Event Logic for Specifying Abstract Dynamic Data Types[⋆]

Gianna Reggio

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova – Italy

Introduction

Abstract and concrete dynamic data types, have been introduced by the author and others in several previous papers (e.g., [2, 3, 6, 10]).

A concrete dynamic data type (dynamic algebra) is just a many-sorted algebra with predicates such that for some of the sorts (sorts of dynamic elements, or dynamic sorts) there is a special predicate defining a labelled transition relation; thus the dynamic elements are modelled by means of labelled transition systems. An abstract dynamic data type (addt) is an isomorphism class of dynamic algebras.

To obtain specifications for addt's in some papers (e.g., [2, 3]) we have used first-order logic; however such logic is not suitable for expressing all interesting abstract properties of dynamic elements. Extensions of first-order logic, particularly suitable to such purpose, have been proposed in other papers; in [6] by adding branching-time temporal logic combinators for expressing abstract properties on the activity of dynamic elements and in [10, 4] by adding special combinators for expressing abstract properties on the concurrent/distributed structure of the dynamic elements (e.g. no multilevel parallelism).

In this paper we present an "event logic" for specifying addt's; the name "event" comes from "event structures" (see, e.g., [13]). Event structures are a nice formalism for abstractly modelling dynamic elements, which allows to express properties such as causality and true concurrency; but they lack well-developed techniques for structuring detailed descriptions of complex systems. Event logic is an attempt at building up an integrated specification framework for:

– expressing abstract properties of dynamic elements by giving some "relationships" between events;
– modelling dynamic elements in a simple and intuitive way as labelled transition systems.

The basic idea is that, given a labelled transition system, an "event" is just a set of partial execution paths (those corresponding to occurrences of that event). Thus, for example, a causality relationship e' `causes` e'' holds on a labelled transition system L iff for all execution paths σ of L, if $\sigma = \sigma_1 \sigma' \sigma_2$, where σ' is a partial path

⋆ This work has been supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of C.N.R. (Italy), COMPASS-Esprit-BRA-W.G. n 3264 and by a grant ENEL/CRA (Milano Italy).

corresponding to an occurrence of e', then $\sigma_2 = \sigma_3\sigma''\sigma_4$, where σ'' is a partial path corresponding to an occurrence of e''.

As a consequence, we have that in this framework events are not considered instantaneous and so we can also express finer relationships between them, as:

- there is no instant at which e' and e'' occur simultaneously,
- e is the sequential composition of e' and e'',
- each occurrence of e' includes an occurrence of e'' (i.e., e'' starts either together or after e' and terminates either before or together e').

It is important to note that the use of non-instantaneous events for specifying labelled transition systems allows us to give very abstract specifications without fixing the atomic grain of the activity of the dynamic elements.

Moreover, in this algebraic framework events are not just a set of names; they are instead elements of particular sorts of a dynamic data type (one sort of events for each sort of dynamic elements) and so it is possible to define operations and predicates having arguments and/or results of event sort. As a consequence:

- we have events of different sort and thus we can relate, for example, the events of compound dynamic elements with the events of their dynamic subcomponents;
- complex events may be described by composing several simpler events, and that helps in writing modular and readable specifications.

Event logic includes as a sublanguage the temporal logic of [6] (the only difference is that here we consider also the past); in some sense the formalism of [6] considers only "instantaneous" events (i.e., events whose occurrences are just paths consisting either of one state or of one transition); while here we still use the same temporal combinators for expressing temporal relationships between non-instantaneous events.

In the literature there are other attempts to combine events and logic (see, e.g., [8]); the main difference with our work is that they use logic to specify classical event structures; while here we consider as models labelled transition systems equipped with non-instantaneous events.

However there are still some points about our event logic, which are worth to be further investigated.

First of all we need a rigorous definition of a kind of *non-classical event structures*, different from those of [13], where events are non-instantaneous and the relationships between events are not simply causality and conflict. Then, we are interested to study the relationships between non-classical and classical event structures. For example, can the first be "simulated" by the latter ?

Moreover it seems that we can associate with each labelled transition system equipped with non-instantaneous events a non-classical event structure and so a specification given using our event logic may have models at two different levels:

- a kind of non-classical event structures;
- labelled transition systems equipped with non-instantaneous events.

Then, we can define an "implementation relationship" between a non-classical event structure and a labelled transition system equipped with non-instantaneous events s.t. given an event logic specification SP:

for all event structures ES, if ES is a model of SP and it is implemented by L, then L is a model of SP;
for all labelled transition systems L, if L is a model of SP, there exists an event structure ES which is a model of SP and is implemented by L.

In this paper we present the event logic; in Sect. 1 we shortly introduce concrete and abstract dynamic data types and show how to integrate the notion of non-instantaneous event; the logic is formally defined in Sect. 2; finally in Sect. 3 we report some simple illustrative examples. It is worth noticing that the resulting specification formalism is an institution (see [5]). For lack of room we cannot report neither the proof that the event logic is an institution, nor a sound deductive system nor more interesting examples of specifications; these parts are reported in the full paper [11].

1 Abstract Dynamic Data Types and their Specification

1.1 Dynamic Algebras and Specifications

First we briefly report the essential definitions about concrete and abstract dynamic data types and their specification (see, e.g., [3, 6]); and then show how events may be introduced in this framework.

Dynamic Algebras. A dynamic algebra (concrete dynamic data type) is just a many-sorted algebra with predicates (see [7]) such that for some of the sorts (sorts of dynamic elements, or dynamic sorts) there is a special predicate defining a labelled transition relation; thus the dynamic elements are modelled by means of labelled transition systems.

Definition 1.

- A *dynamic signature* $D\Sigma$ is a couple (Σ, DS) where:
  - $\Sigma = (S, OP, PR)$ is a predicate signature,
  - $DS \subseteq S$ (the elements in DS are the *dynamic sorts*),
  - for all $ds \in DS$ there exists a sort $lab(ds) \in S - DS$ (labels of the transitions of the elements of sort ds) and a predicate $\_ \xrightarrow{\_} \_ : ds \times lab(ds) \times ds \in PR$ (the transition relation of the dynamic elements of sort ds).
- A *dynamic algebra* on $D\Sigma$ (shortly $D\Sigma$-algebra) is just a Σ-algebra. □

In this paper, for some of the operation and predicate symbols, we use a mixfix notation. For instance, $\rightarrow : ds \times lab(ds) \times ds$ means that we shall write $t \xrightarrow{t'} t''$ instead of $\rightarrow (t, t', t'')$; i.e. terms of appropriate sorts replace underscores.

