment check can be printed at the local agency (print_check).

2.2 Brokers and Services

Besides workflow specification and enactment, further requirements such as the ability to integrate external systems and to reuse old artifacts (parts of workflow specifications, mediation software) are important for WfMSs. All these requirements can only be met if various aspects of a WfMS-architecture are exactly defined:

- *components* representing PEs that take part in one of the workflows — the structural and data aspect,
- *tasks* that can be performed by those components — the functional aspect, and
- *rules* determining how, when and under which constraints the components can/must perform the activities — the behavioral, temporal, and operational aspect.

We propose the BROKER/SERVICES MODEL (B/SM) [14] as a model for such architectures. Workflow specifications are mapped into B/SM and are automatically augmented with the necessary elements to attain executability. B/SM is described in detail in [14] so that this section gives only a short description for the sake of comprehension.

Brokers represent components of a WfMS, i.e., they model PEs involved in the execution of one or more workflows. They are *reactive* and offer *services* to other brokers (their clients). Brokers can access the functionality of their peers through service requests. The set of activities that can be executed within a WfMS (i.e., its functionality) is described by *services*. A service is specified by a *signature* consisting of the service name, a set of typed parameters bound at request time, and the possible replies and exceptions its request may cause. Broker behavior is defined in terms of event-condition-action (ECA) rules, which define the reaction to simple events such as requests or to complex events such as the transition of a workflow to a certain state. In our example workflow we use the following brokers:

- A person which provides the accept_case service represented by a broker (called `Meier`) assuming a clerk role. His behavior is expressed by the rule:
  ```
  ON request(accept_case, mail)
  IF(HIC = create_claim(mail)) AND HIC.amount > 300
  DO request(control_blacklist, HIC.treatment)
     request(control_treatment, HIC.treatment, HIC.diagnosis)
  ```
- Two PEs situated at the company clearing center: a database application (broker `RSA`) to check acceptance of the treatment (`control_blacklist` service) and an expert system (broker `RSEX`) which checks the compatibility between treatment

and diagnosis (`control_treatment` service). If both controls are successful, the check can be printed as expressed by the rule:

```
ON reply(control_treatment, treatment) AND
   reply(control_blacklist, blacklist)
IF treatment="OK" AND blacklist="NO"
DO request(print_check, HIC)
```

- A demon (broker `printer`) providing printing services (`print_check`, `print_denial`).

A specific service is provided by one or more brokers and can be initiated by client brokers through request events. Service execution is terminated when a reply or one of the possible exception events defined for the service is generated. The association between a service and its providers is established by means of *m:n* relationships called *responsibilities*. The way a service is executed within the context of a specific workflow depends on additional factors such as the fulfillment of defined preconditions for its provision, actual request parameters, etc. In typical cases, a workflow and its activities consume input data, produce output data, and read and manipulate data items stored in some sort of data store. Thus, *production data* must flow through a workflow according to the workflow's definition. Most of the production data will be managed by external systems, and it is not justified to assume that the WfMS and the production data store are tightly integrated. However, the B/SM supports the access of processing entities to these data through operations defined for their types. In B/SM, workflows are defined by a name, a set of initiating requests and a set of terminating replies. The workflow structure and related execution constraints are defined by the reaction of certain brokers to specific situations (request of services, etc.). Workflows are executed by the execution of the services provided. A workflow starts executing when its *initiation event* occurs which is a request of the first service to be provided.

2.3 EvE

Brokers and consequently, workflows are executed through the use of the distributed event engine *EvE* [7]. EvE's major purpose is to support brokers by providing event management, storage, and notification functionality. The principle by which brokers use EvE is the following: broker ECA-rules are translated to rules stored and executed in EvE. When a broker generates a primitive event, it notifies its local EvE-server about the occurrence. EvE then performs composite event detection and determines brokers that have registered a notification interest for such primitive and potentially detected composite event occurrences which may result from this event. These brokers are then appropriately informed and react as defined by their ECA-rules (whereby these reactions can in turn generate new events, and so forth). An important task of EvE upon event detection is the maintenance of an event history in which all primitive and composite event occurrences are represented.

In the present paper, we will not consider in detail the technical aspects of event generation, (composite) event detection and storage of events in the history in EvE. It is sufficient to note that these actions take place in a distributed environment, a fact that has to be considered for the correct definition of the semantics of events.

