XAL: A Web Oriented Programming Language Based On Timed-Automata

Salvatore Campana  Luca Spalazzi
s.campana@computervaritt.it spalazzi@univpm.it
Computer VAR ITT, Verona, Italy DIIGA, Università Politecnica delle Marche, Ancona, Italy
Francesco Spegni
francesco.spegni@gmail.com
BINT S.r.l., Verona, Italy

Abstract

We developed XAL, a framework that, in our opinion, allows to build web-oriented applications and services in a more productive way. The core of the framework is a programming language based upon timed-automata. We believe this formalism reflects the nature of many web-oriented applications, each page being a state, and each link being a transition toward another state. Once the programmer defined the set of states that characterize the application, she/he can provide a behavior to each single state, binding the state to a small program written in its favorite programming language. Furthermore, we realized that often companies require an application to behave differently depending on some conditions over real-time. Our language, being a modified version of the timed-automata, allows the programmer to specify constraints over real-time in a declarative way, rather than mix them within the logic of the application.

1. Introduction

In this paper we present a programming paradigm and the related framework for building web-oriented applications and services in a more productive way, with respect to traditional languages. The core of the framework is XAL (XML Automata Language), a language based upon timed-automata [1, 17]. This paradigm, as also stated in [7, 19, 21], reflects the nature of many web-oriented applications, where each page can be viewed as a state and each link as a transition between two states. As original contribution, in our approach, the automaton describing the application is linked to the environment (e.g., the user, the back-end, . . . ) by means of appropriate “sensors” and “actuators”: pieces of code that are executed before and after the interpretation of a transition. Interacting with the environment, the sensors summarize the state of the system in order to produce the next input symbol for the automaton, while the actuators are sorts of output produced by the automaton itself. As a consequence, the web application design methodology that arises from our paradigm is in three steps: 1) the web application engineer designs the application in terms of its states, links among them, and time constraints (if any); 2) she/he defines the interaction with the environment in terms of input and output languages of the automaton; 3) she/he implements such interactions in its favorite programming language. For our experimental version we based XAL upon the PHP language, but in principle it could be used any other programming language. Furthermore, one could think to use another XAL automaton to model sensors and actuators, opening up the possibility of composing automaton definitions and of using some sort of stepwise refinement methodology.

Our approach has some advantages compared with traditional web programming paradigms. First of all, from a software engineering point of view, automata (as well as statecharts or UML state diagrams) are simple to use and design [12]. They are easily understandable by stakeholders and, thus, design validation is simpler. Second, using XAL the programmer is asked to write a model of the application (the automaton) together with its concrete behavior (the sensors and the actuators). Being the model part of the definition of the program, the developer is, in some sense, “forced” to be ordered when writing its program. Third, often companies require an application to behave differently depending on some conditions over real-time. XAL allows the programmer to specify temporal constraints in a declarative way, rather than mix them with the logic of the application. From a formal point of view, timed automata theory brings us a lot of work on automatic synthesis and verification [17, 18, 2] that we can use for developing a web application. Finally, we found that XAL is useful as a very simple but easy to understand workflow language, as well.
Even though this paper does not aim to compare our work in progress technology with well known tools in the world of workflows (among others, YAWL [23], WS-BPEL [3] and UML Activity Diagrams), we would like to remark that we successfully experimented XAL as workflow language in several real-world applications. In order to give a flavour of it, we used XAL in Figure 1 for describing the XAL interpreter itself.

In Section 2, we shall introduce a real-world case study that helped us to design our framework. Sections 3 and 4 describe the syntax and the semantics of XAL, defined upon the syntax and semantics of the concrete language. After that, a description of the architecture we realized around the language is given in Section 5, followed by the implementation of the case study we proposed (Section 6). Finally, Section 7 discusses how our work is related to other existing solutions.

2. Activating a phone line : a case study

Let us introduce a case study. A telephone company needed a web application to activate the phone and data services they provided. For example, once they received a request from the commercial side for activating a new phone line (rather than a DSL internet connection or a UMTS Sim card), the customer service needed to send a request to the so-called carrier, i.e. the company managing the communicating infrastructure. After that, the customer service should track the request status: if it is accepted or not, whether the carrier begins their technical works for activating the line or not, and finally whether the line is actually working when the carrier job is done. Moreover, in this application was very important to satisfy few temporal constraints: the carrier should communicate whether it accepts to activate the line or not within one day, and the line should be activated within thirty days from the customer’s request. If the latter constraint does not hold, the customer should be released from any duty toward the company. We shall see in Section 6 the solution we devised to this problem using XAL.

3. The syntax

The definition of the syntax is in two steps: first of all, we report the portion of the concrete language syntax that we are going to reuse; second, we define XAL as an “extension” of the concrete language.

3.1. From the concrete language . . .

The syntactic entities that we expect to be already defined in the concrete language are, intuitively, the following ones: names of variables, functions, classes and methods; bodies of functions, programs, evaluable expressions and values (meant as possible results of expressions). We baptize them, respectively: VarName, FunName, ClassName, MethodName, FunBody, Program, Exp, and Value.

In the following we shall refer to these entities as sets of well formed words or phrases of the language. For example, if we say that \( v \in \text{Value} \), we mean that \( v \) is a syntattically correct \( \text{Value} \) for the concrete language.