If DA is a $D\Sigma$-algebra and $ds \in DS$, then the carrier of sort ds, DA_{ds}, the carrier of sort $lab(ds)$, $DA_{lab(ds)}$, and the interpretation of the predicate $\rightarrow$, $\rightarrow^{DA}$, are respectively the states, the labels and the transition relation of a labelled transition system describing the activity of the dynamic elements of sort ds.

Dynamic Specifications. Following a widely accepted idea (see e.g. [14]) a (static) *abstract data type* (shortly *adt*) is an isomorphism class of Σ-algebras (note that [14] considers only term-generated algebras). A *(simple) specification* is a couple $SP = (\Sigma, AX)$, where Σ is a signature and AX a set of first-order formulae on Σ (the *axioms* of SP); the *models* of SP are the Σ-algebras satisfying the axioms in AX.

In the *initial algebra approach* SP defines the adt consisting of the (isomorphism class of the) initial elements, if any, of the class $Mod(SP)$. In the *loose*approach, instead, SP is viewed as a description of the main properties of an adt; thus it represents a class, consisting of all the adt's satisfying the properties expressed by the axioms (more formally: the class of all isomorphism classes included in $Mod(SP)$).

The above definitions can be easily adapted to the dynamic case: an *abstract dynamic data type* (shortly *addt*) is an isomorphism class of dynamic algebras and a *(simple) dynamic specification* is a couple $DSP = (D\Sigma, AX)$. In this case the set of axioms AX must express both usual "static" properties of the data and properties about the activity of the dynamic elements; the problem is, now, to find an appropriate logic for expressing such properties.

In some previous papers (see e.g., [3]) we have used first-order logic; since dynamic signatures include explicitly a predicate corresponding to the transition relation, we can formalize several properties about the activity of the dynamic elements using first-order logic. For example $\nexists l, s \,.\, t \xrightarrow{l} s \wedge P(s)$ requires that the dynamic element represented by the term t cannot pass into a state s s.t. $P(s)$ holds. However, these are only "local" properties on the activity of the dynamic elements (about the immediate future and past). In general we want to express "global properties" as liveness and safety properties.

Given a dynamic algebra DA and a dynamic sort ds, a global view of the activity of a dynamic element $d \in DA_{ds}$ may be represented by the set of all its *execution paths*, i.e. maximal sequences of labelled transitions of the form:

$$\ldots d_{-2} \xrightarrow{l_{-2}}{}^{DA} d_{-1} \xrightarrow{l_{-1}}{}^{DA} d \xrightarrow{l_0}{}^{DA} d_1 \xrightarrow{l_1}{}^{DA} d_2 \ldots$$

(clearly such sequences may be either bounded or unbounded both at the left and at the right); a sequence as above represents a possible behaviour of d; the partial execution path $\ldots d_{-2} \xrightarrow{l_{-2}}{}^{DA} d_{-1} \xrightarrow{l_{-1}}{}^{DA} d$ represents the past and $d \xrightarrow{l_0}{}^{DA} d_1 \xrightarrow{l_1}{}^{DA} d_2 \ldots$ the future of such behaviour. A graphic representation of the activity of a dynamic element is reported in Fig. 1. Thus a property on the activity of d may be formalized as a property of the execution paths for d.

The temporal logic for specifications of addt's of [6] already allows to do that; however it permits only to express properties about the "points" of the paths (i.e. on the states and on the labels of the paths). For example, the formula $\triangle(t, \Diamond[\lambda s.P(s)])$ requires that each execution path of the dynamic element represented by the term t contains a state s s.t. $P(s)$; while $\nabla(t, \Box{<}\lambda l.Q(l){>})$ requires that the dynamic element represented by the term t has at least one execution path with labels l s.t. $Q(l)$. But in that formalism it is rather hard to require that after receiving in input a value eventually the system will output some other value, when inputting

Fig. 1. The activity of a dynamic element d

and outputting a value consists of several transitions (e.g., because each transition corresponds to move one bit).

1.2 Event Algebras

Paths and Events. Given a $D\Sigma$-dynamic algebra DA and a dynamic sort ds of $D\Sigma$, PAR_PATH(DA, ds) denotes the set of the *partial execution paths* for the dynamic elements of sort ds, i.e. it is the set of all sequences having either form either (1), ... or (4) below:

(1)	$\dots\ d_{-2}\ l_{-2}\ d_{-1}\ l_{-1}\ d_0\ l_0\ d_1\ l_1\ d_2\ l_2\ \dots$		(unbounded path),
(2)	$d_0\ l_0\ d_1\ l_1\ d_2\ l_2\ \dots$		(right-unbounded path),
(3)	$\dots\ d_{-2}\ l_{-2}\ d_{-1}\ l_{-1}\ d_0$		(left-unbounded path),
(4)	$d_0\ l_0\ d_1\ l_1\ d_2\ l_2\ \dots\ d_n$	$n \geq 0$	(bounded path),

where for all integer i $d_i \in DA_{ds}, l_i \in DA_{lab(ds)}$ and $(d_i, l_i, d_{i+1}) \in \rightarrow^{DA}$.

A (partial) *composition operation* is defined on the elements of PAR_PATH(DA, ds):
$\_ \cdot \_$: PAR_PATH(DA, ds) $\times$ PAR_PATH(DA, ds) $\rightarrow$ PAR_PATH(DA, ds)
$\sigma \cdot \sigma' =_{def}$ if $\sigma = \ldots\ d_n\ l_n\ d_{n+1}$ and $\sigma' = d_{n+1}\ l_{n+1}\ \ldots$ then $\ldots\ d_n\ l_n\ d_{n+1}\ l_{n+1}\ \ldots$
else undefined.
PATH(DA, ds) denotes the set of the *execution paths* for elements of sort ds, i.e.:
$\{\sigma \mid \sigma \in$ PAR_PATH(DA, ds) and there does not exist
$\sigma' \in$ PAR_PATH(DA, ds) s.t. $\sigma' \neq \sigma$ and σ is a subpath of $\sigma'\}$
where σ is a *subpath* of σ' iff there exist σ_1, σ_2 s.t. either $\sigma' = \sigma_1 \cdot \sigma \cdot \sigma_2$ or $\sigma' = \sigma_1 \cdot \sigma$ or $\sigma' = \sigma \cdot \sigma_2$.