3 Event History

In this section, we define the semantics of primitive and composite event types and event occurrences. These semantics depend on the notion of time used in the distributed WFMSs. In an executing workflow event occurrences are collected in an event history (EH) which is also formally defined.

3.1 Event Types

We base our definitions on the auxiliary sorts $\mathcal{SN}$ which is a set of service names, $\mathcal{EN}$ which is a set of exception names, $\mathcal{S}$ which is a set of participating sites, and $\mathcal{C}$ which is a set of distributed reactive components (more precisely specified below).

Definition 1 (Primitive event types)

- broker interaction event types: let name $\in \mathcal{SN}$ be a service name, and plist a possibly empty list of typed parameters, then
    - REQ(name, plist) is a *service request* event type,
    - CFM(name) is a *request confirmation* event type,
    - RPL(name, plist) is a *reply* event type
    - EXC(name, ename) is an *exception* event type, where ename $\in \mathcal{EN}$
- TET is an explicit *time* event type.

Event parameter types are defined from a set of base types, $\mathcal{BT}$ = *{integer, float, boolean, string, reference}*. $\mathcal{REQ}$, $\mathcal{CFM}$, $\mathcal{RPL}$, and $\mathcal{EXC}$ are the sets of request, confirmation, reply and exception event types correspondingly defined in a given system and $\mathcal{PET} = \mathcal{REQ} \cup \mathcal{CFM} \cup \mathcal{RPL} \cup \mathcal{EXC}$. Note that primitive event types are local in the sense that they relate to one site.

Through the (recursive) use of unary and binary event operators composite event types can be constructed. In addition, a restriction sw on the component event types may be specified requiring that they occur within the same workflow.

Definition 2 (Composite event types)

Assume that ET1, ET2, and ET3 are event types. Then

- CON(ET1, ET2, sw) is a *conjunctive* event type,
- DEX (ET1, ET2) is an *exclusive-or disjunctive* event type,
- SEQ (ET1, ET2, sw) is a *sequential* event type,
- CCR (ET1, ET2, sw) is a *concurrent* event type,
- NEG (ET1, (ET2, ET3,sw), sw) is a *negative* event type, and
- REP (ET1, times, sw) is an *repetitive* event type.

We define an operation comptype on event types which returns the set of participating component event types (including the event type itself). $\mathcal{CET}$ is the set of all composite event types. $\mathcal{ET} = \mathcal{PET} \cup \mathcal{CET}$ is the set of all event types defined in the system. Note that composite event types whose component types relate to different sites are considered as *global*. The semantics of event composition are formally defined in section 3.3. At this point however, we describe them informally:

- CON is detected when both component events occur independent of their relative order.
- DEX is detected when either one of the two component events has occurred.
- SEQ is detected when the two component events occur in a certain time order.

- CCR is detected when the two component events occur at the same time.
- NEG is detected if ET1 does not occur in the interval defined by ET2 and ET3.
- REP is detected when ET1 occurs a predefined number of times.

3.2 Timestamps and Primitive Event Occurrences

In order to define the semantics of event occurrences we have to define our notion of time. A WfMS is composed of a set of distributed *sites* $\mathcal{S}$ each of which has a single local clock. An concept of time [10] with the following characteristics is supported:

- Time is linear allowing comparison of *timestamps*.
- Time has a *lower bound* coinciding with system initialization but no upper bound.
- A *global time* is approximated by adjusting the granularity of local clocks to the global reference clock granularity g_g.We assume that g_g (dependent on the synchronization precision among the local clocks) is small enough so that no two primitive events originating on the same site occur at the same global time. This assumption allows us to use a simplified semantic model for timestamps in distributed systems as suggested in [13] without affecting the power of our model. More specifically the assumption excludes workflow execution concurrency at any given site. In the typical application scenarios of the B/SM this restriction is easily met (or can be enforced without performance penalties).
- The three conceptual primitives of time *points* (e.g., 17:00 on 30.11.1996), time *intervals* with a lower and an upper bound (e.g., from this instant until 17:00 on 30.11.1996), and time *durations* (e.g., 24 hours) must be supported. These primitives are expressed in B/SM as part of event type definitions.

