Towards Modularization and Composition in Distributed Event Based Systems

ROLANDO BLANCO, PAULO ALENCAR
David R. Cheriton School of Computer Science, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Technical Report CS-2009-08

Abstract

A distributed and interface-based publish/subscribe system is proposed in this report. Components in the proposed system react with each other via events only, and these reactions are described in the component interfaces using a variation of Harel statecharts. By encapsulating component behaviour within the interfaces, the goal of the system is to allow the study of modularization and composition mechanisms in distributed event based applications. The presentation of the proposed system is complemented by a metamodel that describes the structural, control, and runtime aspects of distributed event systems.

1 Introduction

Distributed Event Based Systems (DEBSs) are implicit invocation systems comprised of distributed functional components that interact with each other via events. Events represent happenings in the system or the environment where the system runs. Events are generated by components called *publishers*. Components interested in the events that have been generated, are called *subscribers*. A subscriber is notified when an event of interest to the component is generated by a publisher. The reaction by the subscriber is consider functionality that is implicitly invoked by the publisher component via the event.

DEBSs allow the development of applications by integrating functionality implemented by components that are autonomous and heterogeneous. Providers and users of functionality can be decided at run time, without requiring a priori knowledge of their names and locations. This low coupling between functional components make DEBS suitable for the development of applications in systems with a large number of functional components, running in different computers/devices. Such systems are expected in ubiquitous computing environments [28], as well as web environments integrating a large, or unpredictable, number of applications [14, 23].

The implicit invocation of functionality via events, as well as the autonomy, heterogeneity, and potentially large number of components make the development and maintenance of applications in DEBSs difficult. Moreover, the development of event based systems is still an ad hoc and informal process poorly supported by current software engineering methodologies [10, 21]. As observed in [21], hierarchical structuring mechanisms do not exists for the development of applications on DEBSs. In the case of UML [2], the treatment of implicit invocation is limited to annotating class diagrams and using interaction diagrams to model how system components react to events. Fiege [12] proposes to use event visibility as a structuring abstraction in implicit invocation systems. The visibility of an event determines the components that can produce and react to the event. Fiege's proposal does not include methodologies for the identification and modeling of the structural and other properties of an implicit invocation system.

In this report we propose a generic DEBS that will be the basis for our work in modularization and composition of functionality in DEBSs. Central to the generic DEBS, and to our efforts for modularization and composition of functionality in DEBSs, is the concept of reactive component interfaces. Reactive component interfaces describe the functionality implemented by the components in the system: the behaviour that causes the generation of events, and the behaviour that must be exhibited when reacting to events. Based on the work by [24] the behavioural descriptions in the reactive component interfaces are done using a variation of Harel statecharts. This variation is suitable for describing functionality executing in autonomous and heterogeneous components that communicate via events only.

Complementing the description of a generic DEBS, is a metamodel for distributed event based systems. This metamodel serves to study the structural, control, and runtime aspects of DEBS. Included in the metamodel is the enumeration of some of the relationships occurring among reactive component interfaces.

The event model for a generic DEBS is presented in section 2, including assumptions about time, event schemas, regions and access roles. Reactive Component Interfaces are introduced in section 3. Interface statecharts, a variation of Harel statecharts used in reactive component interfaces are presented in section 4. A metamodel for DEBSs is presented in section 5. We conclude the report with the presentation of related work in section 6, and future work in section 7.

2 Event Model

We assume a DEBS is a system made up of independent functional components (*components* for short) interacting with each other via events. Components are uniquely identified and each component executes in its own program space. Components may be located on separate computers (devices) communicating via a network. Components have the ability to publish (announce) events and/or subscribe to (react, be informed about) events in the system. Reactive component interfaces (*interfaces* for short) describe the functionality provided by the components. A component implements one or more interfaces.

Components are logically grouped in *regions*. Each region has a number of specialized components in charge of administrative and component interaction controlling activities.

2.1 Events

An event is a data representation of a happening in the system or the environment in which the system executes. Events are published by components via component interfaces. An event has a schema that determines, among several properties, the data attributes associated to the event. Attributes are of basic types (int, float, string, ...) or data structures defined on basic types or other data structures. An event has exactly one event schema.

Each published event in the system has a context. The context specifies the interface that produced the event, the time when the event happened or was observed, and the time when the event was published. A time-to-live (TTL), determined by the interface publishing the event, specifies the amount of time after which the event is no longer relevant to the system.

2.2 Time

Each component has a clock. Components periodically synchronize their clocks with the clocks of specialized components. These specialized components are called *time servers*. Time at a component is represented by a pair $(t, \Delta t)$ where t is the component's clock time and Δt is a upper bound estimation of the divergence between the clock of the component and the clocks of the time servers. Hence, a pair $(t, \Delta t)$ represents the time interval $(t - \Delta t)$ to $(t + \Delta t)$.

For a component, the time offset Δt should be dynamically adjusted based on the frequency of the synchronization between the component's clock and the time servers clocks, and the divergence between clocks when they were last synchronized. Providing a formula to calculate the time offset is outside the scope of this work.

2.3 Access Roles

An access role (*role* for short) characterizes a set of components [11]. Roles are uniquely identified and can be dynamically created and dropped. Roles are granted to (revoked from) components, regions, and other roles. When a role is granted to a region, every single component in the region is granted the role. Similarly, if a role is revoked from a region, every component in the region is revoked the role, even if the role was directly granted to a component in the region.

A role can be granted to another roles if both roles belong to the same region. Requiring the granted and grantee roles to be in the same region, guarantees that a region does not rescind the control over the granted role.

The role *public* is granted by default to every single component and cannot be revoked from any component. The *public* role itself cannot be dropped from the system.

Roles are created within a region by invoking the operation `addR(rname, ttl)`. Parameters are the name of the role and the amount of time the role is relevant in the system. A role is automatically dropped when a time `ttl` has passed since the role was created. Invoking `addR` for a role already in a region has the effect of updating the the TTL of the role by `ttl`. By invoking the operation `dropR(rname)`, a role can be dropped before its TTL. Roles and role membership are managed by specialized components in each region.

Role names are assumed to be unique and are used as role identifiers. Naming conventions based on some namespace hierarchy can be used to guarantee the uniqueness of role names. The problem of uniquely naming roles is outside the scope of this work.

Roles are granted to components by invoking the operation `grant(rname, cid)`, where `rname` is the name of the role, and `cid` is the identifier of the component. Roles are revoked from components by invoking `revoke(rname, cid)`. The component granting or revoking the role must have been the creator of the role. Similarly, roles can be granted/revoked to/from all components in a region by invoking `grant(rname, rgname)/revoke(rname, rgname)`, where `rgname` is the region's name.

Roles can be granted to, and revoked from, other roles effectively creating hierarchies of roles. It is assumed that there are no cycles in these hierarchies and, as previously mentioned, roles within a hierarchy must belong to the same region. The operations to invoke in these cases are `grant(rname`$_1$`, rname`$_2$`)` and `revoke(rname`$_1$`, rname`$_2$`)` where `rname`$_1$ is the role being granted (revoked), and `rname`$_2$ is the role to (from) which `role`$_1$ is being granted (revoked).

Formally, R is the set of roles, Reg is the set of regions, and C is the set of components in the system. Relations $G_R \subset R \times R$, $G_{Reg} \subseteq R \times Reg$ and $G_C \subseteq R \times C$, specify the roles granted to other roles (G_R), to regions (G_{Reg}), and directly to components (G_C). The role *public* always exists in the system, $\{public\} \subseteq R$. Since no cycles are allowed in G_R, if $(r_a, r_b) \in G_R$, then $r_a \neq r_b$ and $\neg\exists(r_1, ..., r_n)$ with $n > 2$ and $r_1 = r_a$, $r_n = r_b$, such that $(r_i, r_{i-1}) \in G_R$ for $i = 2...n$.