Given $d \in DA_{ds}$, an *execution path for* d is a triple $<\sigma_p, d, \sigma_f>$ s.t. $\sigma_p \cdot d \cdot \sigma_f \in$ PATH(DA, ds).

In brief, we recall that a classical event structure of instantaneous events (see [13]) consists of a set of events E plus a partial order $\geq$ on E (causality relation) and a binary relation $\neq$ on E (conflict relation); see e.g. Fig. 2.

Fig. 2. Two classical event structures ES_1 and ES_2

Now we have to connect events and labelled transition systems; the basic idea is that an event e corresponds to the set of its occurrences i.e. to the set of the partial execution paths corresponding to its occurrences; and that a relation R among e_1, $\ldots$, e_n corresponds to a relation R' among the occurrences of $e_1, \ldots, e_n$. For example in Fig. 1 we have outlined some partial execution paths corresponding to occurrences of the three events E_1, E_2, E_3; notice that the relationships described by ES_2 are satisfied, while those of ES_1 are not. But, note that here the relationship between E_2 and E_3 is finer than the one formalized by ES_2: E_2 terminates with E_3 (i.e., each occurrence of E_3 terminates with an occurrence of E_2).

Event Signatures and Algebras. Event algebras are particular dynamic algebras where with each dynamic sort a special event sort is associated, whose elements are events, i.e. sets of event occurrences, i.e. sets of partial execution paths.

Definition 2.

- An *event signature* $E\Sigma$ is a dynamic signature (Σ, DS) where for all $ds \in DS$ there exists a sort $event(ds) \in S - DS$.
- An *event algebra* EA on $E\Sigma$ ($E\Sigma$-algebra) is an $E\Sigma$-dynamic algebra s.t. for all $ds \in DS$ $EA_{event(ds)} = \mathcal{P}$(PAR_PATH($EA, ds$)) [2].

[2] If A is a set, then $\mathcal{P}(A)$ denotes the set of the parts of A.

– Given two $E\Sigma$-algebras EA and EB, an *$E\Sigma$-event morphism* p from EA into EB (written $p: EA \to EB$) is a Σ-morphism of total algebras with predicates s.t. for all $ds \in DS$, $e \in EA_{event(ds)}$ $p(e) \supseteq$
$$\{\ldots\ p(d_{i-1})\ p(l_{i-1})\ p(d_i)\ p(l_i)\ p(d_{i+1})\ p(l_{i+1})\ \ldots \mid \ldots d_{i-1}\ l_{i-1}\ d_i\ l_i\ d_{i+1} l_{i+1}\ \ldots \in e\}.\square$$

For a dynamic sort $ds \in DS$ each element of sort *event*(ds) represents an event suitable for the dynamic elements of sort ds; such an event consists of the set of all partial execution paths corresponding to its occurrences; every such path will be called an *occurrence* of the event.

Event morphisms are particular morphisms of algebras with predicates; thus they preserve the truth of predicates (see [7]) and so also the activity of the dynamic elements: if $d \xrightarrow{l} d'$ in EA, then $p(d) \xrightarrow{p(l)} p(d')$ in EB. For this reason they also preserve the partial paths, i.e. if an event e has an occurrence σ in EA, then the image of σ by p is a partial path in EB.

Event algebras and morphisms on a signature $E\Sigma$ constitute a category, denoted by $EAlg(E\Sigma)$.

2 Event Logic

2.1 Syntax and Informal Semantics

Here we introduce a minimal set of combinators for the event logic; several other interesting derived combinators are reported in Appendix A.

In the following let $E\Sigma = (\Sigma, DS)$, with $\Sigma = (S, OP, PR)$, be an event signature and $\mathcal{X}$ an infinite set of variable symbols. We recall that a *sort assignment* for $E\Sigma$ is a partial function $X: \mathcal{X} \to S$ and it will be seen as an S-indexed family $\{X_s\}_{s \in S}$, where $X_s = \{x \mid x \in \mathcal{X} \text{ and } X(x) = s\}$.

Definition 3. The *terms*, the *path formulae* and the *event logic formulae* on $E\Sigma$ and a sort assignment X for $E\Sigma$ are defined as follows. We denote respectively by $\{T_{E\Sigma}(X)_s\}_{s \in S}$ (the family of the sets of terms), $\{PF_{E\Sigma}(X)_{ds}\}_{ds \in DS}$ (the family of sets of path formulae) and $EF_{E\Sigma}(X)$ (the set of the event logic formulae). For all $s \in S$, $ds \in DS$

• *terms*

– $X_s \subseteq T_{E\Sigma}(X)_s$
– $Op(t_1, \ldots, t_n) \in T_{E\Sigma}(X)_s$
  for all $Op: s_1 \times \ldots \times s_n \to s \in OP$, $t_i \in T_{E\Sigma}(X)_{s_i}$ $i = 1, \ldots, n$
– $e_1; e_2 \in T_{E\Sigma}(X)_{event(ds)}$ for all $e_1, e_2 \in T_{E\Sigma}(X)_{event(ds)}$
– e_1 `and` $e_2 \in T_{E\Sigma}(X)_{event(ds)}$ for all $e_1, e_2 \in T_{E\Sigma}(X)_{event(ds)}$
– `not` $e, e^+, e^\omega \in T_{E\Sigma}(X)_{event(ds)}$ for all $e \in T_{E\Sigma}(X)_{event(ds)}$
– $[\lambda x.\phi] \in T_{E\Sigma}(X)_{event(ds)}$ for all $x \in X_{ds}$, $\phi \in EF_{E\Sigma}(X)$
– $<\lambda x.\phi> \in T_{E\Sigma}(X)_{event(ds)}$ for all $x \in X_{lab(ds)}$, $\phi \in EF_{E\Sigma}(X)$