In a WfMS, a $2g_g$-restricted temporal ordering between primitive event occurrences based on their timestamps can be established. Thus in order to determine the temporal order of two primitive events occurring at different sites, their timestamp difference must be at least $2g_g$. Otherwise these events are perceived to occur concurrently. This means that a partial order structure of primitive event occurrences can be defined. Given $\mathcal{S}$ and a function gt$: \mathit{local} \rightarrow \mathit{global}$ calculating the global time gt_s of a local clock lt_s at a site s, a timestamp of an event occurrence is defined as follows:

Definition 3 (Timestamp)

A *timestamp* T(e) of an event occurrence is a partial function T(e): $\mathcal{S} \rightarrow \mathit{global}$ defining a global time $global_s = gt_s(lt_s)$ for each site s participating in the timestamp domain.

Based on the definition of event types and timestamps we define primitive event occurrences.

Definition 4 (Primitive event occurrence)

A *primitive event occurrence* is a 9-tuple pe = (type, site, eid, T, serv, wid, source, sid, plst), where

- type $\in \mathcal{PET}$ is the event type of pe
- site $\in \mathcal{S}$ is the site where peo occurred
- eid $\in \mathcal{N}$ is the unique identifier for site of the event occurrence
- serv $\in \mathcal{SN}$, is the name of the requested service if type = REQ, else serv = $\emptyset$
- T(pe) is the timestamp of pe defined as follows:
    - o T(pe)(site).global = $gt_s(lt_s)$

- o $T(pe)(s').global = \perp \;\; \forall s' \neq site$
- $wid \in \mathcal{N}$ is the identifier of the workflow instance in which peo occurred
- $source \in \mathcal{C}$ is the event originator (broker)
- $sid \begin{cases} \text{(eid,source) of the corresponding request event if type} \in \{CFM, RPL, EXC\} \\ \varnothing \text{ otherwise} \end{cases}$
- plist is a list of actual parameters

Note that wid, source, and plist are meaningless for time events, since these occur independently from workflows. In case of a request occurrence, its source is the client of the service while in case of confirmations and replies it is the service provider.

3.3 Event Composition and Event History

To define the semantics of event composition in a distributed system we use the notion of a component timestamp:

Definition 5 (Component timestamp)

A *component timestamp* t of a timestamp T(e) is a tuple (site, global), where $site \in domain(T(e))$ (see Definition 3) and $global = T(e)(site).global$. We can also write that $t \in T(e)$.

On the basis of the $2g_g$-precedence model [13] i.e., under the assumption that there are no two primitive events occurring at the same site at the same global time, the temporal relationships between two timestamps $T(e_1)$ and $T(e_2)$ are defined as follows:

Definition 6 (Concurrent timestamps)

The *concurrency relationship* between timestamps — written as $T(e_1) \sim T(e_2)$ — is said to hold iff $\forall t_1 \in T(e_1) \; \forall t_2 \in T(e_2)$: $(t_1.site = t_2.site \wedge t_1.global = t_2.global) \vee (t_1.site \neq t_2.site \wedge | t_1.global - t_2.global | < 2g_g)$

Two timestamps are thus concurrent if all component timestamps from the same site are equal and all component timestamps from different sites differ less than $2g_g$.

Definition 7 (Sequential timestamps)

The *sequential relationship* between timestamps — written as $T(e_1) < T(e_2)$ — is said to hold iff

$$\exists t_1 \in T(e_1) \; \exists t_2 \in T(e_2) : (t_1.site = t_2.site \wedge t_1.global < t_2.global) \vee (t_1.site \neq t_2.site \wedge t_1.global < t_2.global - g_g) \wedge$$
$$\forall t_1 \in T(e_1) \; \forall t_2 \in T(e_2) \;\; t_1.global \leq t_2.global$$

Two timestamps are sequential if for any of the component timestamps a precedence can be established[1] and for every preceding component timestamp the global time is not larger than that of the following component timestamp. Note that the sequential relationship is transitive. Two timestamps for which neither a concurrency nor a precedence relationship can be established are said to be unrelated when their base-values — at least one of which is non-atomic — are less than one clock tick apart.

---

1. This means that if component events originate at different sites their difference must be at least two global clock ticks, while if they originate at the same site they must be at least one global clock tick apart.

Definition 8 (Unrelated timestamps)

Two timestamps $T(e_1)$ and $T(e_2)$ are said to be *unrelated* — written as $T(e_1) \lozenge T(e_2)$ — iff $\neg\ ((T(e_1) \sim T(e_2)) \vee (T(e_1) < T(e_2)) \vee (T(e_2) < T(e_1)))$

The timestamps of composite events are determined by the latest timestamp of a component occurrence. In order to unambiguously determine the composite event timestamp if there are concurrent or unrelated timestamps, a *timestamp join* procedure is used [13]. Note that if no precedence relationship can be established between the participating timestamps $T(e_1)$ and $T(e_2)$ then at most two global ticks cover them (for a proof see [13]).