R_c is the set of roles a component $c \in C$ has been granted. R_c contains at least the *public* role. Formally, with G_R^+ the transitive closure of G_R, then $r \in R_c$ if:

(a) $r = public$, or

(b) $(r, c) \in G_C$, or

(c) $\exists r'$ with $(r', c) \in G_C$ and $(r', r) \in G_R^+$, or

(d) $\exists E \in Reg$, with $c \in E$, and

(d1) $\big(\ (r, E) \in G_{Reg}$, or

(d2) $\exists r'$ with $(r', E) \in G_{Reg}$ and $(r', r) \in G_R^+\ \big)$

Hence, role *public* is in R_c (expr. (a)), as well as any role directly granted to the component (expr. (b)). If a role r' has been granted to component c, then any role r granted to r' is also in R_c (expr. (c)). Since roles can also be granted to all components in a region, assuming that component c belongs to region E, then any role granted to E (expr. (d1)), or granted to a role granted to E (expr. (d2)) is also in R_c.

If the role r is not the *public* role and $(r, c) \in G_C$, then it is said that r has been *directly* granted to component c. In any other case r is said to be *indirectly* granted. A role r can be both, directly and indirectly granted to a component.

2.4 Event Schemas

Every event in the system has an event schema. An event schema specifies the data attributes that every event of the given event schema must have. Event schemas are created within a region by invoking `addES(ename, attrs, ttl)`, where `ename` is the name of the event schema, `attrs` are the data attributes associated to events of the event schema, and `ttl` is the TTL for the event schema. Event schemas are stored and maintained by specialized components in each region. The name of an event schema is assumed to be unique. As with role names, naming conventions based on some namespace hierarchy can be used to guarantee the uniqueness of event schema names. The problem of uniquely naming event schemas is outside the scope of this work.

When a period of time `ttl` has passed since an event schema was created, the event schema is removed from the region. As with roles, invoking `addES` for an event schema that already exists, has the effect of extending the TTL for the event schema by a period of time `ttl`. Event schemas can be dropped before they expire by invoking `dropES(ename)`.

2.5 Event Advertisement, Publishing, and Subscribing

Before publishing an event, a component must advertise the event to the system by invoking `adv(iname, ename, ttl)`. `iname` is the name of the interface doing the advertisement, `ename` is the event schema of the events that are being advertised, and `ttl` is the TTL for the advertisement. To avoid removal of the advertisement, the component must re-advertise the event type before `ttl`. An advertisement can be removed before it expires by invoking `unadv(iname, ename)`. Advertisement of an event is rejected by the system if the component executing the request has not been granted the access role required to implement the interface `iname` (see Section 3).

Once an event type has been advertised, a component can publish the event by invoking `pub(iname, event)`. A component wishing to react to events of a schema `ename` published via the interface `iname` must first subscribe to the events by invoking `subs(iname, ename, ttl)`. The subscription expires after a `ttl` time. Expiration of the subscription can be avoided by invoking `subs` for the interface and event schema before a period `ttl`. A component can subscribe to an event schema produced by an interface, only if it has been granted the role required by the interface to react to events of the given event schema.

2.6 Regions

Components are logically grouped in regions. Regions can contain other regions forming a tree hierarchy. The root of the hierarchy is named the *world* region. Every region has specialized components in charge of membership, role, interface, and event schema administration.

The region hierarchy can be used to guarantee uniqueness of component identifiers, role names, interface names, and event schemas names.

Reg is the set of regions in the system. The relation $\mathit{SubReg} = \mathit{Reg} \times \mathit{Reg}$ determines the region hierarchy. For regions e_p and e_c in *Reg*, $(e_p, e_c) \in \mathit{SubReg}$ specifies that region e_c is a *direct subregion* of e_p. Since the *world* region is the root of the hierarchy, then it cannot be a direct subregion of any other region: $\mathit{world} \subseteq \mathit{Reg}$ and $\neg\exists r \in \mathit{Reg}$ such that $(r, \mathit{world}) \in \mathit{SubReg}$. Also, if $(e_p, e_c) \in \mathit{SubReg}$, then:

(a) e_c is a direct subregion of one and only one region: $\neg\exists e'_p \in \mathit{Reg}$ with $e_p \neq e'_p$, such that $(e'_p, e_c) \in \mathit{SubReg}$.

(b) There is a path from the region *world* to e_c, as defined by the relation *SubReg*. Hence, $\exists e_1, e_2, ..., e_n \in \mathit{Reg}$ with $e_1 = \mathit{world}$, $e_n = e_c$, $2 \leq n \leq |\mathit{Reg}|$, such that $(e_i, e_{i+1}) \in \mathit{SubReg}$, $1 \leq i < n$.

Every region different than the *world* region is a direct subregion of another region. Hence, if $e \in \mathit{Reg}$ and $e \neq \mathit{world}$, then $\exists e' \in \mathit{Reg}$ such that $(e', e) \in \mathit{SubReg}$.

A region e_c is said to be a subregion of e, if either:

(a) e_c is a direct subregion of e: $(e, e_c) \in \mathit{SubReg}$, or

(b) e_c is an indirect subregion of e: There is a path from e to e_c as specified by the *SubReg* relation: $\exists e_1, ..., e_n \in \mathit{Reg}$ with $e_1 = e$, $e_n = e_c$, $2 \leq n \leq |\mathit{Reg}|$, such that $(e_i, e_{i+1}) \in \mathit{SubReg}$, $1 \leq i < n$.

We define the relation $\mathit{InRegion} = \mathit{Reg} \times \mathit{Reg}$, such that $(e_p, e_c) \in \mathit{InRegion}$ if e_c is a direct or indirect subregion of e_p.

2.7 Cascade Operations

Removal of event schemas and roles have cascading effects. If an event schema is removed, all related interfaces, advertisements and subscriptions will be removed as well. Publishing of events of a removed event schema are rejected by the system. Similarly if a role is removed from the system, any dependent interfaces are removed from the system as well.

3 Reactive Component Interfaces

Interfaces are at the centre of the distributed event system here proposed. Components execute one or multiple interfaces. A interface specifies the events that are published and subscribed to by every component implementing the interface, as well as the behaviour that these components must exhibit. Interfaces also specify the roles required by other components that wish to react to the events generated by the interface, as well as the role required by a component to be authorized to execute an implementation of the interface.

An interface I is represented by a tuple $< iname, In_e, Out_e, ST, r_{impl}, ttl >$, where:

- *iname* is the name of the interface. This name is unique within the system.
- In_e specifies the events the interface reacts to. I_e is a set of tuples $< i, e >$, where i is the name of an interface, and e is an event schema. $< i, e >$ indicates that the interface reacts to events with schema e generated by interface i.
- Out_e specifies the events the interface generates. Out_e is a set of tuples $< e, r, ttl >$, indicating that the interface generates events with event schema e, that role r is required by any component wishing to react to these events, and that each event with schema e generated by the interface is relevant to the system for a TTL *ttl*. If the interface imposes no restriction on the components that can react to the event, then r is *public*.
- ST is a statechart specifying the behaviour of the interface. We define the function *behaviour* such that $behaviour(I) = ST$.
- r_{impl} is the role required by a component to run an implementation of the interface.
- *ttl* is the interface's own TTL.

Interfaces are created within a region by invoking the operation `addI(iname, In`$_e$`, Out`$_e$`, ST,` $r_{\texttt{impl}}$`, ttl)`. An interface is dropped by invoking the operation `dropI(iname)`, or after the interface's TTL. Invoking `addI` for an interface already created has the effect of refreshing the TTL of the interface by `ttl`.

