Developing Formal Semantics of `epkml`

Alexander Knapp and Piotr Kosiuczenko
Ludwig–Maximilians–Universität München
{knapp,kosiucze}@pst.informatik.uni-muenchen.de

9th May 1997

Abstract

`epkml` is a window oriented language for the design of electronic product catalogues. It is an instance of the Standard Generalized Markup Language (SGML). In this paper we enhance the formal operational semantics of `epkml` to be able to deal with concurrent processes. The semantics allows limited concurrency which is modelled by interleaving. It allows to spawn concurrently multiple processes and to control their behaviour in time. To do it, we have extended `epkml` by some auxiliary expressions. A proper `epkml` expression, if needed, is translated to such an auxiliary expression and formal semantics of such an expresion is given. States, in this semantics, comprise the necessary information to control the execution flow of an `epkml` program. Transitions between states are given by rules that show how an `epkml` statement affects the state in which it is executed. We build in constructs for specification of soft real-time requirements allowing to specify time bounds of process execution and to specify periodical processes.

Introduction

`epkml` is a language for the design of electronic products catalogues. It is an instance of the Standard Generalized Markup Language (SGML, cf. [Gol94])), based on the Hypertext Markup Language (HTML, cf. [Gra95]). The reader is referred to [KKM96] for details.

In this paper we enhance the formal operational semantics [Plo91] of `epkml` proposed in [KKM96]. The semantics provides a static binding for the variables and the procedures (i.e. the layout elements), a block structure for the scopes of variables, and "call-by-value" parameter handling. It takes care of possible interactions by the user and of the time aspects the language offers. States comprise the necessary information to control the execution flow of an `epkml` program, but as in the previous version, we abstract from the exact layout and database internals. The possible transitions between states are given by rules that show how an `epkml` statement affects the state in which it is executed. In contrary to the previous semantics, we allow limited concurrency, but as in the previous version, we specify only the control flow of a program. An `epkml` program is executed sequentially except of time-line. This construct allows to spawn concurrently multiple processes and to control their behaviour in time. Concurrency is modelled by interleaving (i.e. allowing nondeterministic execution of those steps which can be interleaved). We build in constructs for specification of soft real-time requirements allowing to specify time bounds of processes (or more generally time-line) execution and to specify periodical processes. To achieve it, we have extended `epkml` by some auxiliary expressions. A proper `epkml` expression, if needed, is translated to such an auxiliary expression and then an operational semantics is given. Time constraints are expressed using an auxiliary expression:

$$\text{do } \{\ s\ \} \text{ watching}(c, t)$$

where s is a statement, c is a condition, and t is a time value. It allows to execute statement s as long as condition c holds. The time value indicated when the execution has been started and allows for ignoring older interrupts (i.e. those interrupts, which has been raised before time t). Like in SDL [IT94] we assume, that the current system time can be accessed by a *now* operator. We use also a

$$\text{repeat } \{\ s\ \}$$

expression, which allows to repeat execution of s arbitrary many times.
The paper is organized as follows. Section 1 presents formal background and the basic definitions of our semantics. Section 2 presents basic principles of the semantics concerning sequential control flow. The principles are illustrates with few examples. In Section 3, we present semantics of the so-called time-lines and slices which allow for concurrent control flow. We conclude with some remarks on the proposed semantics.

1 Formal Background

The set of all possible fully expanded `epkml` statements (the programs) is called S. Any statement that does not contain free variables is a value of our semantics.
Some statements contain actions (e.g. `<on-click>` in `<button>`); we define a function

$$\text{act} : S \rightharpoonup S$$

that returns the lexicographically first action for a given statement, whenever this is possible. In general, we define functions with names as used in the syntax of `epkml` to yield that part of a statement that their syntactical counterpart surrounds, say

$$\text{template} : S \to S$$

returns for a statement s the text contained in the first occurring pairs of `<template>` and `</template>` in s, according to the block structure in the lexicographic ordering. The set of all `epkml` variable names is called N.
We assume one global time. This global time can only be accessed by the predefined operation *now*, which is supposed to return the current time value. We assume that there is an unbounded universum of time variables contained in N representing timers.

1.1 The State Model

The semantics works on states, elements of a set Σ that are six-tuples consisting of a theme structure graph (from a set Γ), an environment (from E), a database instance (from Δ), the laid out elements (from Λ), a memory store (from M), and set of signals (from Π):

$$\Sigma = \Gamma \times E \times \Delta \times \Lambda \times M \times \Pi.$$

A memory store out of M is a set of locations C, where a pair of an `epkml` statement and an environment can be saved.