Definition 9 (Joined timestamps)

Given two timestamps $T(e_1)$ and $T(e_2)$ for which neither $T(e_1) < T(e_2)$ nor $T(e_2) < T(e_1)$ their *joined timestamp* is defined by the function $\mathcal{T}$: $T(e_1),T(e_2) \rightarrow T(e_1) \cup T(e_2)$

The join procedure is omitted here due to space considerations. Informally, the joined timestamp is the higher of the local clock values recorded for each participating event occurrence. Before we describe the semantics of event composition we provide the definition of a composite event occurrence:

Definition 10 (Composite event occurrence)

A composite event occurrence is a 6-tuple ce = (type, site, eid, T, wid, comp), where

- type $\in \mathcal{CET}$ is the event type of ce
- site $\in \mathcal{S}$ is the detection site
- eid $\in \mathcal{N}$ is the unique occurrence identifier for that detection site
- T(ce) is the timestamp of ce
- wid $\in \mathcal{N}$ is the identifier of the workflow instance in which ce occurred
- comp is a list of *component event occurrences* with elements of the form (site,eid)

In order to define the subset of $\mathcal{CET}$ of actually occurring composite events we use the concept of a (global) event occurrence sequence:

Definition 11 (Event occurrence sequence)

An *event occurrence sequence* eos, is a finite sequence of n primitive and/or composite event occurrences. eos is written as $<e_1, ..., e_n>$, where each e_i, $1 \leq i \leq n$ is an occurrence of some event type at some point in global time.

It is also valid that $\forall$ i,k, $1 \leq i,k \leq n$: $(T(e_i) < T(e_{i+k})) \vee (T(e_i) \sim T(e_{i+k})) \vee (T(e_i) \lozenge T(e_{i+k}))$, i.e. eos depicts a partial order among the participating event occurrences.

As already mentioned, event operators are used to construct composite events. Informally the timestamps of the occurrences are defined as follows:

- A CON is assigned the timestamp of the later of the component occurrences if their temporal order can be decided. Otherwise the two timestamps have to be joined (see Definition 9).
- A DEX is assigned the timestamp of the component event that actually occurred.
- A SEQ is assigned the timestamp of the later event occurrence.
- A CCR is assigned the joined timestamp of the component occurrences. Note that the component occurrences of a concurrent event can only occur at different sites.

- A NEG is a special case of a SEQT. Its timestamp is the one of the component occurrence defining the end of the interval.
- A REP is a special case of a SEQT where the precedence relationship is valid among each pair of consecutive component occurrences. The timestamp of the composite occurrence is the one of the last component occurrence.

Formally the semantics of these operators can be defined as follows:

Definition 12 (Semantics of event composition)

Let CET be a composite event type, eos an event occurrence sequence, and ET1, ET2, and ET3 event types. Then CET has an occurrence ce with component event occurrences ce.comp = $<e_1, ..., e_n>$ iff