We define the functions *intevsin*, *intevsout*, *intbehav*, such that for an interface $I =< iname$, In_e, Out_e, ST, r_{impl}, $ttl >$: $intevsin(I) = In_e$, $intevsout(I) = Out_e$, and $intbehav(I) = ST$.

The behavioural specification of an interface is done using a statechart. The statechart represents the actions that lead to the publication of the interface events, as well as the actions taken when an event subscribed to is received. The statecharts here proposed are a variation of Harel statecharts [17], and include some of the principles for statechart composition proposed by Simons in [24], and properties of the Requirements-Oriented Statechart (ROSC) variant proposed by Glinz [15].

4 Interface Statecharts

We represent a statechart ST with a tuple $< S, s_0, s_f, L, \delta >$, where S is the set of states, s_0 is the initial state, s_f is the final state, L are the transition labels, and δ is the transition relation $\delta : S \times L \times S$. We define functions *states*, *init*, *fin*, *labels* and *trans* such that for every statechart $ST =< S, s_0, S_f, L, \delta >$, then $states(ST) = S$, $init(ST) = s_0$, $fin(ST) = s_f$, $labels(ST) = L$ and $trans(ST) = \delta$.

For a statechart ST, every state s in S is either s_0 the initial state, s_f the final state, a simple state, a history state, or a composite state. A composite state represents one substatechart, or several substatecharts operating concurrently. We call S_{hist} the set of history states, S_{simple} the set of simple states, and S_{comp} the set of composite states. $S = \{s_0\} \cup \{s_f\} \cup S_{simple} \cup S_{hist} \cup S_{comp}$, with $\{s_0\} \cap \{s_f\} \cap S_{simple} \cap S_{hist} \cap S_{comp} = \emptyset$. We also define functions *initial*, *final*, *simple*, *comp*, *hist* such that $initial(S) = s_0$, $final(S) = s_f$, $simple(S) = S_{simple}$, $comp(S) = S_{comp}$, and $hist(S) = S_{hist}$.

Each substatechart in a composite state is itself a statechart. For a state $s \in S_{comp}$, we define $substc(s)$, the set of substatecharts in s. Clearly, $|substc(s)| \geq 1$ for every $s \in S_{comp}$, otherwise the state s would not be a composite state. Substatecharts in a composite state do not share states. We define *allsubstc*, the set of all substatecharts in ST as:

$$allsubstc(ST) = \bigcup_{s \in comp(states(ST))} substc(s)$$

The set of all substatecharts in ST and, recursively, their substatecharts is defined as:

$$allsubstc^*(ST) = \bigcup_{s \in comp(states(ST))} substc^*(s)$$

where $substc^*$ is the recursive application of function *substc*, on s and all substatecharts in $substc(s)$.

Since there are no shared states, no transition goes from one state in one substatechart to another state in another substatechart: if $|substc(s)| > 1$, then $\forall ST', ST''$ with $ST' \in substc(s)$, $ST'' \in substc(s)$ and $ST' \neq ST''$, $states(ST') \cap states(ST'') = \emptyset$.

Final states in a substatechart do not halt the execution of the statechart containing the substatechart reaching its final state. In other words, final states are not halting states. Note this is different than UML, in which a final state may be a halting state.

4.1 Remote Substatecharts

An interface may compose the behaviour of other interfaces. For a statechart ST we define the function $remotesubstc(ST)$ as the substatecharts in ST representing behaviour from other interfaces. Formally, assuming IS to be the set of all interfaces, and having the interface $I \in IS$ with behaviour $ST = intbehav(I)$, then $ST' \in remotesubstc(ST)$ if $\exists I' \in IS$ such that $intbehav(I') = ST'$. Similarly we define the function $allremotesubstc(ST)$ as the statecharts in $allsubstc^*(ST)$ representing behaviour from other interfaces.

Graphically, a notation shall exist that indicates whether a substatechart is remote, and whether a composite state contains remote substatecharts. If a substatechart is remote, the notation should also include the name of the interface being composed.

4.2 Transitions

The transition labels of a statechart are determined by the relation $L = 2^E \times G \times A$, where E is the set of events in the statechart. G are guard conditions on local variables V, plus *true* which is used to indicate that the transition label does not impose conditions. A is the set of uninterruptible actions, plus the symbol $\perp$. $\perp$ is used to represent that no uninterruptible action is taken. We define functions *events*, *guards* and *vars* on L, such that $events(labels(ST)) = E$, *guards*(*labels*(ST)) $= G$, and *vars*(*labels*(ST)) $= V$. Also, for a label $l = (N, g, a)$, with $N \subseteq E$, $l \in L$, we define functions *levents*(l) $= N$, *lguard*(l) $= g$, and *laction*(l) $= a$.

In most cases, the set of events N specified in a transition label will contain only one event. Multiple events $N = e_1, ...e_n$ with $2 \leq n < |E|$, trigger a transition only after all events $e_i \in N$ have occurred, and no event $e_j \in N$ with $e_j \neq e_i$ has expired before e_i occurs – recall that events expire after an associated TTL (see Section 2.1). As will be shown later, multi-event triggered transitions are used to synchronize multiple concurrent substatecharts in a composite state (see Section 4.5).

Two different substatecharts in an interface can refer to the same local variable in their transition labels only if the substatecharts are both local to the interface. Hence, for a interface statechart ST, with both ST' and ST'' in *allsubstc**(ST), if *vars*(*labels*(ST'))$\cap$*vars*(*labels*(ST'')) $\neq \emptyset$, then $ST' \notin$ *remotesubstc**(ST) and $ST'' \notin$ *remotesubstc**(ST). This requirement guarantees that communication between different interfaces is not performed using shared variables.

4.3 Internal and Distributed Events

In a statechart, we distinguish between events that are generated internally by the component for its own functioning and *distributed events* that are advertised/published/subscribed to via invocations to the `adv`, `pub`, `subs` operations. Internal events are broadcasted within an interface, while distributed events are communicated to other components subscribed to the event. Therefore, distributed events are primarily used for inter-component interaction.

For a statechart ST, part of an interface $<$ *iname*, In_e, Out_e, ST, r_{impl}, *ttl* $>$, we define the functions *intevents* and *distevents*, such that *intevents* determines the events in $E =$ *events*(*labels*(ST)) that are internal events, and *distevents* determines the events in E that are distributed events. $E =$ *distevents*(E) $\cup$ *intevents*(E) and *distevents*(E) $\cap$ *intevents*(E) $= \emptyset$. Also, if an event is a distributed event, then there is a state in the statechart or any of its substatecharts such that the event is announced and published, or subscribed to. Note that we assume that operations `adv`, `pub`, `subs` are interruptible operations and, therefore, need to be modeled as states in a statechart.

More formally, if $e \in$ *distevents*(*events*(*labels*(ST))), and R is the set of roles in the system, then:

(a) Advertisement and publishing of e is done in a simple state of the statechart: $\exists r, ttl, s_i, s_j$ with $r \in R$, $< typeof(e), r, ttl > \in Out_e$, $s_i \in$ *simple*(*states*(ST)) and $s_j \in$ *simple*(*states*(ST)), such that $s_i \equiv adv(typeof(e))$ and $s_j \equiv pub(e)$, or

(b) The event e is subscribed to in a simple state of the state chart: $\exists i, s$ with $< i, typeof(e) > \in In_e$, $s \in$ *simple*(*states*(ST)), such that $s \equiv subs(i, typeof(e))$, or

(c) Advertisement and publishing of e, or its subscription is done in a substatechart: $\exists s, ST'$ with $s \in$ *comp*(*states*(ST)) and $ST' \in$ *substc**(s), such that:

  (c1) $\exists s_i, s_j$ with $s_i \in$ *simple*(*states*(ST')) and $s_j \in$ *simple*(*states*(ST')), such that $s_i \equiv adv(typeof(e))$ and $s_j \equiv pub(e)$, or

  (c2) $\exists i, s$ with $< i, typeof(e) > \in I_e$, $s \in$ *simple*(*states*(ST')), such that $s \equiv subs(i, typeof(e))$.