$$M = C \rightharpoonup S \times E$$

An operation

$$\text{new} : M \to C$$

returns a new location for a given memory state, not in the domain of this memory state. The operations

$$\text{stm} : C \to S \qquad \text{and} \qquad \text{env} : C \to E$$

return for a location the statement and the environment contained, respectively.
An environment is a list (a stack) of partial mappings from the variable names N to locations,

$$E = (N \rightharpoonup C)^*$$

written as $\varepsilon = [e_1, \ldots, e_n|\varepsilon']$ (where $e_1, \ldots, e_n$ are partial mappings and ε' another environment). The application of such an environment (seen as a partial function) to a variable name n, written $\varepsilon(n)$, yields the value of the first (the uppermost) applicable mapping in the list for this variable. An environment may be extended by a function

$$+ : E \times N \times C \to E$$

It adds another pair of name and location to the first partial mapping of the environment; the name must not be in the domain of this first mapping. We write this operation as $e + \{n \mapsto l\}$. A value of an already defined variable may be changed by the function

$$\nleftarrow\!\!+ : E \times N \times C \to E.$$

For a given (ε, n, l) with $\varepsilon = [e_1, \ldots, e_k]$ it searches the first component e_i in ε such that $n \in \operatorname{dom} e_i$ and sets the value of n to l. We write $\varepsilon \nleftarrow\!\!+ \{n \mapsto l\}$ for this operation. The set of theme structure graphs Γ comprises all lists of trees that are describable in `epkml`, the vertices of which are marked with a name, an `epkml` statement (the contents of `<theme>` without the sub-themes), an environment, and a boolean value to mark a theme as the current or not. This set Γ is a subset of

$$\Gamma_0 = (N \times (S \times E) \times \mathbf{B} + (N \times (S \times E) \times \mathbf{B}) \times \Gamma_0)^*$$

Here, a (more general) theme structure graph is a list of either single nodes or father nodes with a theme structure graph as their son. In the second case it looks as follows:

$$[(n_1, (s_1, \varepsilon_1), b_1, \gamma_1), \ldots, (n_k, (s_k, \varepsilon_k), b_k, \gamma_k)] \ .$$

In the sequel, we will only use those graphs in Γ_0 in which all names are distinct and where at most one theme is marked as current; the set of such graphs will be called Γ. On these graphs, the function

$$\text{pos} : \Gamma \times N \to \Gamma$$

returns for a given graph and a name a new graph, with the boolean mark set only for the theme with the given name. The function

$$\text{nxt} : \Gamma \to N$$

returns the name of the vertex with the next theme, it is the right sibling of the node marked as current (or the current node itself if it has no right sibling.) The partial function

$$\text{bld} : \Gamma \times N \times (S \times E) \times \Gamma^* \rightharpoonup \Gamma$$

constructs for a given graph, node information, and a list of graphs, a new one that extends the first graph by a tree out of the node information as the father node and the list of graphs as its son nodes. For example, we have

$$\begin{aligned}&\mathrm{bld}((n_1,(s_1,\varepsilon),\top),n,(s,\varepsilon),[(n_2,(s_2,\varepsilon),\bot),(n_3,(s_3,\varepsilon),\bot)]) = \\ &\quad [(n_1,(s_1,\varepsilon),\top),(n,(s,\varepsilon),\bot,[(n_2,(s_2,\varepsilon),\bot),(n_3,(s_3,\varepsilon),\bot)])]\end{aligned}$$

As for variable names, the function

$$\mathrm{env} : \Gamma \to E$$

returns for a theme structure graph the environment of the node that is marked as current.
The set of database instances Δ contains lists of tuples according to the relevant database schemes. We do not model these schemes. We assume the required SQL statements as given.
The elements of the layout Λ are sets of locations the contents of which is supposed to be visible on the screen.

$$L = \wp(C)$$

Interaction and synchronization is modeled using the set Π which contains all available signals which are supposed to trigger transitions. Every signal has a time stamp and a name of an `epkml` element it was arised for. We set

$$\begin{aligned}\Pi = \bigcup_{t,n}(\wp(\{\ &\mathit{clicked}(t,n), \mathit{end}(t,n), \\ &\mathit{play}(t,n), \mathit{stop}(t,n), \mathit{pause}(t,n), \mathit{rewind}(t,n), \mathit{forward}(t,n)\}))\end{aligned}$$

These signals inform the system that a certain action has taken place for a certain layout element at a certain time (cf. Section 3.1).

1.2 The Rules

We only present rules for so-called normalized `epkml` programs. These are syntactically correct statements in which every attribute assignment `status=open` is substituted by `status=closed` and an `<open>` statement, after the owning statement. For example, we replace

```
<frame name=f>
  f
</frame>
```

by

```
<frame name=f
   status=closed>
  f
</frame>
<open name=f>
```

Obviously, this is no restriction of the possible `epkml` programs.
To take into account side effects of the interactive elements, we think of their variable changes as being permanent (the memory of their location is changed, too); we only partially state the necessary memory manipulations.
Every rule transforms a pair of an `epkml` statement and a state either merely to a state, saying that the statement has been fully executed, for example

$$\langle s, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow (\gamma', \varepsilon', \delta', \lambda', \mu', \pi') \ ,$$

or again to a pair of a statement and a state, for example

$$\langle s, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow \langle s', (\gamma', \varepsilon', \delta', \lambda', \mu', \pi')\rangle \ .$$

We introduce an empty statement $\emptyset$ not part of the `epkml` syntax which is treated as a syntactical neutral element.