- $\forall_{ET}$ e : e $\in$ ce.comp $\Rightarrow \neg \exists_{CET}$ ce' : ce $\neq$ ce' $\wedge$ ce' $\in$ eos $\wedge$ $e_i \in$ ce'.comp
- and one of the following hold:
  - o CET = CON (ET1, ET2, sw) $\wedge$
    ($\exists_{ET1} e_1 \exists_{ET2} e_2$: ce.comp = $<e_1, e_2>$ $\wedge$ $e_1 \in$ eos $\wedge$ $e_2 \in$ eos $\wedge$
    (T(ce) = T(e_2) iff T(e_1) < T(e_2)) $\wedge$ (T(ce) = T(e_1) iff T(e_2) < T(e_1)) $\wedge$
    (T(ce) = T(e_1) $\cup$ T(e_2) iff T(e_1) ~ T(e_2) $\vee$ T(e_1) $\Diamond$ T(e_2)) $\wedge$ e_1.wid = e_2.wid)
  - o CET = DEX (ET1, ET2) $\wedge$
    (($\exists_{ET1} e_1$: ce.comp = $<e_1>$ $\wedge$ $e_1 \in$ eos $\wedge$ T(ce) = T(e_1)) $\vee$
    ($\exists_{ET2} e_2$: ce.comp = $<e_2>$ $\wedge$ $e_2 \in$ eos $\wedge$ T(ce) = T(e_2)))
  - o CET = SEQ(ET1, ET2, sw) $\wedge$
    ($\exists_{ET1} e_1 \exists_{ET2} e_2$: ce.comp = $<e_1, e_2>$ $\wedge$ $e_1 \in$ eos $\wedge$ $e_2 \in$ eos $\wedge$
    T(e_1) < T(e_2) $\wedge$ T(ce) = T(e_2) $\wedge$ e_1.wid = e_2.wid)
  - o CET = CCR (ET1, ET2, sw) $\wedge$
    ($\exists_{ET1} e_1 \exists_{ET2} e_2$: ce.comp = <e1, e2> $\wedge$ $e_1 \in$ eos $\wedge$ $e_2 \in$ eos $\wedge$
    T(e_1) ~ T(e_2) $\wedge$ T(ce) = T(e_1) $\cup$ T(e_2) $\wedge$ e_1.wid = e_2.wid)
  - o CET = NEG (ET1, (ET2, ET3,sw) sw) $\wedge$
    ($\exists_{ET2} e_2 \exists_{ET3} e_3$: ce.comp = $<e_2, e_3>$ $\wedge$ $e_2 \in$ eos $\wedge$ $e_3 \in$ eos $\wedge$ T(ce) = T(e_3) $\wedge$ e_2.wid = e_3.wid) $\wedge$ ($\neg \exists_{ET1} e_1$: $e_1 \in$ eos $\wedge$ T(e_2) < T(e_1)< T(e_3) $\wedge$ e_1.wid = e_2.wid = e_3.wid)
  - o CET = REP (ET1, n, sw) $\wedge$
    ($\forall$ i, $1 \leq i \leq n$ $\exists_{ETi} e_i$: ce.comp = $<e_1, ..., e_n>$ $\wedge$ T(ce) = T(e_n) $\wedge$
    ($\forall$ k, $1 \leq k \leq n$: $e_k \in$ eos $\wedge$ e_1.wid = e_k.wid)

Composite event occurrences are detected at the site where the corresponding detector resides. The detection is synchronous and based on the assumption that all relevant component event occurrences with smaller timestamps will have arrived at the detector (*FIFO* delivery) [13]. Based on the previous definitions we can define the event history of the system execution at time t as follows:

Definition 13 (Event history at global time t, EH_t)

EH_t is an event occurrence sequence with n occurrences such that for all e $\in$ EH_t holds

- T(e) < t - max_sync_delay
- $\exists$ ET $\in$ $\mathcal{ET}$: e.type = ET
- $\exists$ i, $1 \leq i \leq n$: $\exists$ $ET_i \in \mathcal{ET}$: $\exists_{ETi} e_i$: $\exists$ CET $\in$ $\mathcal{CET}$: ($e_i \in EH_t$ $\wedge$ $<e_1, ..., e_{i-1}, e_i, e_{i+1}, ... e_n>$ occurs_for CET) $\Rightarrow \exists_{CET}$ ce : ce $\in$ EH_t $\wedge$ e $\in$ e.comp

Informally, an event history contains all global event occurrences in the system which occurred up to the last detector synchronization point (t - max_sync_delay). All occurrences in the event history must have a corresponding event type. Furthermore, if an occurrence can form a composite event occurrence together with other occurrences, then a corresponding composite global event occurrence must also be contained in the event history. A possible event history for the execution of the presented example is contained in Tab. 1. The event history EH_{t+1} will be constructed by appending to EH_t all primitive and composite event occurrences with timestamp t+1. The order of the appended events is not important as there are no hidden time dependencies defined in the workflow within the appended occurrences. These new composite event occurrences are determined as defined in Definition 12.