Where *typeof*(e) is a function that specifies the event schema of an event e. Recall that *substc**(s), is the recursive application of function *substc*, on s and all substatecharts in *substc*(s).

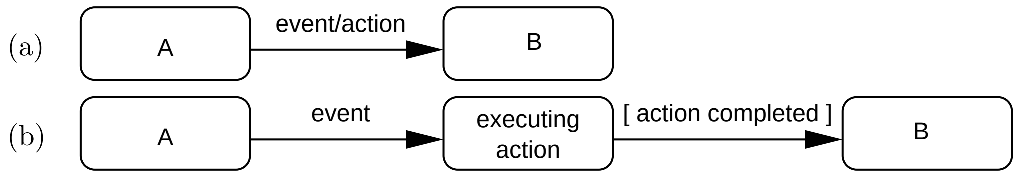


Figure 1: Synchronous and Asynchronous Event Processing

We are not the first ones to propose a separation between the event model used for the internal behaviour of a component and the event model used for inter-component interactions. For example, in RSML, the Requirements State Machine Language [18], events are broadcasted within a statechart modeling a single physical component. Inter-component interaction in RSML is modeled via addressed (directed) events: a component executes a `send` operation that specifies the component intended to receive the event. The recipient receives the event by executing a `receive` operation. Addressing of events has the disadvantage of increasing the coupling between components. This is because components need to be aware about what other components are interested in their events. In contrast to RSML, the use of a publish/subscribe mechanism, as proposed in our event model, provides inter-component communication via unaddressed (also known as undirected) events.

If a statechart has states representing behaviour from other interfaces (remote substatecharts), only distributed events can be used to communicate with the remote substatecharts. Hence, internal events can be reacted to only by states representing local behaviour. Formally, for every statechart ST, and every substatechart $ST' \in allsubstc^*(ST)$:

$$inteventss(events(labels(ST))) \cap intevents(events(labels(ST'))) = \emptyset$$

4.4 Asynchronous Event Processing

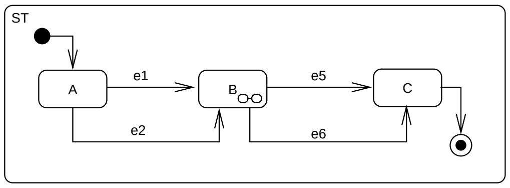
In contrast to Harel statecharts, events in interface statecharts are processed asynchronously: a component may not react instantaneously to an event, and the reaction may take time to execute. Figure 1a shows part of a statechart with states A and B. In this figure, the statechart reacts instantaneously to the event *event* by executing the action *action*. *action* executes instantaneously and, therefore, is uninterruptible. Figure 1b shows the same part of the statechart with the assumption that events are handled asynchronously. Like UML statecharts [16], in our statecharts, events are queued until the statechart is ready to process them. Unlike Harel and UML statecharts, a state is required to model the execution of the reaction to the event. Hence the statecharts here used are closer to flowcharts than other statecharts proposed elsewhere [24].

4.5 History, Variables, and Interlevel Transitions

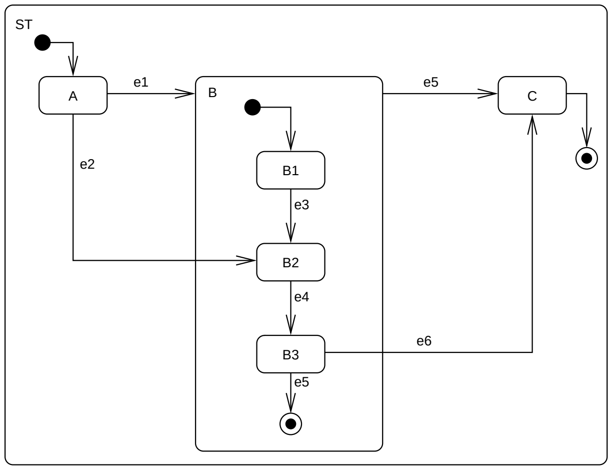
History and deep history states are allowed.

There is no shared memory between components. Hence, variables are only supported for statecharts modeling the behaviour of non-derived components. Guard conditions on state transitions can only specify conditions on local variables.

In order to be able to analyze the behaviour of component interfaces at different levels of containment, it is required that the statecharts modeling the behaviour be encapsulated. In other words, it should be possible to analyze a substatechart independently of the operation of the statechart containing the substatechart. Conversely, it should be possible to analyze a



(a)



(b)

Figure 2: Statechart Boundary-crossing

statechart without knowing the operation of its substatecharts. Simons [24] enumerates several properties that statecharts must have to encapsulate behaviour. From these properties, we require the following:

- *Boundary-crossings transitions are not allowed.* Transitions do not lead directly from or to substates of composite states. Hence, for a statechart ST we have that $\forall ST' \in allsubstc^{*}(ST)$, $states(ST) \cap states(ST') = \emptyset$. The statechart in Figure 2 illustrates why boundary-crossing transitions break statechart encapsulation. In Figure 2a states A and C are simple states, while state B is a composite state. Figure 2b shows the same statechart with the substatechart for state B expanded. There are two boundary crossing transitions in this statechart: from state A to state $B2$ when event $e2$, and from state $B3$ to C when event $e6$. The transition from A to B when $e2$, prevents the analysis in isolation of the statechart in 2a. This is because in order to analyze the statechart one needs to be aware of the substate $B2$, a state not at the same level of abstraction as A, B, or C. The other boundary crossing transition, from state $B3$ to C, prevents the analysis in isolation of the substatechart for B, since one needs to be aware of state C. In this case C is not a the same abstraction level as $B1$, $B2$, or $B3$.

- *Events are generated at final states.* When a substatechart reaches a final state, it must generate an event that lets the outer statechart know that the substatechart has reached a final state. Formally, $\forall ST' \in allsubstc(ST)$, then $\forall (s, l, s_f) \in trans(ST')$, with $s_f \in final(states(ST'))$, and $l = (N, g, a)$, $N \subseteq events(labels(ST))$, then $N \neq \emptyset$. These events are called "outcome events". As an example, in Figure 3, events $o1$, and $o2$ are the outcome events for the substatechart in state B.

  Figure 4 shows the case when a composite state contains two substatecharts operating concurrently. In this example, state C is reached only after both substatecharts in B reach their final states. The transition from B to C is triggered after both $o1$ and $o2$ occur. If $o1$ and $o2$ occur at different times, the state C is not reached if the TTL of the first occurring event is reached before the other event occurs.