1.3 Auxiliary Statements

We extend the `epkml` language (and therefore the set S) by some auxiliary elements. These elements will be used to model interaction and concurrency. First,

$$\text{do } \{\ p\ \} \text{ watching}(c, t)$$

means, that the `epkml` statement p will be executed as long as condition c holds. The time-value t indicates when the execution of this statement was started.

$$\frac{\langle p, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow \langle p', (\gamma', \varepsilon', \delta', \lambda', \mu', \pi')\rangle}{\begin{array}{l}\langle \text{do } \{\ p\ \} \text{ watching}(c, t), (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow \\ \quad \langle \text{do } \{\ p'\ \} \text{ watching}(c, t), (\gamma', \varepsilon', \delta', \lambda', \mu', \pi')\rangle \\ \quad \text{if } c \text{ holds in } (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\end{array}}$$

If c does not hold in a state, then this command is aborted:

$$\begin{array}{l}\langle \text{do } \{\ p\ \} \text{ watching}(c, t), (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow (\gamma, \varepsilon, \delta, \lambda, \mu, \pi) \\ \quad \text{if } c \text{ does not hold in } (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\end{array}$$

If no statements are to be executed, the command aborts, too:

$$\langle \text{do } \{\ \emptyset\ \} \text{ watching}(c, t), (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)$$

The second auxiliary construct is

$$\text{repeat } \{\ p\ \}$$

which allows to execute statement p an arbitrary number of times:

$$\langle \text{repeat } \{\ p\ \}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow \langle p \ \text{ repeat } \{\ p\ \}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle$$

To break the loop one needs to use sentences of the form: do { repeat { p } } watching(c, t), because otherwise process p would be executed forever. Again, if no statements are to be repeated, the loop aborts:

$$\langle \text{repeat } \{\ \emptyset\ \}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)$$

In order to describe the idle process, we add the statement skip with semantics

$$\langle \text{skip}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)$$

We also introduce two auxiliary expressions push and pop that open and close a new binding scope. The push expression carries the new environment of the scope, the pop expression carries the old environment to be restored. Note that we lift semantical information to the syntax.

$$\langle \text{push}(\varepsilon'), (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow (\gamma, [\emptyset|\varepsilon'], \delta, \lambda, \mu, \pi)$$
$$\langle \text{pop}(\varepsilon'), (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow (\gamma, \varepsilon', \delta, \lambda, \mu, \pi)$$

2 Sequential `epkml`

The sequential composition of statements is reflected by the usual structural rule.

$$\frac{\langle s_1, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow \langle s_1', (\gamma', \varepsilon', \delta', \lambda', \mu', \pi')\rangle}{\langle s_1\ s_2, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow \langle s_1'\ s_2, (\gamma', \varepsilon', \delta', \lambda', \mu', \pi')\rangle}$$

2.1 Variables

The variables are dealt with conventionally: A variable name declaration adds a new location to the environment, and stores the appropriate contents into this location.

$$\begin{aligned}&\langle \texttt{<var name=}n\texttt{ value=}v\texttt{> </var>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow \\ &\qquad (\gamma, \varepsilon + \{n \mapsto \text{new}(\mu)\}, \delta, \lambda, \mu + \{\text{new}(\mu) \mapsto (v, \varepsilon)\}, \pi)\\ &\langle \texttt{<var name=}n\texttt{>}\ s\ \texttt{</var>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow \\ &\qquad (\gamma, \varepsilon + \{n \mapsto \text{new}(\mu)\}, \delta, \lambda, \mu + \{\text{new}(\mu) \mapsto (s, \varepsilon)\}, \pi)\end{aligned}$$

Similarily, a declared variable can be manipulated by `<set>`;

$$\langle \texttt{<set name=}n\ \texttt{value=}v\texttt{> </set>}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow$$
$$(\gamma,\varepsilon,\delta,\lambda,\mu \nleftrightarrow \{\varepsilon(n) \mapsto (v,\varepsilon)\},\pi)$$
$$\langle \texttt{<set name=}n\texttt{>}\ s\ \texttt{</set>}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow$$
$$(\gamma,\varepsilon,\delta,\lambda,\mu \nleftrightarrow \{\varepsilon(n) \mapsto (s,\varepsilon)\},\pi)$$

In the same way attributes of tags may be changed (which also affects the main memory).
The contents of a variable may be treated as a statement with its own environment:

$$\langle \texttt{\$}n\texttt{\$}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow$$
$$\langle \mathrm{push}(\mathrm{env}(\varepsilon(n)))\ \ \mathrm{stm}(\varepsilon(n))\ \ \mathrm{pop}(\varepsilon), (\gamma, [\mathrm{env}(\varepsilon(n))|\varepsilon],\delta,\lambda,\mu,\pi)\rangle$$