- *path formulae*

– $\triangleright e, e\triangleleft \in PF_{E\Sigma}(X)_{ds}$ for all $e \in T_{E\Sigma}(X)_{event(ds)}$
– π_1 `until` π_2, π_1 `since` $\pi_2 \in PF_{E\Sigma}(X)_{ds}$ for all $\pi_1, \pi_2 \in PF_{E\Sigma}(X)_{ds}$
– $\neg\pi_1, \pi_1 \supset \pi_2, \forall x.\pi_1 \in PF_{E\Sigma}(X)_{ds}$ for all $\pi \in PF_{E\Sigma}(X)_{ds}, x \in X$

- *formulae*

– $Pr(t_1, \ldots, t_n) \in EF_{E\Sigma}(X)$
for all $Pr\colon s_1 \times \ldots \times s_n \in PR,\ t_i \in T_{E\Sigma}(X)_{s_i}\ \ i = 1, \ldots, n$
– $t_1 = t_2 \in EF_{E\Sigma}(X)$ for all $t_1, t_2 \in T_{E\Sigma}(X)_s$
– $\neg\phi_1, \phi_1 \supset \phi_2, \forall x.\phi_1 \in EF_{E\Sigma}(X)$ for all $\phi_1, \phi_2 \in EF_{E\Sigma}(X), x \in X$
– $e_1 \Rightarrow e_2 \in EF_{E\Sigma}(X)$ for all $e_1, e_2 \in T_{E\Sigma}(X)_{event(ds)}$
– $\triangle(t, \pi) \in EF_{E\Sigma}(X)$ for all $t \in T_{E\Sigma}(X)_{ds}, \pi \in PF_{E\Sigma}(X)_{ds}$ □

The semantics of the various combinators is informally described below and defined formally in Def. 4. To do that we need the following definitions on paths. Given $\sigma, \sigma' \in$ PAR_PATH(DA, ds):

– if σ is right-bounded, $Last(\sigma)$ denotes the *last element* of σ; analogously if σ is left-bounded, $First(\sigma)$ denotes the *first element* of σ;
– σ is a *prefix* (*postfix*) of σ' iff either $\sigma = \sigma'$ or there exists σ'' s.t. $\sigma' = \sigma \cdot \sigma''$ ($\sigma' = \sigma'' \cdot \sigma$);
– σ is a *proper* prefix (postfix) of σ' iff σ is a prefix (postfix) of σ' and $\sigma \neq \sigma'$;
– if σ' is a prefix (postfix) of σ, then $\sigma - \sigma'$ denotes the path σ'' s.t. $\sigma' \cdot \sigma'' = \sigma$ ($\sigma'' \cdot \sigma' = \sigma$), if it exists. Notice that if σ is left (right) bounded, then $\sigma - \sigma = First(\sigma)$ ($\sigma - \sigma = Last(\sigma)$) since σ is both a prefix and a postfix of itself.

The special operators, which represent events, are described by giving the occurrences of the represented events.

" `;` " sequential composition of two events: an occurrence of $e_1; e_2$ is a partial path consisting of the composition of an occurrence of e_1 and of an occurrence of e_2.
"`and`" conjunction of events: an occurrence of e_1 `and` e_2 is a partial path which is an occurrence of e_1 and of e_2.
"`not`" negation of an event: an occurrence of `not` e is a partial path which is not an occurrence of e.
" $^+$" finite repetition of an event: an occurrence of e^+ is a finite (non-null) sequence of occurrences of e.
" $^\omega$" infinite repetition of an event: an occurrence of e^ω is a countably infinite sequence of occurrences of e.
"[. . .]" conditional state event: an occurrence of $[\lambda x.\phi]$ is a partial path consisting of one state satisfying the condition expressed by the formula ϕ.
"$< \ldots >$" conditional atomic event: an occurrence of $<\lambda x.\phi>$ is a partial path consisting of one transition labelled with a label satisfying ϕ.

Path formulae express properties of the execution paths for some dynamic element; thus, given an execution path for a dynamic element d, say $\delta = <\sigma_p, d, \sigma_f>$, we have that:

"$\rhd$" occurrence of an event in the future: $\rhd e$ holds on δ iff there exists σ occurrence of e which is a prefix of σ_f;

"$\lhd$" occurrence of an event in the past: $e\lhd$ holds on δ iff there exists σ occurrence of e which is a postfix of σ_p;

`until` is defined analogously to the homonymous temporal logic combinator;

π_1 `until` π_2 holds on δ iff there exists σ postfix of σ_f s.t.

- π_2 holds on $<\sigma_p \cdot (\sigma_f - \sigma), First(\sigma), \sigma>$,
- π_1 holds in each point between d and the beginning of σ, i.e. for all proper prefixes σ' of $\sigma_f - \sigma$, π_1 holds on $<\sigma_p \cdot \sigma', Last(\sigma'), \sigma_f - \sigma'>$ (notice that σ' is also a prefix of σ_f so we can consider $\sigma_f - \sigma'$).

The above condition is graphically represented in Fig. 3.

Fig. 3. The until operator

π_1 `since` π_2 is the analogous of `until` for the past.

$\pi_1 \supset \pi_2$ holds on δ iff either π_1 does not hold on δ or π_2 holds on δ; analogously $\neg$ and $\forall$ are defined as the analogous first order combinators.

$\triangle(t, \pi)$ holds iff each execution path for the element represented by the term t satisfies the condition expressed by the path formula π.

$e_1 \Rightarrow e_2$ holds iff each occurrence of e_1 is also an occurrence of e_2.

2.2 The Institution of Event Logic

In this section we state (without proof which is in [11]) that the logical formalism defined in the previous section constitutes an institution (see [5])

$$\mathcal{EV} = (\mathbf{ESig}, \mathrm{ESen}, \mathrm{EMod}, \models)$$

whose components are defined below.

In [5] the concept of institution has been advocated as a rigorous presentation of a logical formalism; being an institution means that a a formalism has an intrinsic coherence and of course various results are at hand for building specifications, specification languages, transferring results from an institution to another (see, e.g. [12, 5, 1, 9]).

In the following $\mathcal{P} = (\mathbf{PSig}, \mathrm{PSen}, \mathrm{PMod}, \models^P)$ denotes the institution of the total (many-sorted) algebras with predicates and first-order formulae (see [5]).

ESig is the category defined as follows.