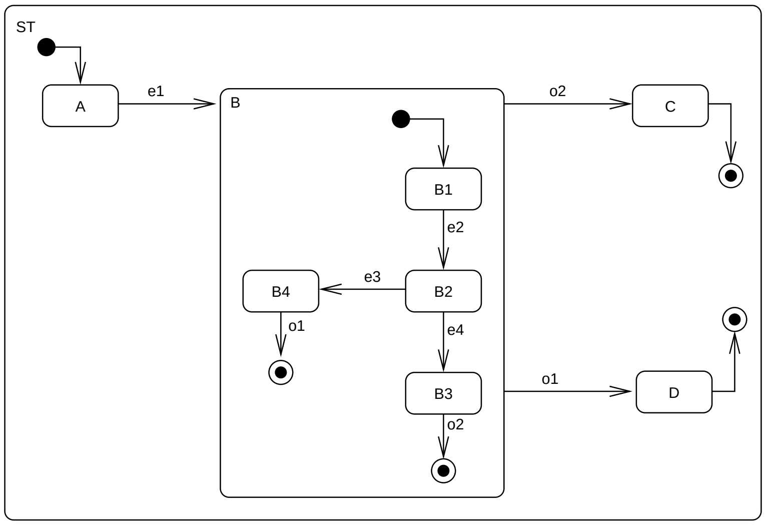


Figure 3: Events Generated at Final States

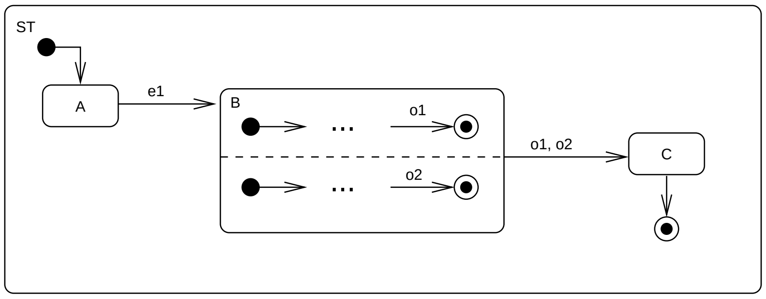


Figure 4: Substatechart Synchronization

- *Transitions from composite states can only be triggered by outcome events.* In UML and Harel statecharts it is possible to have a transition leaving a composite state, where the event triggering the transition is not an outcome event. Figure 5a shows an example of a statechart with a transition from the composite state B to state C triggered by an event $e4$ that is not an outcome event of any substatechart in B. In both UML and Harel's statecharts, this construct is almost equivalent to a transition from every substate within the composite state, to the state the transition leads to (Figure5b). It is not completely equivalent because in the statechart in Figure 5b there is nondeterminism if the substatechart for B is in state $B2$ and both events $e3$ and $e4$ occur. On the other hand, the statechart in Figure 5a is deterministic. This is because both UML and Harel handle events $e3$ and $e4$ with different priority. In the case of Harel, state C is selected as the next step. UML reverses the priority, and state $B3$ is selected instead. By giving priority to the inner state $B3$ over the outer state C, UML statecharts are difficult to analyze. The reason is that, to decide the next state in a statechart, it is required to inspect any relevant substatecharts. In Harel statecharts, there is not such problem, since the outer state will always have the priority over any active substate. In any case, we disallow such a construct since it contains boundary crossing transitions. Formally, for a statechart ST and $\forall(s_i, l, s_j) \in \mathit{trans}(ST)$ with $l = (N, g, a)$, if $s_i \in \mathit{comp}(\mathit{states}(ST))$, then $\forall e \in E$,

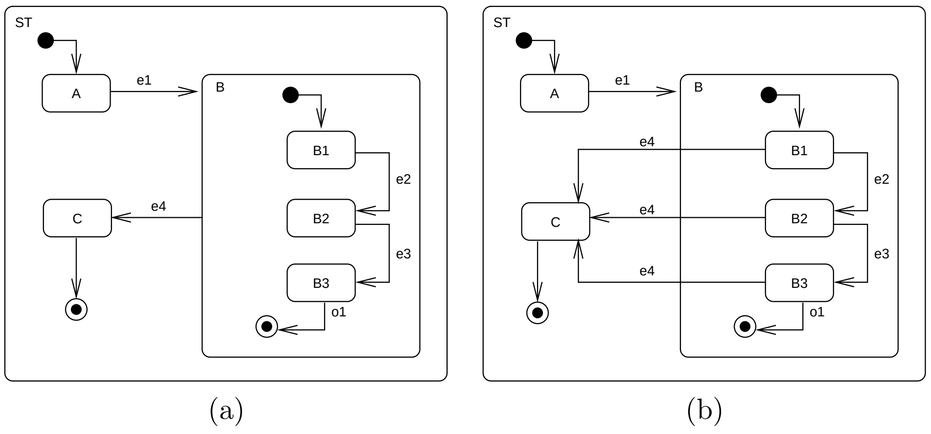


Figure 5: Transitions from Composite States

$\exists ST' \in substc(s_i), (s', l', s'_f) \in trans(ST'), s'_f \in final(states(ST')), l' = (N', g', a')$ such that $e \in N'$.

Note that since all substatecharts are statechart themselves, these requirements propagate down all levels of statechart containment.

4.6 Other Statechart Considerations

We assume no free boundary transitions. Hence, each transition must be labeled with an event and/or condition and/or action. Note any action in a transition is assumed to be uninterruptible.

5 Distributed Event System Metamodel

Based on the distributed event model described in the previous sections, and some of the distributed event systems proposed elsewhere [4, 22], this section introduces a metamodel for distributed event systems. Structural and control models are used to describe the reactive components interfaces in the system. A runtime model describes the entities in the system while in operation. Finally, the relationships between interfaces are described.

5.1 Structure and Control

Components in a DEBS exhibit behaviour described by reactive component interfaces. Structurally, these interfaces specify the distributed events the components generate and react to. Each interface also has a statechart that describes the execution of the component implementing the interface as it pertains to the generation of, and reaction to, distributed events. The UML class diagram in Figure 6, shows the structure of the interfaces. Distributed events are themselves composed by data attributes of predetermined data types.

Actual code (e.g. library, program, module, etc) implementing an interface is represented in Figure 6 by the *interface implementation* class. A given interface may have multiple implementations, and these implementations are run by the components.

Interfaces may derive some of their behaviour from other interfaces. Section 5.3, list some of the relationships that occur when interfaces are composed. Behaviour specified by a statechart can compose behaviour from other statecharts as well.

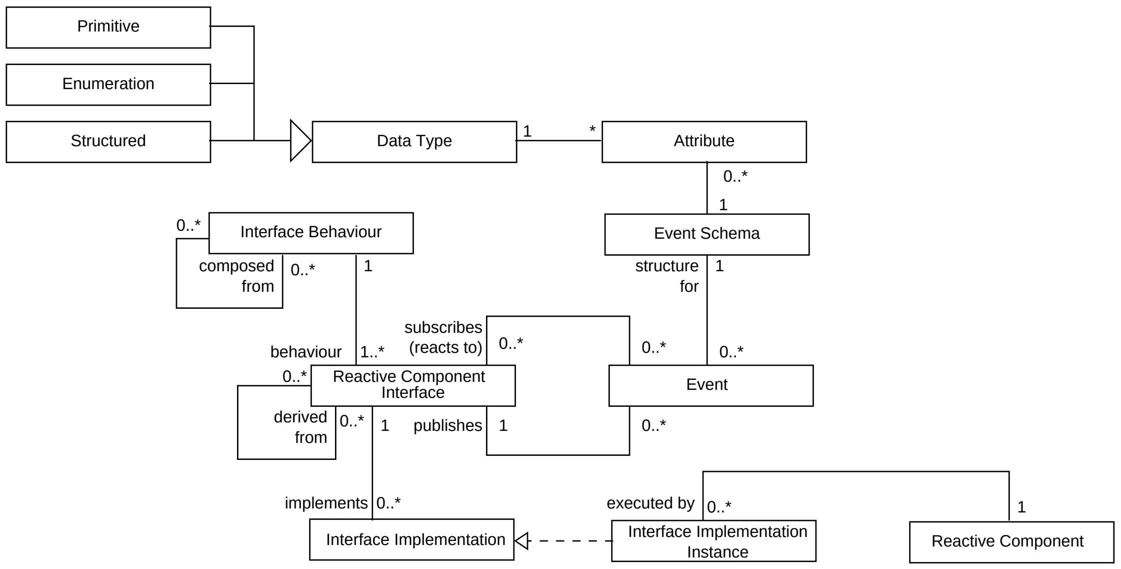


Figure 6: DEBS Metamodel - Structural View

Logically, components are grouped in regions. As shown in Figure 7, a region is always within another region, forming a tree-like hierarchical structure. At the root of the hierarchy is the “world” region (not shown in the Figure). Two kinds of components exist in the system: application and coordination components. Application components run application-specific interface implementations that are not related to the basic operation of the DEBS and its services. Coordination components, on the other hand, are in charge of the administrative activities in the DEBS. We assume that these administrative activities are implemented via events, although some systems may provide an explicit (synchronous RPC-like) implementation. Irrespectively of their implementation, administrative activities include the management of component membership, roles, event schemas, and interfaces.