Tab. 1: Sample event history (not all fields have meaning for every event type)

type	site	eid	T	serv	wid	source	sid	plist	comp
REQ	agnt	1	13/1,10:01	accept_case	1	*initiator*		"Pay 450.- to ..."	
REQ	agnt	2	13/1,10:04	control_blacklist	1	meier		"x-rays"	
REQ	agnt	3	13/1,10:06	control_treatment	1	meier		"chest pain", "x-rays"	
CFM	clrh	1	13/1,11:15		1	RSEX	(2,meier)		
CFM	clrh	2	13/1,14:30		1	RSA	(3,meier)		
RPL	clrh	3	14/1,16:30		1	RSEX	(3,meier)	"OK"	
RPL	clrh	4	15/1,09:47		1	RSA	(2,meier)	"NO"	
CON	agnt	4	15/1,09:47		1				(clrh,3), (clrh,4)
REQ	agnt	5	15/1,11:45	print_check	1	meier		HIC	
CFM	agnt	6	15/1,11:45		1	printer	(5,meier)		
RPL	agnt	7	15/1,11:46		1	printer	(5,meier)	"job is ready"	

In a WfMS, events (e.g., participating in a composite event type) can be generated by sources at different sites. Event occurrences are detected at the site where a detector resides, which may be different from the generating sites and/or different from the interested sites (i.e., sites where brokers reacting to the event reside). The detector is responsible for recording the occurrence in the event history. This can either take place immediately after an event is signalled or after a composite event is detected, or finally at specified synchronization intervals. However, interested brokers can only react to an event occurrence after this has been recorded in the history; as only then has the global event actually occurred. Thus, the recording of event occurrences at some global time from detectors in the event history synchronizes all sites participating in the execution of a workflow. The choice of strategy depends on application requirements, the expected message propagation times between participating sites, and the granularity of time. Workflows are time-dependent but not time-critical. Thus, a conservative strategy for the synchronization of remote sites can be applied.

4 Semantics of Event-Driven Workflow Execution

4.1 Semantics of Brokers

In this chapter, and based on the formalism developed in the previous section we first define the notion of broker semantics under a certain event history. We then define those event histories in which a broker demonstrates *observably* correct behavior.

The semantics of a broker is defined by the events it generates, i.e., for a given broker, its semantics in terms of events is the restriction of the event history to those occurrences that originated in the broker:

Definition 14 (Broker Semantics)

The semantics of a broker b under an event history EH are defined as an event history EH' such that

$(\forall$ eo $\in$ EH': eo $\in$ EH $\wedge$ eo.source = b$)$ $\wedge$ $(\forall$ eo $\in$ EH: eo.source = b $\Rightarrow$ eo $\in$ EH'$)$

A broker is considered a blackbox with observable external behavior expressed by the events it generates. It is notified upon the occurrence of specific events and reacts according to its definition. We assume that each broker has an associated *event delivery interface (EDI)* and that the event delivery system is reliable, i.e. that every event occurrence will be delivered to registered brokers. Formally:

Definition 15 (Broker)

A broker (reactive component) is described by a triple brkr = (name, $\mathcal{RU}$, EDI)

- name $\in$ $\mathcal{BN}$
- $\mathcal{RU}$ = {r1, r2, ...} is a set of rules which are tuples of the following form (on_etype, $\mathcal{GENTYPES}$) with on_etype $\in$ $\mathcal{ET}$, $\mathcal{GENTYPES}$ = $\{et_1,...,et_n\}$ with $et_i \in \mathcal{PET}$.
- EDI is an event delivery interface for incoming event occurrences. Given eo_in, eo_out event occurrences, EH an event history, we define the following operations for EDI:

  put: EDI x eo_in $\rightarrow$ EDI'
  consume: EDI $\rightarrow$ eo_in x EDI'
  flush: EDI x EH $\rightarrow$ EDI' x EH' x eo_out
  contains: EDI x t x e $\rightarrow$ boolean

Informally, we make the following observations. We define for a broker its *event registration set regset* = $\cup_{r \in \mathcal{RU}}$ r.comptype(etype) i.e. the set of all (primitive and composite) event types to which the broker has to react as defined in the broker's rule set. Event occurrences are delivered to the EDI by the put operation, as soon as they are logged in the event history. We require that delivered occurrences have to be consumed by the broker within a finite amount of time (max_delay). If the occurrence consumed is a request then a confirmation event is generated. If max_delay is exceeded an exception is raised by flush. We also require that once an event occurrence has been consumed by a broker the rules which define reactions to that occurrence fire i.e., broker rules express dependencies between consumed and produced event occurrences.

Given the formal notion of event history, we can reason about the correctness of observable broker behavior. Informally, a broker demonstrates observably correct behavior (under a given event history EH_{end} resulting after the workflow execution) iff:

- it has reacted to all requests it is responsible for,

- it has reacted to all situations for which it defines workflow-specific rules, and
- it has produced either a reply or an exception after a finite time interval for each request it has received.