2.2 Layout

Closed layout elements (i.e. `status=closed`) are treated much the same as variables. Here *layout* may have the values `window`, `frame`, `button` and so on.

$$\langle \texttt{<}\mathit{layout}\ \texttt{name=}n\ \texttt{status=closed>}\ s\ \texttt{</}\mathit{layout}\texttt{>}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow$$
$$(\gamma,\varepsilon + \{n \mapsto \mathrm{new}(\mu)\},\delta,\lambda,$$
$$\mu + \{\mathrm{new}(\mu) \mapsto (\texttt{<}\mathit{layout}\ \texttt{name=}n\ \texttt{status=closed>}\ s\ \texttt{</}\mathit{layout}\texttt{>},\varepsilon)\},\pi)$$

2.3 Database

The semantics of the database tag relies on an appropriate semantics of the corresponding SQL statements. A semantics of such a statement s, relative to the semantics of a database instance δ, is denoted by $[\![s]\!]_{\mathrm{SQL}}(\delta)$. Below, we specify only how it influences control flow.

$$\langle \texttt{<sql result=}r\texttt{> SELECT}\ s\ \texttt{</sql>}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow$$
$$(\gamma,\varepsilon,\delta,\lambda,\mu + \{r \mapsto [\![\texttt{SELECT}\ s]\!]_{\mathrm{sql}}(\delta)\},\pi)$$
$$\langle \texttt{<sql> INSERT}\ s\ \texttt{</sql>}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow (\gamma,\varepsilon,[\![\texttt{INSERT}\ s]\!]_{\mathrm{sql}}(\delta),\lambda,\mu,\pi)$$
$$\langle \texttt{<sql> DELETE}\ s\ \texttt{</sql>}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow (\gamma,\varepsilon,[\![\texttt{DELETE}\ s]\!]_{\mathrm{sql}}(\delta),\lambda,\mu,\pi)$$

2.4 Themes

The theme structure graph is built up from bottom to top taking into account the scope of the theme names.

$$\frac{\langle(\texttt{<theme name=}n_i\texttt{>}\ s_i\ \texttt{</theme>}, (\gamma, \varepsilon, \delta, \lambda, \mu_{i-1})\rangle \Rightarrow (\gamma_i, \varepsilon_i, \delta, \lambda, \mu_i, \pi))_{1\le i\le k}}{\begin{array}{l}\langle\texttt{<theme name=}n\texttt{>}\ s\\ \quad(\texttt{<theme name=}n_i\texttt{>}\ s_i\ \texttt{</theme>})_{1\le i\le k}\\ \ \texttt{</theme>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow\\ \qquad(\mathrm{bld}(\gamma, n, (s, \theta), (\gamma_i)_{1\le i\le k}), \varepsilon + \{n \mapsto \mathrm{new}(\mu_k)\}, \delta, \lambda,\\ \qquad\ \mu + \{\mathrm{new}(\mu_k) \mapsto (s, \varepsilon)\}, \pi)\end{array}},$$

where θ is the partial function mapping all theme names of the theme structure graph to their corresponding locations stacked upon ε, and $\mu_0 = \mu$. Note that every theme structure subgraph carries its own relevant environment information such that the environment changes in the antecedens of the rule need not be taken into account.

Example. The `epkml` statement

```
<theme name=father>
  $f$
  <theme name=son-1>
    $s_1$
  </theme>
  <theme name=son-2>
    $s_2$
  </theme>
</theme>
```

yields the following theme structure graph in the empty environment by using the bld function three times:

$$[(\texttt{father}, (f, \theta), \bot), [(\texttt{son-1}, (s_1, \theta), \bot), (\texttt{son-2}, (s_2, \theta), \bot)]]$$

where $\theta = \{\texttt{father} \mapsto l_1, \texttt{son-1} \mapsto l_2, \texttt{son-2} \mapsto l_3\}$ for locations $l_1 = \mathrm{new}(\emptyset)$, $l_2 = \mathrm{new}(\{\texttt{father} \mapsto l_1\}$, and $l_3 = \mathrm{new}(\{\texttt{father} \mapsto l_1, \texttt{son-1} \mapsto l_2\}$ from C.

Note that we used big step semantics here, since `<theme>` statements do not contribute to execution but provide environmental information.