Component membership services allow components to discover, join, and leave regions. Role coordination services, allow components to create and delete roles, and to grant (revoke) roles to (from) other components, regions, and roles. Granting a role to a region is equivalent to granting the role to every component in the region. Also, every component that joins the region will be assumed to have been granted the role, until the role is revoked from the region.

Roles are used to restrict which components can execute interface implementations, and which components can react to events generated by an interface implementation. Only components that have been granted the required roles, are permitted to execute these functionalities. Enforcement of the required roles is done by the role coordination services.

As shown in Figure 7, a role can be granted to another role. The restriction that both roles belong to the same region is assumed in this case. The reason for this restriction, is that if a role r in region reg is granted to a role r' in another region reg', then the region reg where the granted role r is defined looses control of the role. This is because the role r can then be indirectly granted to a component c' in region reg', if the role r' is granted to c.

Event schema coordination services allow the creation, deletion and modification of event schemas. Interface coordination services provide similar functionality with regard to interfaces.

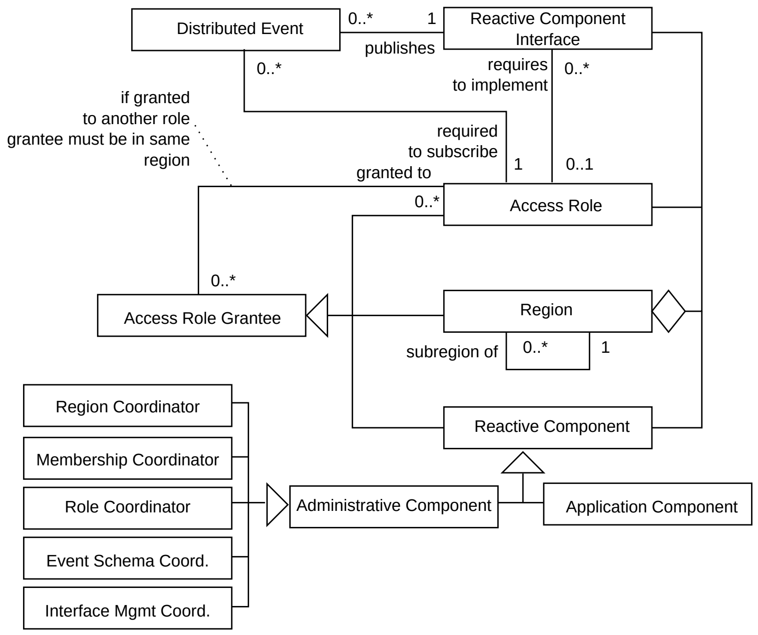


Figure 7: DEBS Metamodel - Control View

5.2 Runtime

Execution of a DEBS starts by launching the components in charge of managing the “world” region. Once the main region has been setup, components can join the region. These components can create other regions, roles, event schemas, and interfaces. Time-to-live values are specified when event schemas and interfaces are created. Event schemas and interfaces are automatically removed from the systems after their TTLs. TTLs, specified when the interface is created, indicate the TTL of each event generated by implementations of the interface (Figure 8).

Components run on hardware nodes. Each node has a clock that is periodically synchronized with time servers in the system. Although, this does not guarantee that all nodes’ clocks will run in sync, it is assumed that their clocks are running close to each other. Some of the operations executed by components are listed in Figure 8. These operations are executed via interface implementations.

5.3 Interface Relationships

The simplest relationship between components in a DEBS is event visibility: two components are related if they can potentially react to the same events [12]. In the event model here proposed, event visibility is related to the concept of access roles granted to the components. Hence, two components have the same event visibility if the roles they have been granted allow them to subscribe to the same events. Therefore, event visibility is, at least in our proposal, an access based relationship.

Since components implement reactive component interfaces, it is possible to analyze the relationships between components at an interface level. This analysis can be done by looking at the actions that interfaces perform on events. Another option is to analyze the interactions between interfaces when implementing application-level functionality. Interface relationships defined based on event actions are called event-centred relationships. Interface relationships

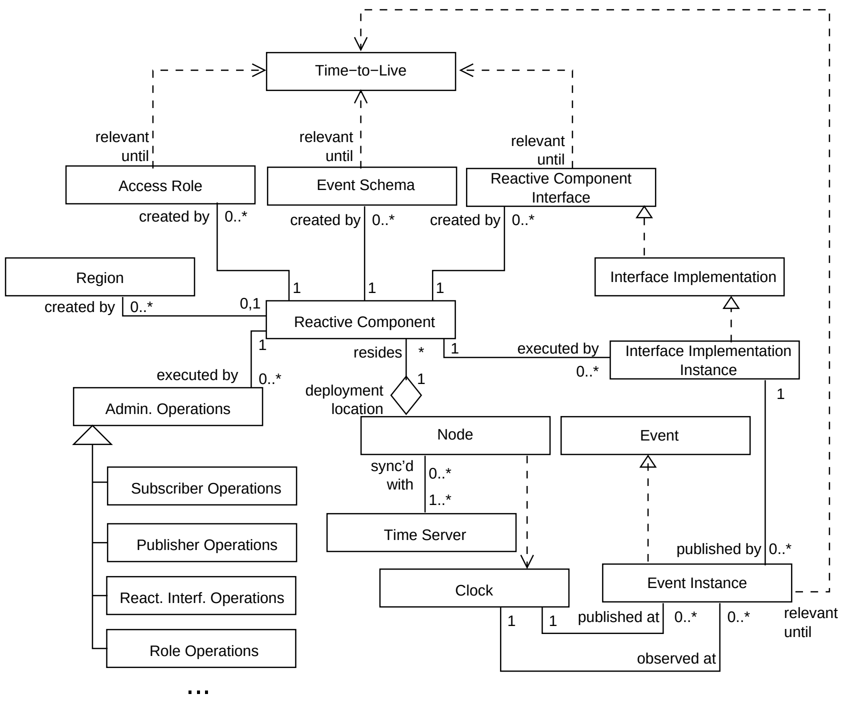


Figure 8: DEBS Metamodel - Runtime View

based on interactions are called structural relationships.