- The objects are the event signatures (see Def. 2);
- The morphisms: given $E\Sigma_1 = (\Sigma_1, DS_1)$ and $E\Sigma_2 = (\Sigma_2, DS_2)$, $\rho \in \mathbf{ESig}(E\Sigma_1, E\Sigma_2)$ iff $\rho \in \mathbf{PSig}(\Sigma_1, \Sigma_2)$ and for all $ds \in DS_1$
  - $\rho(ds) \in DS_2$,
  - $\rho(lab(ds)) = lab(\rho(ds))$,
  - $\rho(event(ds)) = event(\rho(ds))$,
  - $\rho(- \overset{-}{\longrightarrow} -: ds \times lab(ds) \times ds) = - \overset{-}{\longrightarrow} -: \rho(ds) \times lab(\rho(ds)) \times \rho(ds)$.
- The identities: given $E\Sigma = (\Sigma, DS)$ $I_{E\Sigma} = I_\Sigma$ (identity of Σ in $\mathcal{P}$).
- The morphism composition: given $\rho_1: E\Sigma_1 \to E\Sigma_2$, $\rho_2: E\Sigma_2 \to E\Sigma_3$ $\rho_1 \cdot \rho_2 = \rho_1 \cdot_{\mathcal{P}} \rho_2$, where $\cdot_{\mathcal{P}}$ is the morphism composition of **PSig**.

ESen: $\mathbf{ESig} \to \mathbf{Set}$ is the functor defined as follows.

- On objects ($EF_{E\Sigma}(X)$ has been given in Def. 3): $\text{ESen}(E\Sigma) = \{(X, \phi) \mid X$ is a sort assignment for $E\Sigma$ and $\phi \in EF_{E\Sigma}(X)\}$.
- On morphisms: $\text{ESen}(\rho: E\Sigma_1 \to E\Sigma_2): \text{ESen}(E\Sigma_1) \to \text{ESen}(E\Sigma_2)$ is the functor associating with ϕ the formula obtained by replacing in ϕ each symbol Sym of $E\Sigma_1$ with $\rho(Sym)$.

EMod: $\mathbf{ESig} \to \mathbf{Cat}^{\mathbf{OP}}$ is the functor defined as follows.

- On objects: $\text{EMod}(E\Sigma) = EAlg(E\Sigma)$ (the category $EAlg(E\Sigma)$ has been defined in Sect. 1.2).
- On morphisms: given $\rho: E\Sigma_1 \to E\Sigma_2$ $\text{EMod}(\rho)$ is the restriction of $\text{PMod}(\rho)$ to $\text{EMod}(E\Sigma_1)$.

Validity relation: it is given in the following definition.

We recall that given an $E\Sigma$-algebra EA, a *valuation of the variables* in X into EA is a family of total functions $V = \{V_s\}_{s \in S}$ s.t. for all $s \in S$ $V_s: X_s \to EA_s$.

Definition 4. Let EA be an $E\Sigma$-algebra and V a valuation of X into EA; then we define by multiple induction:

- the interpretation of a term t in EA under V ($t^{EA,V}$),
- when a path formula π holds on δ (an execution path for a dynamic element) in EA under V (written $\delta, EA, V \models \pi$),
- when a formula ϕ holds in EA under V (written $EA, V \models \phi$)

in the following way (where we assume that each formula is well-formed).

• *Interpretation of terms*

- $x^{EA,V} = V(x)$
- $Op(t_1, \ldots, t_n)^{EA,V} = Op^{EA}({t_1}^{EA,V}, \ldots, {t_n}^{EA,V})$
- $(e_1; e_2)^{EA,V} = \{\sigma_1 \cdot \sigma_2 \mid \sigma_1 \in {e_1}^{EA,V}, \sigma_2 \in {e_2}^{EA,V}$ and $\sigma_1 \cdot \sigma_2$ is defined$\}$
- $(e_1 \texttt{ and } e_2)^{EA,V} = e_1^{EA,V} \cap e_2^{EA,V}$
- $(\texttt{not}\, e)^{EA,V} = \text{PAR\_PATH}(EA, ds) - e^{EA,V}$
- $(e^+)^{EA,V} =$
  $\{\sigma_1 \cdot \ldots \cdot \sigma_n \mid n \geq 1,$ for $i = 1, \ldots, n$ $\sigma_i \in e^{EA,V}$ and $\sigma_1 \cdot \ldots \cdot \sigma_n$ is defined$\}$
- $(e^\omega)^{EA,V} = \{\sigma_1 \cdot \ldots \cdot \sigma_n \cdot \ldots \mid$ for $i \geq 1$ $\sigma_i \in e^{EA,V}$ and $\sigma_1 \cdot \ldots \cdot \sigma_n \cdot \ldots$ is defined$\}$
- $[\lambda x.\phi]^{EA,V} = \{d \mid d \in EA_{ds}$ and $EA, V[d/x] \models \phi\}$

- $<\lambda x.\phi>^{EA,V} =$
  $$\{d\ l\ d' \mid d, d' \in EA_{ds}, l \in EA_{lab(ds)}, (d,l,d') \in \rightarrow^{EA} \text{ and } EA, V[l/x] \models \phi\}$$

- *Validity of path formulae* In the following we assume $\delta = <\sigma_p, d, \sigma_f>$.

  - $\delta, EA, V \models\ \triangleright e$ iff there exists $\sigma \in e^{EA,V}$ s.t. σ is a prefix of σ_f
  - $\delta, EA, V \models e\triangleleft$ iff there exists $\sigma \in e^{EA,V}$ s.t. σ is a postfix of σ_p
  - $\delta, EA, V \models \pi_1\ \mathtt{until}\ \pi_2$ iff there exists σ postfix of σ_f s.t.
    - ⋆ $<\sigma_p \cdot (\sigma_f - \sigma), First(\sigma), \sigma>, EA, V \models \pi_2$
    - ⋆ for all proper prefixes σ' of $\sigma_f - \sigma$ $<\sigma_p \cdot \sigma', Last(\sigma'), \sigma_f - \sigma'>, EA, V \models \pi_1$
  - $\delta, EA, V \models \pi_1\ \mathtt{since}\ \pi_2$ iff there exists σ prefix of σ_p s.t.
    - ⋆ $<\sigma, Last(\sigma), (\sigma_p - \sigma) \cdot \sigma_f>, EA, V \models \pi_2$
    - ⋆ for all proper postfixes σ' of $\sigma_p - \sigma$ $<\sigma_p - \sigma', First(\sigma'), \sigma' \cdot \sigma_f>, EA, V \models \pi_1$
  - $\delta, EA, V \models \neg\pi$ iff $\delta, EA, V \not\models \pi$
  - $\delta, EA, V \models \pi_1 \supset \pi_2$ iff either $\delta, EA, V \not\models \pi_1$ or $\delta, EA, V \models \pi_2$
  - $\delta, EA, V \models \forall x.\pi$ iff for all $v \in EA_s$ $\delta, EA, V[v/x] \models \pi$