2.5 Control

The conditionals are dealt with conventionally:

$$\langle \texttt{<empty which=}n\texttt{>}\ s\ \texttt{</empty>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$$
$$\quad \langle s, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle, \qquad \text{if } \varepsilon(n) = \emptyset$$
$$\langle \texttt{<empty which=}n\texttt{>}\ s\ \texttt{</empty>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$$
$$\quad (\gamma, \varepsilon, \delta, \lambda, \mu, \pi), \qquad \text{if } \varepsilon(n) \neq \emptyset$$

The `<non-empty>` statement is handled dually.

$$\langle \texttt{<non-empty which=}n\texttt{>}\ s\ \texttt{</non-empty>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$$
$$\quad \langle s, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle, \qquad \text{if } \varepsilon(n) \neq \emptyset$$
$$\langle \texttt{<non-empty which=}n\texttt{>}\ s\ \texttt{</non-empty>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$$
$$\quad (\gamma, \varepsilon, \delta, \lambda, \mu, \pi), \qquad \text{if } \varepsilon(n) = \emptyset$$

The semantics of the `<open>` statement must discern between themes and non-themes. In order to open non-themes, the attributes of the `<open>` statement have to be evaluated first.

$$\langle \texttt{<attribute name=}n\ \texttt{value=}v\texttt{>}\ \texttt{</attribute>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$$
$$\quad (\gamma, \varepsilon \nleftarrow\!\!+ \{n \mapsto \text{new}(\mu)\}, \delta, \lambda, \mu + \{\text{new}(\mu) \mapsto (v, \varepsilon)\}, \pi)$$
$$\langle \texttt{<element name=}n\texttt{>}\ s\ \texttt{</element>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$$
$$\quad (\gamma, \varepsilon \nleftarrow\!\!+ \{n \mapsto \text{new}(\mu)\}, \delta, \lambda, \mu + \{\text{new}(\mu) \mapsto (s, \varepsilon)\}, \pi)$$

The exectution of an `<open>`–statement for a layout element recursively performs an `<open>` for all layout elements that are contained inside. This recursion is treated syntactically by adding such `<open>` statements in the same way as normalized `epkml` programs were introduced; we denote the corresponding manipulation function by open : $S \to S$.

$$\frac{\begin{array}{l}(\langle \texttt{<attribute name=}a_i\ \texttt{value=}v_i\texttt{>}\ \texttt{</attribute>}, (\gamma, \varepsilon, \delta, \lambda, \mu_{i-1}, \pi)\rangle \Longrightarrow \\ \quad (\gamma, \varepsilon_i, \delta, \lambda, \mu_i, \pi))_{1 \leq i \leq m} \\ (\langle \texttt{<element name=}e_i\texttt{>}\ s_i\ \texttt{</element>}, (\gamma, \varepsilon, \delta, \lambda, \mu_{m+i-1}, \pi)\rangle \Longrightarrow \\ \quad (\gamma, \varepsilon_{m+i}, \delta, \lambda, \mu_{m+i}, \pi))_{1 \leq i \leq n}\end{array}}{\begin{array}{l}\langle \texttt{<open name=}n\texttt{>} \\ \quad \texttt{<attribute name=}a_i\ \texttt{value=}v_i\texttt{>}_{1 \leq i \leq m} \\ \quad \texttt{<element name=}e_i\texttt{>}\ s_i\ \texttt{</element>}_{1 \leq i \leq n} \\ \ \texttt{</open>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow \\ \qquad \langle \text{push}([\{(a_i \mapsto \varepsilon_i(a_i))_{1 \leq i \leq m}, (e_i \mapsto \varepsilon_{m+i-1}(e_i))_{1 \leq i \leq n}\} \mid \text{env}(\varepsilon(n))]) \\ \qquad\quad s \\ \qquad\ \ \text{pop}(\varepsilon), (\gamma', \varepsilon, \delta', \lambda \cup \{\varepsilon(n)\}, \mu_{m+n} + \{\varepsilon(n)\texttt{.status} \mapsto \texttt{opened}\}, \pi)\rangle\end{array}}$$

if n is not a theme and where $\mathrm{stm}(\varepsilon(n)) = \texttt{<}\mathit{layout}\texttt{>}\ s'\ \texttt{</}\mathit{layout}\texttt{>}$, and $s = \mathrm{open}(s')$.

Example. The `epkml` statement

```
<frame name=f
  attribs="a"
  status=closed>
  f
</frame>
<open name=f>
  <attribute name=a value=15>
</open>
```

yields the following transitions and environment changes starting from an empty environment ε_0, an empty layout, an arbitrary database δ, an empty memory μ_0, and arbitrary signals π: First the frame is stored in the environment, that is

$$\varepsilon_1 = [\{\texttt{f} \mapsto l_1\}]$$
$$\mu_1 = \{l_1 \mapsto (\texttt{<frame name=f attribs="a" status=closed>}\ f\ \texttt{</frame>}, \emptyset)\}$$

for a new location $l_1 \in C$. Next the attribute inside the `<open>` statement is evaluated leading to

$$\varepsilon_2 = [\{\texttt{a} \mapsto l_2, \texttt{f} \mapsto l_1\}]$$
$$\mu_2 = \mu_1 + \{l_2 \mapsto (\texttt{15}, \varepsilon_1)\}$$

for a new location $l_2 \in C$. Consequently, the statements $\mathrm{open}(f)$ of the frame have to be evaluated in

$$(\gamma, [\emptyset|[\{\texttt{a} \mapsto l_2\}|\varepsilon_1]], \delta, \{l_1\}, \mu_2 + \{\varepsilon(n)\texttt{.status} \mapsto \texttt{opened}\}, \pi),$$

the layout is updated, and the status of `f` is set to `opened`.