5.3.1 Event-Centred Relationships

Relationships between interfaces can be defined based on the actions that the interfaces perform on the events. Interfaces can generate, forward, filter, transform, and consume events:

- A generator interface for an event, is an interface that is able to sense a happening from the environment or its own state and generates (publishes) an event to represent this happening.
- A forwarder interface reacts to an event by publishing another event with exactly the same event schema as the original event.
- A filter interface, reacts to events by publishing another event with exactly the same event schema as the original event if a condition holds. This condition is typically expressed in terms of the received event itself or its context, but it may also include conditions on the environment, the interface's own state, or other previously received events.
- A transformer interface reacts to an event by publishing one or multiple events with event schemas resulting from the transformation of the original schema. Possible event transformations are: translation, aggregation, splitting, and enrichment [8]. Translation occurs when the data in the event is modified to another representation. Semantically, the translated data may or may not be equivalent to the original data. Aggregation represents the case when data from more than one event is aggregated into the event schema of another

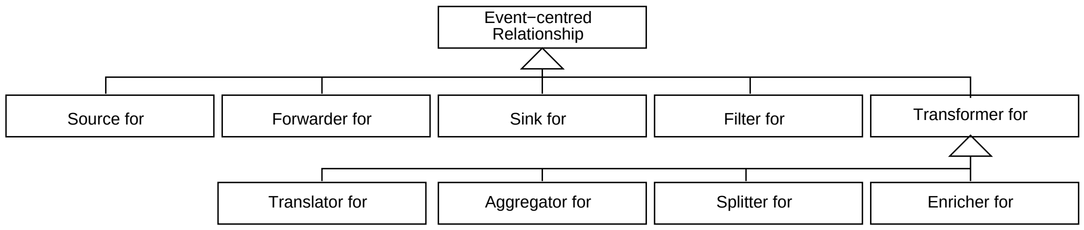


Figure 9: DEBS Metamodel - Event-Centred Relationships

event. Splitting occurs when data from one event is republished as several events. Enrichment occurs when the data from the received event is complemented with data from other sources.

- A sink interface for a given event, is an interface that does not publish other events as a reaction to the received event.

Relationships between interfaces can then be defined based on these actions. Figure 9 shows the relationships. The simplest relationship between two interfaces is the *source-for* relationship. This relationship occurs when one interface is the source of events that another interface is subscribed to.

5.3.2 Behavioural and Structural Relationships

Interface relationships can be characterized by studying the compositions between interfaces when carrying out application level functionality. The UML class diagram in Figure 10 shows the relationships:

- The simplest and most common relationship is the *reacts to* relationship. One interface $I =< iname, In_e, Out_e, ST, r_{impl}, ttl >$ reacts to another interface $I' =< iname', In'_e, Out'_e, ST', r'_{impl}, ttl' >$ if there is at least one event generated by I' that I subscribes to, and the behaviour ST' of the interface I' specifies a reaction to the event. Formally, $\exists o'_e =< e', r', ttl' >$ with $o'_e \in Out'_e$ such that $< e', iname' > \in In_e$.

- One interface I *delegates* the handling of an event to another interface I', when the interface I, forwards all received events from an interface I'' with a particular event schema, to I'. If the interface I' reacts to events from I only, and all events published by I' are only handled by i, then I *encapsulates* the behaviour of I'.

- One interface I *extends* another interface I', when the functionality provided by I is a superset of the functionality provided by I' and the behaviour of I matches the behaviour of I' with regards to the events it generates and the events it reacts to.

- Interfaces exhibit some of the part-whole relationships found elsewhere [29, 20, 1]. In particular, an interface I may implement functionality required by another interface I' to carry out a particular activity. In such a case I is *member-of* the activity being implemented by I'. Note there is no exclusivity nor life-cycle requirements in this relationship. Also note that this relationship is transitive. When the interface I exists only while I' exists, and its functionality is only performed for I', then there is an *integral part-of* relationship.

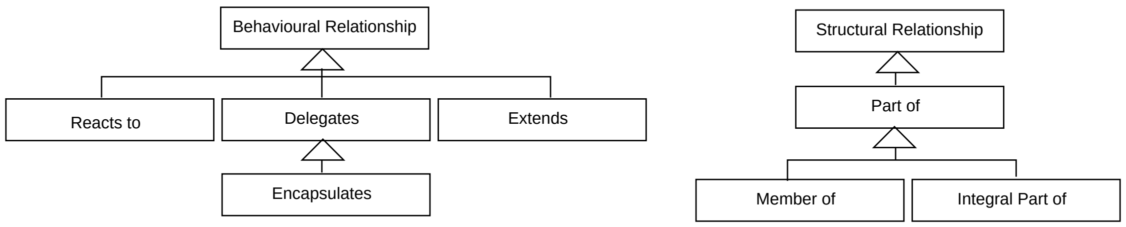


Figure 10: DEBS Metamodel - Behavioural/Structural Relationships

Some of these relationships model interactions occurring in other types of systems. For example, the delegation relationship is similar to the relationship that exists between a class in its superclass in object orientation. In OO, method invocation is delegated from a class to its superclass when the subclass does not provide an implementation for the method. In the case of DEBSs, the delegation models the case when the reaction to the event is delegated from one interface to another.

5.4 Other Relationships

Besides interface relationships, many other relationships occur in DEBSs, including between regions, between regions and interfaces, and between components and interfaces.

The simplest relationship between regions is the *contained in* relationship, used to represent the fact that a region is immediately contained within another region. A region can only be immediately contained within one other region. Other relationships between regions can be established when administrative components of one region server as administrative components of other regions. Similarly, event schema and interface definitions in one region may be used by interfaces in other regions. For example, one can envision widely used event schema definitions in the "world" region being available to all other regions. In this case, relationships can be established between the regions and the interfaces that use the event schemas defined within those regions.

Components create interfaces and run interface implementations. Besides these obvious relationships, the context provided by a component (e.g. its location), may influence the behaviour of an interface. The study of the relationships between components and interfaces is part of our ongoing investigation into composition of reactive component interfaces.

6 Related Work

Several models have been proposed to represent event systems and the composition of functionality in these systems. Some representative works use Process Algebra [7, 9], Interface Automata [6], and variations of Finite State Machines [3]. These works differ from our proposal in the types of event systems they model, and/or the communication mechanisms assumed between components. For example, [7] assumes an event models with centralized dispatching of events, whilst [9] allows communication of components via global variables. [9] is interested in computing the behavioural specification of the whole system, while we are interested in the the ability to model component behaviour and their compositions.

With regards to structuring DEBSs, we are only aware of the proposal to use event visibility [13] as an structuring abstraction.

Process Algebra and Trace Semantics In [7], Dingel et al. recognize the lack of a established methodology for specifying and verifying systems where the invocation of functionality is done via events ("implicitly" is the term used by Dingel and other researchers). Dingel et al. argue that such a methodology should be compositional: the verification of a component should be decoupled from the verification of the system in which its events are bound to other components. Otherwise changes to the binding between publishers and subscribers would require re-verification of the components that publish the events [1].

In the same work, Dingel et al. propose a formal model for the compositional verification of synchronous implicit invocation systems. In their model, a system S consist of a set of methods M. A distinguished dispatcher method $disp \in M$ stores and delivers the events. Another distinguished method $m_{Env} \in M$ represents the environment where the system executes. A method $m \in M$ is a UNITY program [5]. Actions in the method, can be regular UNITY actions plus communication actions that allow the method to announce and consume events. Behaviour of each method is represented by an automaton. First order linear time temporal logic without the next operator is used to specify the ongoing behaviour of the system.

In order to define the semantics on an event, the environment is constrained to be a method that just announces the event. Syntactic conditions on the variables used by each of the methods are used to specify groups of methods that are independent. The idea is that two sets of methods are independent if the methods in one group do not mention the variables used by the methods in the other group. The intention of this definition, is to find independent groups of methods, so that reasoning made within each of the groups can be generalized to the whole system. Hence, a verification of an independent group implies the same verification for the whole system. This is in essence the compositional reasoning that Dingel et al. argue a verification formalism for implicit invocation systems should have.

Dingel's work applies to centralized synchronous systems. No consideration is made for semantical issues in event-based systems related to delivery policies, event-type issues, event distribution guarantees, nor event composition. It is not clear if by considering each event in isolation, the semantics of the event can be defined/studied. This is related to the fact that in Dingel's work, an event cannot cause the announcement of other events.