Formally, the correctness of the observable behavior of a broker with respect to a complete event history (EH_{end}) can be expressed as follows:

$$
\begin{aligned}
&\exists\, eo\_in \in EH_{end} : eo\_in.type \in b.\mathit{regset} \Rightarrow \\
&\quad contains(EDI, T(eo) + max\_delay, eo\_in) = \mathit{false} \\
&\quad \textbf{and} \\
&\quad \forall\, r \in b.\mathcal{RU} : eo\_in.type = r.on\_etype\ \forall\, et \in r.\mathcal{GENTYPES} \Rightarrow \\
&\qquad \exists\, eo\_out \in EH_{end} : eo\_out.source = b.name \wedge eo\_out.type = et \wedge \\
&\qquad\qquad T(eo\_out) > T(eo\_in) \\
&\quad \textbf{and if}\ eo\_in.type = REQ \Rightarrow \\
&\qquad ((\exists\, eo_2 \in EH_{end} : eo_2.sid.eid = eo\_in.eid \wedge eo_2.type = EXC \wedge \\
&\qquad\qquad eo_2.name = \text{"eo\_in.type.name unavailable"}) \\
&\qquad \textbf{or} \\
&\qquad (\exists\, eo_2, eo_3 \in EH_{end} : eo\_in.eid = eo_2.sid.eid = eo_3.sid.eid \wedge eo_2.type = CFM \wedge \\
&\qquad\qquad eo_3.type = RPL))
\end{aligned}
$$

Practically, this means that in an event history resulting after a workflow has finished executing, for all requests that have been consumed by brokers there must exist corresponding reply and confirmation occurrences (Table 1). For non-serviced or incompletely serviced requests on the other hand there have to exist exception occurrences. Given such a well-formed event history, we can argue that the components that participated in it demonstrated an observably correct behavior according to the workflow specification or at the very least aborted (parts of) its execution in a well-defined way.

4.2 Semantics of Workflow Execution

Based on the semantics of brokers operating on an event history we can now define the semantics of workflows over that history. As already mentioned in section 2, processing entities are represented by brokers and workflow activities by services. The execution of workflows by brokers is reflected in the event history. Formally, the semantics of the execution of workflow instance w is the projection of the event history on those events that occurred within w: $EH|_{wid = w}$. Based on the mapping of specifications to brokers and services, a workflow execution is correct if each responsible broker reacts and generates adequate confirmations/replies or exceptions. Thus, if the event registration set of a broker is adequately defined (with respect to the intended workflow execution semantics), the workflow execution is correct iff each involved broker behaves in an observably correct manner. Thus, the notion of correct behavior of single brokers expands to the correctness of a workflow execution: a workflow w is executed correctly under a given event history $EH|_{wid = w}$ if each broker behaves observably correctly with respect to $EH|_{wid = w}$.

During the transformation of workflow specifications into brokers, a designer has to ensure that each PE is represented by an adequate broker. "Adequacy" is hereby defined in terms of responsibility for services, rules, and replies/exceptions, all of which have to conform to the role of the processing entity in the workflow specification. The implementation of brokers on top of the event engine then guarantees that they behave observably correct under resulting event histories (whereby, of course, we cannot ensure that the workflow specification itself is correct, i.e., conforms to the intended real-world meaning). Thus, summarizing, the formalization of the event history and the corresponding broker semantics allows to reason about the semantics of event-driven workflow execution based on reactive components.

5 Related Work

The semantics of composite events in non-distributed active DBMS have been described in [2, 4, 5, 12]. The semantics of composite events in distributed systems in general and distributed DBMS in particular have been discussed in [9]. [13] proposes a general application-independent framework for the detection of composite events in distributed systems.

Event histories have also been used to specify the semantics of advanced transaction models. In [11], dependencies between events are specified using the precedence relationship and the event occurrence implication (occurrence of one event implies that of another one). Similarly, ACTA [3] is a framework in which transaction models are specified in terms of dependencies that must be fulfilled by a history in order to be legitimate under the specified transaction model. In [6], we show how ACTA-specifications can be implemented by ECA-rules.