Closing a layout element removes its layout representation.

$$\langle \texttt{<close name=}n\texttt{> </close>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow (\gamma, \varepsilon, \delta, \lambda \setminus \{\varepsilon(n)\}, \mu, \pi)$$

For themes, `<open>` has to be treated specially:

$$\begin{aligned}&\langle \texttt{<open name=}n\texttt{> </open>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow \\ &\qquad \langle \mathrm{presenentation}(\mathrm{stm}(\varepsilon(n))), \\ &\qquad\ (\mathrm{pos}(\gamma, n), \mathrm{env}(n), \delta, \lambda, \mu + \{\varepsilon(n)\texttt{.status} \mapsto \texttt{opened}\}, \pi)\rangle \quad \text{if } n \text{ is a theme}\end{aligned}$$

The navigational commands `<next>`, `<previous>`, etc. make use of the theme graph.

$$\langle \texttt{<next theme> </next>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$$
$$\langle \mathrm{stm}(\varepsilon(\mathrm{nxt}(\gamma))), (\mathrm{pos}(\gamma, \mathrm{nxt}(\gamma)), \mathrm{env}(\varepsilon(\mathrm{nxt}(\gamma))), \delta, \lambda, \mu, \pi)\rangle$$

The other navigation commands are evaluated alike.

2.6 Interaction

The user may interact with the program via buttons, the keyboard, etc. The program awaits signals from interactions with the `<wait>` tag. It can wait indefinitly long or until certain conditions are fulfilled. Indefinite waiting is treated as follows:

$$\langle \texttt{<wait> </wait>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$$
$$\langle \texttt{<var name=}S\texttt{ value=}\mathit{now}\texttt{> </var>}$$
$$\text{do } \{ \text{ repeat } \{ \text{ skip } \} \} \text{ watching}(\top, S), (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle$$
$$\text{for a new time variable } S$$

If the statment `<wait end-of=`t`>` is to be executed, then a new time variable S is chosen and set with value *now*. The system is supposed to examine whether $S + t < \mathit{now}$.

$$\langle \texttt{<wait end-of=}t\texttt{> </wait>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$$
$$\langle \texttt{<var name=}S\texttt{ value=}\mathit{now}\texttt{> </var>}$$
$$\text{do } \{ \text{ repeat } \{ \text{ skip } \} \} \text{watching}(\mathit{now} < S + t, S), (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle,$$
$$\text{for a new time variable } S$$

If a `<wait>` statement has its attribute `end-of` set to a name, execution is delayed until the *end* signal for this name occurs in the global signal set. All interrupts which occured before execution of `<wait>` has been started, will be ignored. It is expressed by the condition $\mathit{end}(t, n) \in \pi \to t < S, S)$.

$$\langle \texttt{<wait end=of=}n\texttt{>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$$
$$\langle \texttt{<var name=}S\texttt{ value=}\mathit{now}\texttt{> </var>}$$
$$\text{do } \{ \text{ repeat } \{ \text{ skip } \} \} \text{ watching}(\mathit{end}(t, n) \in \pi \to t < S, S),$$
$$(\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle, \qquad \text{for a new time variable } S$$

When the user interacts with a layout element this layout element's action statement is executed in its own environment. Here the permanence of changes of the memory space (not those of the environment) is important. The system is supposed to react on the earliest actual signal.

$$\begin{array}{l}
\langle \text{do } \{\ s\ \}\ \text{ watching}(c,S), (\gamma,\varepsilon,\delta,\lambda,\mu,\pi \uplus \{\mathit{clicked}(t,n)\})\rangle \Longrightarrow \\
\quad \langle \text{push}(\text{env}(\varepsilon(n)))\ \ \text{act}(\text{stm}(\varepsilon(n)))\ \ \text{pop}(\varepsilon) \\
\quad\ \ \texttt{<set name=}S\ \texttt{value=}\mathit{now}\texttt{> </var>} \\
\quad\ \ \text{do } \{ \text{ repeat } \{ \text{ skip } \} \ \}\ \text{ watching}(c,S), (\gamma,\varepsilon,\gamma,\delta,\lambda,\mu)\rangle \\
\qquad\ \text{if } t \text{ is minimal such that } S < t \text{ and } \mathit{clicked}(t,n) \in \pi
\end{array}$$

Note that only signals are taken into account that were send after we started waiting. Also note that after the interaction starting time for the new waiting loop is updated.