- *Validity of event logic formulae*

  - $EA, V \models Pr(t_1, \ldots, t_n)$ iff $<{t_1}^{EA,V}, \ldots, {t_n}^{EA,V}> \in Pr^{EA}$
  - $EA, V \models t_1 = t_2$ iff ${t_1}^{EA,V} = {t_2}^{EA,V}$
  - $EA, V \models \neg\phi$ iff $EA, V \not\models \phi$
  - $EA, V \models \phi_1 \supset \phi_2$ iff either $EA, V \not\models \phi_1$ or $EA, V \models \phi_2$
  - $EA, V \models \forall x.\phi$ iff for all $v \in EA_s$ $EA, V[v/x] \models \phi$
  - $EA, V \models e_1 \Rightarrow e_2$ iff ${e_1}^{EA,V} \subseteq {e_2}^{EA,V}$
  - $EA, V \models \triangle(t, \pi)$ iff for all execution paths δ for $t^{EA,V}$ $\delta, EA, V \models \pi$.

$\phi \in EF_{E\Sigma}(X)$ is *valid* in EA ($EA \models \phi$) iff $EA, V \models \phi$ for all valuations V. □

Proposition 5. *$\mathcal{EV}$ is an institution.* □

3 Examples

Here we give two simple examples illustrating the main features of our specification formalism; in [11] more interesting specifications of well-known examples in the field of concurrency are given (e.g., alternating bit protocol, serializable data base).

In this section for keeping the specifications simpler and more readable, we

- use the derived combinators of the event logic given in the appendix A;
- write **sorts** S **dsorts** DS **opns** OP **preds** PR for the event signature (Σ, DS), where Σ is:
  $(S \cup DS \cup \{lab(ds), event(ds) \mid ds \in DS\}, OP,$
  $PR \cup \{- \overset{-}{\longrightarrow} -: ds \times lab(ds) \times ds \mid ds \in DS\})$,
  i.e. we leave the canonical sorts and predicates implicit;
- use the following well-known constructs for structuring specifications: ...+... (sum of specifications), **enrich**...**by** ...and ...[.../...] (renaming of either sort or operation or predicate symbols), see e.g., [14].

3.1 Specification of a Sequential Nondeterministic Program with Interactive I/O.

In this paragraph we abstractly specify a nondeterministic sequential program with interactive I/O, i.e. where I/O is realized by dynamically interacting with an external environment (e.g., a terminal).

We operationally model the program by means of a labelled transition system, with labels correspond to input and output actions. Specifying such a program means to require some relationship between the receiving of some input and the returning of some output: precisely some relationship among the "values" which constitute the inputs and the outputs and "temporal"/"causal" relationships among the transitions corresponding to inputting and outputting.

Static Properties. First we abstractly qualify what are the inputs and the outputs of the program by giving two classical (static) specifications. For example, if

spec *INPUT* =
  sorts input
  opns
    $- \& -: input \times input \to input$
  preds
    Atomic: *input*
  axioms
    $i\&(i'\&i'') = (i\&i')\&i''$

spec *OUTPUT* =*INPUT*[*output*/*input*]

we only require that there are "atomic" inputs (not further specified) and an associative composition operation; analogously for outputs.

Using algebraic specifications we can also describe which are the correct outputs for the various inputs (i.e., what the program calculates).

spec *CORRECTNESS* =
  enrich *INPUT*+ *OUTPUT* **by**
  preds
    Corr: *input* × *output*
  axioms
    ...properties about *Corr*...

Notice that we use a predicate *Corr* and not an operation from inputs to outputs, since we consider nondeterministic programs.

Dynamic Properties. We consider two kind of events:

- *input events* (the inputting of some value) and
- *output events* (the outputting of some value).

spec *PROG* =
  enrich *CORRECTNESS* **by**

dsorts prog
opns
  $Init: \to prog$
  $Input: input \to lab(prog)$
  $Output: output \to lab(prog)$
  $IN: input \to event(prog)$
  $OUT: output \to event(prog)$
axioms
  $\nexists p, l . p \xrightarrow{l} Init$
– $Init$ is truly the initial state
  . . . axioms expressing dynamic properties . . .

Here we do not give just one specification $PROG$, but instead show how to express various sample dynamic properties of the program.

Properties About Events. Recall that events are particular data of our specification, so we can express properties about them "in isolation".

– Input events are finite. $IN(i)$ `is finite`
– Inputting a compound value is the same that inputting the component values.

$$IN(i_1 \& i_2) = IN(i_1); IN(i_2)$$

– The input event of an atomic value consists in performing one transition labelled in a particular way.

$$Atomic(i) \supset IN(i) = <Input(i)>$$

For the output events we can give either similar or different properties.

Relationships Between Events.

– During the activity of the program at most one input event may happen.

$$i \neq i' \supset \triangle(p, \triangleright IN(i) \supset \neg \blacklozenge \triangleright IN(i')) \qquad \triangle\,(p, \triangleright IN(i) \supset \neg\Diamond IN(i))$$

If at a certain moment the inputting of i starts, then in the future (including now) i' s.t. $i' \neq i$ will never be input and in the future (excluding now) i will never be input.
Notice that the axiom $\triangle(Init, \triangleright IN(i) \supset \neg \blacklozenge \triangleright IN(i'))$ does not formalize this property; indeed it would prevent any occurrence of the the input events.
– The so called "partial correctness" with respect to the requirement expressed by $Corr$, i.e., whatever output is the result of a correct elaboration of some input (output events are caused by appropriate input events).

$$\triangle(p, \triangleright OUT(o) \supset (\exists i . Corr(i, o) \wedge \blacklozenge \mathtt{P} \triangleright IN(i)))$$

If at a certain moment the output of o starts, then there exists an appropriate i s.t. the input of i started sometimes in the past.
In Fig. 4 we report a graphical representation of the activity of two programs with occurrences of input and output events satisfying and not the above axiom.