Interface Automata Interface Automata [6] has been used to describe the behaviour of reactive systems [25, 26]. Interface Automata is an automata-based language used to capture both input assumptions about the order in which a component reacts to events, and output guarantees about the order in which the component generates events. Interface compatibility is decided based on an *optimistic* approach. In a traditional pessimistic approach two components are compatible if they can be used together in all systems. In the optimistic approach proposed with Interface Automata, a helpful environment is assumed: two components are compatible if they can be used together in at least one design. The advantage of the optimistic approach is a simpler model. Interface automata interact through the synchronization of input and output events. Internally, actions of concurrent automata are interleaved asynchronously. This makes them different than our proposed statecharts, where both the interaction between interfaces and internal substatecharts is done asynchronously. Another major difference with our work, is their assumption that messages are not queued. In Interface Automata, the arrival of a message while on an state not prepared to handle the message, would indicate an incompatibility between the environment and the automaton. In statecharts, on the other hand, the message would be

[1]The terms "announce" and "register" are used by Dingel for "publish" and "subscribe"

queued until the statechart is at a state ready to handle the message.

Finite State Machines In [3], Bultan et al. analyze component composition by looking at the conversations between the components. A conversation is the concatenation of all the events exchanged by the components being composed. The behaviour of the components themselves is represented by Mealy machines [19]: finite state machines with input and output. A component is then viewed as a Mealy machine that decides, based on the received events and the events already sent, if a new event should be sent. In contrast to interface automata, and similarly to our statecharts, Mealy machines interact asynchronously. But in order to perform the analysis of the compositions, it is required to have a global watcher that keeps track of all events as they occur. The authors start by trying to deduce global behaviour by analyzing the behaviour of the components. They find this bottom-up approach flawed and propose to perform a top-down approach instead. Their argument is that given a conversation, it is not possible to find a regular language ("global behaviour") as its core. This is because of the asynchronous nature of the interactions. In the top down approach, on the other hand, they start with conversations that represent the intended *global* behaviour of the system, and construct Mealy machines that realize that conversation. Similarly to our work, the authors final goal is to understand component composition in distributed systems. Our approach diverges from theirs since our focus is to study the composability of functionality in DEBSs, instead of deducing a global (local) behaviour based on a local (global) behaviour.

Structuring DEBSs Fiege et al. propose to use event visibility, and its abstraction *scopes* as a mechanism for encapsulation and information hiding in event-based systems [13, 12, 21]. A scope is a set of event producers and consumers. Visibility of events produced within a scope is limited to the consumers in the same scope. Scope interfaces allow the exchange of event notifications with the rest of the system. In our proposal, event visibility is considered an access restriction issue instead of a engineering construct. Moreover, although by doing a mapping between scopes and regions one could implement a scope-based system in our generic DEBS, we consider that a more essential analysis of the relationships between components in a DEBS is required. Beyond whether or not two components are within the same scope, we are more interested in studying how components interact via events, and how functionality is composed based on those interactions.

7 Conclusion and Future Work

We have presented a generic DEBS to be used in the study of modularization of applications where functionality is composed via events only. Interfaces specify not only the events generated and of interest to components, but the behaviour that a component must exhibit when the events are generated and received. *Gem*, an initial implementation of the DEBS is described in [27].

Based on the proposed metamodel and the initial DEBS implementation, we are currently working towards formalizing the DEBS computational model. Our intention is to use the formalization to represent the composition of functionality in DEBS applications, and in combination with the DEBS implementation, to study the properties of these compositions.

References

[1] Franck Barbier, Brian Henderson-Sellers, Annig Le Parc-Lacayrelle, and Jean-Michel Bruel. Formalization of the whole-part relationship in the unified modeling language. *IEEE Transactions on Software Engineering*, 29(5):459–470, 2003.

[2] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series).* Addison-Wesley Professional, 2005.

[3] Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 403–410, New York, NY, USA, 2003. ACM Press.

[4] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[5] K. Mani Chandy and Jayadev Misra. *Parallel program design: a foundation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[6] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *ACM SIGSOFT Software Engeneering Notes*, 26(5):109–120, 2001.

[7] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 209–221, New York, NY, USA, 1998. ACM Press.

[8] Opher Etzion. Semantic Approach to Event Processing. In Hans-Arno Jacobsen, Gero Mühl, and Michael A. Jaeger, editors, *Proceedings of the Inaugural Conference on Distributed Event-Based Systems*, page 139, New York, NY, USA, 2007. ACM Press. Invited talk `http://www.debs.msrg.utoronto.ca/etzion.pdf`.

[9] Pascal Fenkam, Harald Gall, and Mehdi Jazayeri. A systematic approach to the development of event based applications. In *22nd Symposium on Reliable Distributed Systems (SRDS 2003)*, page 199. IEEE Computer Society, 2003.

[10] Pascal Fenkam, Mehdi Jazayeri, and Gerald Reif. On methodologies for constructing correct event-based applications. In Antonio Carzaniga and Pascal Fenkam, editors, *3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, pages 38–43, Edinburgh, Scotland, UK, May 2004. IEEE.

[11] David Ferraiolo and Richard Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[12] Ludger Fiege. *Visibility in Event-Based Systems.* Ph.d. thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, April 2005.

[13] Ludger Fiege, Mira Mezini, Gero Mühl, and Alejandro P. Buchmann. Engineering event-based systems with scopes. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 309–333, London, UK, 2002. Springer-Verlag.

[14] Kurt Geihs. Middleware challenges ahead. *Computer*, 34(6):24–31, 2001.

[15] Martin Glinz. Statecharts for requirements specification - as simple as possible, as rich as needed. In *Proceedings of the First ICSE International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'02)*, Orlando, USA, May 2002.

[16] Object Management Group. UML 2.1.1 superstructure specification. `http://www.omg.org/technology/documents/formal/uml.htm`, February 2007. Chapter 15: State Machines.

[17] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[18] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

[19] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 35(5), 1955.

[20] Renate Motschnig-Pitrik and Jens Kaasbøll. Part-whole relationship categories and their application in object-oriented analysis. *IEEE Transactions on Knowledge and Data Engineering*, 11(5):779–797, 1999.

[21] Gero Mühl, Ludger Fiege, and Peter R. Pietzuch. *Distributed Event-Based Systems.* Springer-Verlag, 2006.

[22] Peter Robert Pietzuch. *Hermes: A Scalable Event-Based Middleware.* PhD thesis, University of Cambridge, Queens' College, February 2004.

[23] David S. Rosenblum and Alexander L. Wolf. A design framework for internet-scale event observation and notification. In *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 344–360, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[24] Anthony J H Simons. On the compositional properties of UML statechart diagrams. In *Proceedings of the Third Workshop on Rigorous Object-Oriented Methods, (ROOM2000)*, Electronic Workshops in Computing (eWiC), York, UK, January 2000. The British Computer Society (BCS).

[25] Margus Veanes, Colin Campbell, Wolfram Schulte, and Pushmeet Kohli. On-the-fly testing of reactive systems. Technical Report MSR-TR-2005-05, Microsoft Research, January 2005.

[26] P. Völgyesi, M. Maróti, S. Dóra, E. Osses, and Á. Lédeczi. Software composition and verification for sensor networks. *Sci. Comput. Program.*, 56(1-2):191–210, 2005.

[27] Jun Wang. An interface based distributed event system. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 2008.

[28] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993.

[29] Morton E. Winston, Roger Chaffin, and Douglas Herrmann. A taxonomy of part-whole relations. *Cognitive Science*, 11(4):417 – 444, 1987.