3 Concurrency in `epkml`

In general, `epkml` allows only limited concurrency. An execution of an `epkml` program is sequential, only processes spawned by `<time-line>` or `<video>`, `<audio>` and `<slide-show>` can be executed concurrently. These concurrent processes may be synchronized via time constraints and interrupting signals.
We model concurrency by interleaving and nondeterminism. A process will simply be an `epkml` statement. Our model of communication is asynchronous.

3.1 Time-Lines

Time-lines (`<time-line>`) organize execution into periods or slices (`<slice>`). Slices are the biggest units allowing to spawn multiple processes executing concurrently. The `<par>` tags inside a `<slice>` allow for execution of processes contained in their scopes in parallel. Consecutive slices are executed one after another but not concurrently.
A time-line can be bounded in duration by the optional attribute `end-of`. It can be executed only once or periodically. In the second case the attribute `periodical` is set. We translate these different time-lines to auxiliary expressions as follows.
If a time-line is scheduled to be executed till the end of process n we get the rule:

$$\begin{array}{l}
\langle \texttt{<time-line end-of=}n\texttt{>}\ s\ \texttt{</time-line>}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow \\
\quad \langle \texttt{<var name=}S\ \texttt{value=}\mathit{now}\texttt{> </var>} \\
\quad\ \ \text{do } \{\ s\ \text{ repeat } \{ \text{ skip } \} \ \}\ \text{ watching}(\mathit{end}(t,n) \in \pi \rightarrow t < S, S), \\
\qquad (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle, \qquad \text{for a new time variable } S
\end{array}$$

We assume that time-lines will be executed until their breaking condition C holds. They cannot be aborted after their statements have been finished. This is specified by repeat { skip }.

If in addition the time-line is periodical then we get rule:

$\langle$`<time-line end-of=`n` periodical>` s `</time-line>`$, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$
  $\langle$`<var name=`S`> </var>`
  repeat {
  `<set name=`S` value=`now`> </var>`
  do { s repeat { skip } } watching$(end(t, n) \in \pi \to t < S, S)$ },
  $(\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle$, for a new time variable S

Time-line execution can also be restricted by explicitly giving a time-bound:

$\langle$`<time-line end-of=`t`>` s `</time-line>`$, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$
  $\langle$`<var name=`S` value=`now`> </var>`
  do { s repeat { skip } } watching$(now < S + t, S), (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle$
  for a new time variable S

Similarily for the case that a time-line is additionally periodical:

$\langle$`<time-line end-of=`t` periodical>` s `</time-line>`$, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$
  $\langle$`</var> <var name=`S`> </var>`
  repeat {
  `<set name=`S` value=`now`> </var>`
  do { s repeat { skip } } watching$(now < S + t, S)$ }$, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle$
  for a new time variable S

The case when the attribute `slice` constraining the overall duration of the slices inside the time-line is set can be treated analogously.
As in the case time-lines, a slice can be bounded in duration and/or be executed periodically.

$\langle$`<slice end-of=`n`>` s `</slice>`$, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow$
  $\langle$`<var name=`S` value=`now`> </var>`
  do { s repeat { skip } } watching$(end(t, n) \in \pi \to t < S, S)$,
  $(\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle$ for a new time variable S

$$\begin{array}{l}
\langle\texttt{<slice end-of=}n\texttt{ periodical>}\ s\ \texttt{</slice>}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow \\
\quad\langle\texttt{<var name=}S\texttt{> </var>} \\
\qquad \text{repeat } \{ \\
\qquad\quad \texttt{<set name=}S\texttt{ value=}\mathit{now}\texttt{> </var>} \\
\qquad\quad \text{do } \{\ s\ \text{ repeat } \{ \text{ skip } \}\ \}\ \text{ watching}(\mathit{end}(t,n) \in \pi \to t < S, S)\ \}, \\
\quad(\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \qquad \text{for a new time variable } S
\end{array}$$

$$\begin{array}{l}
\langle\texttt{<slice end-of=}t\texttt{>}\ s\ \texttt{</slice>}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow \\
\quad\langle\texttt{<var name=}S\texttt{ value=}\mathit{now}\texttt{> </var>} \\
\qquad \text{do } \{\ s\ \text{ repeat } \{ \text{ skip } \}\ \}\ \text{ watching}(\mathit{now} < S + t, S), \\
\quad(\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \qquad \text{for a new time variable } S
\end{array}$$

$$\begin{array}{l}
\langle\texttt{<slice end-of=}t\texttt{ periodical>}\ s\ \texttt{</slice>}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow \\
\quad\langle\texttt{<var name=}S\texttt{> </var>} \\
\qquad \text{repeat } \{ \\
\qquad\quad \texttt{<set name=}S\texttt{ value=}\mathit{now}\texttt{> </var>} \\
\qquad\quad \text{do } \{\ s\ \text{ repeat } \{ \text{ skip } \}\ \}\ \text{ watching}(\mathit{now} < S + t, S)\ \}, \\
\quad(\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \qquad \text{for a new time variable } S
\end{array}$$