Fig. 4. Programs respecting and not the partial correctness requirement

- The so called "total correctness" with respect to the requirement expressed by $Corr$, i.e. the reception of some input will result in the outputting of a correct output (input events cause appropriate output events).

  $$\triangle(p, \, \triangleright IN(i) \supset (\exists o.Corr(i,o) \wedge \blacklozenge OUT(o)))$$

  If at a certain moment the input of i starts, then there exists an appropriate o s.t. the output of o eventually will start. The first case of Fig. 4 does not satisfy the above axiom, while the second does.

Notice that these axioms do not require that the input event ends before the output event; so the specification has models corresponding to programs which terminate by inputting a special value.

- Input and output events are mutually exclusive.

  $$IN(i) \; \texttt{disjoint with} \; OUT(o)$$

  Without this axiom the specification has models where, for example, while part of the output is printed, part of the input is received (e.g., programs which go on receiving one value, processing it and then printing the result) see second case of Fig. 4.

3.2 A buffer

Here we specify an unbounded buffer with the following properties: it

- preserves the order of the received messages;
- is safe (it cannot corrupt any message);
- cannot loose nor duplicate any message (where "cannot loose" is not to be intended that it will deliver each received message, but only that if it delivers a message m, then it has already delivered all messages received before m);

– interacts synchronously with the users (it delivers a message only iff there is a user ready to accept it);
– cannot receive and/or deliver simultaneously several messages.

We have no information about the nature of the messages, thus they are described by the (static) specification:

$$\textbf{spec}\ \mathit{MESSAGE} = \textbf{sorts}\ \mathit{message}$$

The interesting events, which may happen during the activity of a buffer, are receiving and delivering a message. Since the buffer interacts with the users in a synchronous way, such events correspond to perform transitions labelled by $Rec(m)$ and $Del(m)$ respectively. Notice that in this way we have that receiving and delivering cannot occur simultaneously. For expressing the buffer properties we use the complex events corresponding to receive and deliver in an orderly way a sequence of messages (defined by the operations REC_LIST and DEL_LIST).

SEQ is the parametric specification of finite sequences with the operations $\Lambda\colon \to seq(elem)$ (empty sequence) and $-\&-\colon seq(elem) \times elem \to seq(elem)$ (adding an element at the end of a sequence).

spec BUFFER =
 enrich SEQ($\mathit{MESSAGE}$) **by**
 dsorts *buffer*
 opns
  $Init\colon \to buffer$
 – the initial state of the buffer
  $Rec, Del\colon message \to lab(buffer)$
  $NOT\_REC, NOT\_DEL\colon message \to event(buffer)$
  $REC\_LIST, DEL\_LIST\colon seq(message) \to event(buffer)$
 – auxiliary event operations
 axioms
  $\nexists c, l. c \xrightarrow{l} Init$
 – $Init$ is truly the initial state
  $NOT\_REC = < \nexists m. l = Rec(m) >^{\star}$
  $REC\_LIST(\Lambda) = NOT\_REC$
  $REC\_LIST(sm\&m) = REC\_LIST(sm); NOT\_REC; < Rec(m) >$
  $NOT\_DEL = < \nexists m. l = Del(m) >^{\star}$
  $DEL\_LIST(\Lambda) = NOT\_DEL$
  $DEL\_LIST(sm\&m) = DEL\_LIST(sm); NOT\_DEL; < Del(m) >$
 – definition of the auxiliary event operations
  $[Init]; DEL\_LIST(sm) \Rightarrow [Init]; REC\_LIST(sm); \texttt{all}$
 – a partial path starting from $Init$ corresponding to deliver a sequence of messages always has a prefix corresponding to receive the same sequence of values this axiom correspond to require all the buffer properties (ordered, safety, no lost messages, no duplications)
  $\triangle(c, \blacklozenge \triangleright [\exists x'. x \xrightarrow{Rec(m)} x'])$
 – responsiveness (the buffer cannot refuse forever to receive a message)

Notice that the last axiom requires that eventually the buffer will have the capability of receiving the message m and that is strongly different from $\triangle(c, \blacklozenge \triangleright <Rec(m)>)$ requiring that eventually the buffer will receive the message m.

4 Conclusions

By looking at the examples of Sect. 3 we can see some of the advantages of the use of event logic with respect to first-order/temporal logic.

In this framework we can specify dynamic elements modelled as labelled transition systems without fixing the atomicity grain of the system transitions. In the example of Sect. 3.1 we do not fix the atomicity grain of the transitions of the system representing the program. Thus we can have a specification $PROG$ having models where non-internal transitions correspond to inputting/outputting respectively files, atomic values and bits. However by the same formalism we can give also "less abstract" specifications, by qualifying the system transitions (for example in Sect. 3.2 where receiving and delivering a message is an atomic transition).

Since in our algebraic formalism events are particular data, *we can first describe complex events and then we can express relationships between such events.* For example, in Sect. 3.2, the "being caused and ending before" relation between the two events "delivering a sequence of messages" and "receiving the same sequence" describes the main properties of the buffer. In Sect. 3.1 we can simply express properties as partial/total correctness also in the case of programs performing I/O bit after bit (just add axioms qualifying the events IN and OUT); while using only instantaneous events (corresponding to perform one transition) such properties are very hard to express.

On the other hand, our event logic considers non-instantaneous events and so there are also some advantages with respect to classical event structures; for example in our framework *we can express relationship between events finer than causality and conflict*. In Sect. 3.1 the causality property between input and output events (total correctness) does not mean that the second event must occur after the first one is terminated. Thus, we can give a specification $PROG$ having very reasonable models, where inputs events terminate after the caused output events. Think, for example, of programs which go on receiving a value, processing it and then returning the result, until they receive a special value, signalling the end of the activity.

A Derived Event Operations and Logical Combinators

Whenever x is the only variable occurring free in ϕ, $[\lambda x.\phi]$ is shortened in $[\phi]$; moreover $[x = t]$ is further shortened in $[t]$. Analogous abbreviations are defined for $<\ldots>$.