In [1], workflow execution on top of an active (relational) DBMS is proposed. In this case, ECA-rules provide the *operational semantics* of workflows. In MENTOR, state charts are used as the representation formalism for workflow specifications [15]. These assume that the system executes a single step every time-unit reacting to all external changes that occur in the time unit elapsed since the completion of the previous step. In order to support the distributed execution of workflows, the state and activity charts are partitioned by being assigned to different business units. This transformation is proven to preserve the formally derived properties of the centralized specification.

6 Conclusion

In this paper, we considered formal aspects of event-driven workflow execution using brokers. Determining the proper semantics of all the involved components is a crucial task for any kind of WfMS. We have shown how *event histories* can be formalized, thus permitting the description of semantics of higher-level constructs in terms of event histories. We have shown how the semantics (and correctness) of reactive components can be formalized in this context. Thus, it is possible to define workflows and reactive components while knowing that the (workflow or event) engine behaves in an expected and correct way. Brokers are considered as blackboxes consuming and generating events pertaining to a workflow execution. In somes cases, it may be important to exactly understand the internal processing of broker rules by defining the semantics of broker rule execution and not just event occurrence dependencies. This is a subject of our future work.

The contribution of this paper, thus, is that it sets the stage for a formal approach to event-driven workflow execution. Furthermore, the issues discussed here are a prerequisite for analyzing effects and properties of workflow and broker *evolution*. The impact of modifications on active workflows and running components, including the proper identification of cases where evolution is safe or which adaptations have to be made in order to attain consistent executions and histories is so far not very well understood. The formal and technical issues of this kind of evolution based on the formalism developed here is a subject of our future work.

Acknowledgments: We thank Clemens Cap for his helpful suggestions.

References

1. F. Casati, S. Ceri, B. Pernici, G. Pozzi. Deriving Active Rules for Workflow Management. *Proc. 7th DEXA*, Zurich, Switzerland, September 1996.
2. S. Chakravarthy, V. Krishnaprasad, E. Answaar, S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. *Proc. VLDB*, Santiago, Chile, September 1994.
3. P.K. Chrysanthis, K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. *Proc. ACM SIGMOD*, 1990.
4. A.A.A. Fernandes, M.H. Williams, N.W. Paton. *A Logic-Based Integration of Active and Deductive Databases*. New Generation Computing, 1996.
5. S. Gatziu, K.R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. *Proc. RIDE-ADS*, Houston, TX, February 1994.
6. A. Geppert, K.R. Dittrich. Rule-Based Implementation of Transaction Model Specifications. In N.W. Paton, H.W. Williams (eds.). *Proc. 1st Intl. Workshop on Rules in Database Systems*, Edinburgh, UK, August-September 1993.
7. A. Geppert, M. Kradolfer, D. Tombros. EvE, an Event Engine for Workflow Enactment in Distributed Environments. Technical Report 96.05, Department of Computer Science, University of Zurich, May 1996.
8. S. Jablonski, C. Bussler. *Workflow Management. Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London 1996.
9. H.V. Jagadish, O. Shmueli. Composite Events in a Distributed Object-Oriented Database. In M.T. Oezsu, U. Dayal, P. Valduriez (eds.). *Distributed Object Management*. Morgan Kaufmann, 1994.
10. C.S. Jensen, J. Clifford, R. Elmasri, S.K. Gadia, P. Hayes, S. Jajodia (eds.). A consensus glossary of temporal database concepts. *SIGMOD Record*, 23(1), 1994.
11. J. Klein. Advanced Rule Driven Transaction Management. *Proc. IEEE COMPCON*, San Francisco, CA, March 1991.
12. I. Motakis, C. Zaniolo. Composite Temporal Events in Active Database Rules: A Logic-Oriented Approach. *Proc. 4th Intl. Conf. on Deductive and Object-Oriented Databases*, Singapore, December 1995.
13. S. Schwiderski. *Monitoring the Behavior of Distributed Systems*. PhD Thesis, University of Cambridge, 1996.
14. D. Tombros, A. Geppert, K.R. Dittrich. Design and Implementation of Process-Oriented Environments with Brokers and Services. In B. Freitag, C.B. Jones, C. Lengauer and H.-J. Schek (eds.), *Object-Orientation with Parallelism and Persistence*, Kluwer Academic Publishers, 1996.
15. D. Wodtke, G. Weikum. A Formal Foundation for Distributed Workflow Execution Based on State Charts. *Proc. International Conference on Database Theory*, Delphi, Greece, January 1997.