The `<par>` tag within a slice allows to spawn multiple processes in parallel.

$$\frac{\langle s_j, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow \langle s'_j, (\gamma',\varepsilon',\delta',\lambda',\mu',\pi')\rangle}{\begin{array}{l}\langle(\texttt{<par>}\ s_i\ \texttt{</par>})_{1\le i\le k}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow \\ \quad\langle(\texttt{<par>}\ s_i\ \texttt{</par>})_{1\le i<j}\ \texttt{<par>}\ s'_j\ \texttt{</par>}\ (\texttt{<par>}\ s_i\ \texttt{</par>})_{j<i\le k}, \\ \qquad(\gamma',\varepsilon',\delta',\lambda',\mu',\pi')\rangle\end{array}}$$

$$\frac{\langle s_j, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow (\gamma',\varepsilon',\delta',\lambda',\mu',\pi')}{\begin{array}{l}\langle(\texttt{<par>}\ s_i\ \texttt{</par>})_{1\le i\le k}, (\gamma,\varepsilon,\delta,\lambda,\mu,\pi)\rangle \Longrightarrow \\ \quad\langle(\texttt{<par>}\ s_i\ \texttt{</par>})_{1\le i\le k, i\ne j}, (\gamma',\varepsilon',\delta',\lambda',\mu',\pi')\rangle\end{array}}$$

The first (second) rule describes the case when the jth component has something (nothing) left to do.

3.2 Multimedia

The tags `<video>`, `<slide-show>` and `<audio>` spawn processes that are only visible in the memory. We only detail `<video>` here, the other multimedia tags are treated analogously.

We do not give a formal definition of the corresponding processes, but we assume that if such a process, say n, comes to an end it inserts the corresponding $end(t, n)$ signal to the signal set.
A video may be declared in suspended mode (for $t_0 = now$):

$$\begin{aligned}&\langle \texttt{<video name=}n\ \texttt{status=suspended>}\ s\ \texttt{</video>}, (\gamma, \varepsilon, \delta, \lambda, \mu, \pi)\rangle \Longrightarrow \\ &\quad (\gamma, \varepsilon + \{n \mapsto \mathrm{new}(\mu)\}, \delta, \lambda + \{\mathrm{new}(\mu)\}, \\ &\quad \mu + \{\mathrm{new}(\mu) \mapsto (\texttt{<video name=}n\ \texttt{status=}(t_0, \mathrm{suspended})\texttt{>}\ s \\ &\qquad\qquad \texttt{</video>}, \varepsilon)\}, \pi)\end{aligned}$$

In order to model the different status a video may be in, e.g. playing or stopped, we add this status to the syntax. Additionally this status indicates when a video was started. An `epkml` program is not to be able to observe these status; therefore an evaluation of the attribute `status` of a video that actually has another value than `opened`, `closed` or `suspended` yields the value `opened`. Now, videos may react to different signals; here we only state the rule for *play*.

$$\begin{aligned}&\langle s, (\gamma, \varepsilon, \delta, \lambda, \mu + \{l \mapsto \texttt{<video name=}n\ \texttt{status=}(t_0, z)\texttt{>}\ s'\ \texttt{</video>}\}, \\ &\quad \pi \uplus \{play(t, n)\})\rangle \Longrightarrow \\ &\quad \langle s, (\gamma, \varepsilon, \delta, \lambda, \mu + \{l \mapsto \texttt{<video name=}n\ \texttt{status=}(now, \mathrm{playing})\texttt{>}\ s' \\ &\qquad\qquad \texttt{</video>}\}, \pi)\rangle \\ &\qquad \text{for } t_0 < t\end{aligned}$$

Note that t_0 is the time when the video n has been started.

Conclusions

We have proposed a formal semantics for `epkml`, which allows restricted concurrency. This semantics is operational and may guide a possible implementation. As we mentioned, the semantics deals only with the control flow of a program execution. An interesting thing would be to study how this semantics can be unified with a SQL semantics and a semantics of the layout features. Another interesting thing would be to design a denotational semantics for `epkml`and develop a proper notion of implementation.

References

[Gol94] Charles Goldfarb. *The SGML Handbook.* Clarendon, Oxford, 1994.

[Gra95] Ian S. Graham. *HTML Sourcebook.* John Wiley & Sons, New York–etc., 1995.

[IT94] ITU-TS. Recommendation Z.100. CCITT Specification and Description Language (SDL). Technical report, ITU–TS, Genova, 1994.

[KKM96] Alexander Knapp, Nora Koch, and Luis Mandel. The Language EPKML. Technical report 9605, Ludwig–Maximilians–Universität München, November 1996.

[Plo91] Gordon D. Plotkin. Structural Operational Semantics. Lecture Notes DAIMI–FN 19, Aarhus University, 1981 (repr. 1991).