– Nondeterministic choice between two events: an occurrence of e_1 `or` e_2 is either an occurrence of e_1 or an occurrence of e_2.

$$e_1 \;\texttt{or}\; e_2 =_{\mathrm{def}} \;\texttt{not}((\texttt{not}\, e_1) \;\texttt{and}\; (\texttt{not}\, e_2))$$

– Difference between two events: an occurrence of $e_1 - e_2$ is an occurrence of e_1 which is not an occurrence of e_2.

$$e_1 - e_2 =_{\text{def}} \; e_1 \texttt{ and not } e_2$$

– The universal instantaneous event: an occurrence of `inst` is a partial path consisting of one state.

$$\texttt{inst} =_{\text{def}} [\,\texttt{true}\,]$$

– Either null or finite repetition of an event: an occurrence of $e^\star$ is a finite or null sequence of occurrences of e.

$$e^\star =_{\text{def}} \; \texttt{inst or } e^+$$

– The universal atomic event: an occurrence of `step` is a partial path consisting of one transition.

$$\texttt{step} =_{\text{def}} \; < \texttt{true} >$$

– The finite universal event: an occurrence of `steps` is a finite partial path.

$$\texttt{steps} =_{\text{def}} \texttt{step}^\star$$

– The universal event: an occurrence of `all` is a partial path

$$\texttt{all} =_{\text{def}} \texttt{step}^\omega \texttt{ or step}^\star$$

– Test of (weak) disjunction between two events: e_1 and e_2 are (weakly) disjoint iff there is no transition where both events are occurring (`not all` is the impossible event, i.e. the one without occurrences).

e_1 `disjoint with` $e_2 \Leftrightarrow_{\text{def}}$

$[((e_1; \texttt{steps}) \texttt{ and } (\texttt{steps}; e_2)) - (e_1; \texttt{step}^+; e_2)]$ `or`
$[e_1 \texttt{ and } (\texttt{steps}; e_2; \texttt{steps})]$ `or`
$[((e_2; \texttt{steps}) \texttt{ and } (\texttt{steps}; e_1)) - (e_2; \texttt{step}^+; e_1)]$ `or`
$[e_2 \texttt{ and } (\texttt{steps}; e_1; \texttt{steps})] = \quad \texttt{not all}$

– Check of finiteness/infiniteness of an event: e `is finite` (e `is infinite`) iff all occurrences of e are finite (infinite).

$$e \texttt{ is finite } =_{\text{def}} \; e \Rightarrow \texttt{steps} \qquad e \texttt{ is infinite } =_{\text{def}} \; e \Rightarrow \texttt{step}^\omega$$

Temporal Logical Combinators

– Eventually and always in the future including the present.

$$\blacklozenge\pi =_{\text{def}} \; \texttt{true until } \pi \qquad \blacksquare\,\pi =_{\text{def}} \; \neg\blacklozenge\neg\pi$$

– Sometimes and always in the past including the present.

$$\blacklozenge\mathsf{P}\pi =_{\text{def}} \; \texttt{true since } \pi \qquad \blacksquare\,\mathsf{P}\pi =_{\text{def}} \; \neg\blacklozenge\mathsf{P}\neg\pi$$

– Eventually and always in the future excluding the present.

$$\Diamond\pi =_{\text{def}} \; \blacklozenge\pi \wedge \neg\pi \qquad \Box\,\pi =_{\text{def}} \; \neg\Diamond\neg\pi$$

– Sometimes and always in the past excluding the present.

$$\Diamond\mathsf{P}\pi =_{\text{def}} \; \blacklozenge\mathsf{P}\pi \wedge \neg\pi \qquad \Box\mathsf{P}\pi =_{\text{def}} \; \neg\Diamond\mathsf{P}\neg\pi$$

References

1. E. Astesiano and M. Cerioli. Relationships between logical frameworks. In the same volume, 1992.
2. E. Astesiano and G. Reggio. SMoLCS-driven concurrent calculi. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proc. TAPSOFT'87, Vol. 1*, number 249 in Lecture Notes in Computer Science, pages 169–201, Berlin, 1987. Springer Verlag.
3. E. Astesiano and G. Reggio. A structural approach to the formal modelization and specification of concurrent systems. Technical Report 0, Formal Methods Group, Dipartimento di Matematica, Università di Genova, Italy, 1991.
4. E. Astesiano and G. Reggio. Entity institutions: Frameworks for dynamic systems. in preparation, 1992.
5. R.M. Burstall and J.A. Goguen. Introducing institutions. In E. Clarke and D. Kozen, editors, *Logics of Programming Workshop*, number 164 in Lecture Notes in Computer Science, pages 221–255, Berlin, 1984. Springer Verlag.
6. G. Costa and G. Reggio. Abstract dynamic data types: a temporal logic approach. In A. Tarlecki, editor, *Proc. MFCS'91*, number 520 in Lecture Notes in Computer Science, pages 103–112, Berlin, 1991. Springer Verlag.
7. J. Gouguen and J. Meseguer. Models and equality for logic programming. In *Proc. TAPSOFT'87, Vol. 2*, number 250 in Lecture Notes in Computer Science, pages 1–22, Berlin, 1987. Springer Verlag.
8. K. Lodaya and P. S. Thiagarajan. A modal logic for a subclass of event structures. In T. Ottmann, editor, *Proceeding of ICALP'87*, number 267 in Lecture Notes in Computer Science, pages 290–303, Berlin, 1987. Springer Verlag.
9. J. Meseguer. General logic. In *Logic Colluqium'87*, Amsterdam, 1989. North-Holland.
10. G. Reggio. Entities: an istitution for dynamic systems. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, number 534 in Lecture Notes in Computer Science, pages 244–265, Berlin, 1991. Springer Verlag.
11. G. Reggio. Event logic for specifying abstract dynamic data types – Extended version. Technical Report 13, Formal Methods Group, Dipartimento di Matematica, Università di Genova, Italy, 1991.
12. D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76, 1988.
13. G. Winskel. An introduction to event structures. In J.W. de Bakker, W.-P. de Roever, and G. Rozemberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 354 in Lecture Notes in Computer Science, pages 364–397, Berlin, 1989. Springer Verlag.
14. M. Wirsing. Algebraic specifications. In van Leeuwen Jan, editor, *Handbook of Theoret. Comput. Sci.*, volume B, pages 675–788. Elsevier, 1990.

This article was processed using the LaTeX macro package with LLNCS style