3.2. . . . to XAL

Here is the syntax of XAL. Even though we actually specified it through an XML-Schema, we describe it here through a set theoretic representation since we find this a more concise notation. The interested reader can refer to [6] for the XML-Schema definition.

\[
\begin{align*}
\text{FMName} & : = \text{FunName} \mid \text{MethodName} \\
& \quad \text{it is a function or method name;} \\
\text{>ActionType} & : = \{\text{function, method, webservice}\}; \\
\text{FilePath} & : = \text{it is the pathname of a file in the host system}; \\
\text{URI} & : = \text{it is a URI as defined by [4]}; \\
\text{Namespace} & : = \text{it is a namespaces as defined in [5].}
\end{align*}
\]

Definition 1 (Time constraints). Given a set of variables \( C \subseteq \text{VarName} \), called clock variables, and \( \mathbb{Q} \) the set of rational numbers, we call clock constraints the items of the following set:

\[
\text{TC} \ ::= \ C \geq C \mid C < C \mid C \leq C \\
\quad \mid C = \mathbb{Q} \mid C < \mathbb{Q} \mid C \leq \mathbb{Q}
\]

Definition 2 (Program). We call program a tuple as the followings:

\[
\langle \langle \Sigma, S, \hat{s}, F, C, TI, \tau \rangle, \langle V, AP, \iota, \alpha, \mu, \omega \rangle \rangle
\]

where \( \langle \Sigma, S, \hat{s}, F, C, TI, \tau \rangle \) is the timed-automaton:

\[
\begin{align*}
\Sigma & \subseteq \text{Value} \\
S & = \{s_0, \ldots, s_n\} \\
\hat{s} & \in S \\
F & \subseteq S \\
C & \subseteq \text{VarName} \\
TI & : S \rightarrow 2^{\text{TC}} \\
\tau & : S \times 2^{\text{TC}} \times \Sigma \times 2^{C} \rightarrow S
\end{align*}
\]
and \( \langle V, AP, \iota, \alpha, \mu, \omega \rangle \) is the concrete part of the program:

\[
V = \{v_1, \ldots, v_{n_V}\}
\]

where:

\[
v_i = \{\text{name} : \text{VarName}, \text{exp} : \text{Exp}\}
\]

\[
AP = \{a_1, \ldots, a_{n_{AP}}\}
\]

where:

\[
a_i = \{\text{type} : \text{ActionType}, \text{name} : \text{FMName},
\text{code} : \text{FunBody}, \text{class} : \text{ClassName},
\text{filepath} : \text{FilePath},
\text{uri} : \text{URI}, \text{ns} : \text{Namespace}\}
\]

\[
\iota : AP \to 2^V
\]

\[
\alpha : S \to AP
\]

\[
\mu : S \to AP
\]

\[
\omega : S \to \{1, 0\}
\]

As a shortening, we shall write \( s \xrightarrow{g,s,r} s' \) each time \( \exists s, g, \sigma, r, s' \text{ s.t. } \tau(s, g, \sigma, r) = s' \).

The actual semantics of the whole will be given in the next section. The intuitive meanings of each component are the followings:

- \( V \): the programmer can specify a set of variables to be shared among all the action and metric functions;
- \( AP \): the Action Pool gathers a bunch of pointers to pieces of code: they can be written in the concrete language (such as functions or methods) or exported through a SOAP interface (such as web services);
- \( \iota \): it maps each action function to a subset of the automaton variables, namely its input variables;
- \( \alpha \): this associates each state with its action function, or actuator;
- \( \mu \): this maps each state with its metric function, or sensor;
- \( \omega(s) = 1 \) if, and only if, \( s \) is interactive;

From the automaton point of view, instead, we have:

- \( \Sigma \): is the set of all possible symbols that a metric can emit;
- \( S \): is a finite set containing the names of all the states;
- \( \hat{s} \): is the initial state, the first loaded by the interpreter as current state;
- \( F \): is a set of states tagged as final that would break the cycle of the interpreter;
- \( C \): is a set of variables (disjoint from \( V \)) that can be referred as clock variables in the clock contraints and in the time invariants;

- \( TI \): given a state, it specifies zero, one or more temporal constraints for such state, called time invariants; we shall see in the semantics that if one state specifies such invariants, they should be met for a transition step to lead to such state;
- \( \tau \): this function describes which state is reached from one state, given an input symbol produced by some metric function, the specified time constraints hold and a set of time variables have been zeroed.

### 4. The semantics

Analogously to what we did for the syntax in Section 3, for the semantics as well we first introduce what we expect to come with the formal semantics of the concrete language and successively what we build upon that.

#### 4.1. From the concrete language . . .

From the semantical point of view, the concrete language should have a type \( \text{Env} = \text{VarName} \to \text{Value} \), i.e. a mapping from variables to values, and an operator:

\[
\text{eval} : \text{Program} \times \text{Env} \to (\text{Value} \times \text{Env})
\]

\( \text{eval} \) is actually the interpreter of the language. The Program shall be executed in a (possibly) non-empty initial environment and, besides computing a Value as its result, the operator also returns the environment at the end of the computation. Later on, we shall see that Programs will be evaluated starting from an initial environment and letting the Programs themselves to repetitively modify it, along the computation.

#### 4.2. . . . to XAL

Now we give an operational semantics to XAL, directly inspired by [17]. For the sake of brevity, we don’t give a formal specification to every single operator used in the following. Again, the interested reader may refer to [6] for further details.

**Definition 3 (Time increment).** Given a generic clock assignment \( u : C \to \mathbb{Q} \) and a positive rational number \( d \in \mathbb{Q}^+ \), we write \( \text{timeInc}(u, d) = u' \), if \( u' : C \to \mathbb{Q} \) and \( \forall c \in C. u'(c) = u(c) + d \).

**Definition 4 (Clock reset).** Given a clock assignment \( u : C \to \mathbb{Q} \) and a subset \( R \subseteq C \) of the clock variables, we write \( \text{clockReset}(u, R) = u' \) if \( u' : C \to \mathbb{Q} \) and:

\[
\forall c \in C. u'(c) = \begin{cases} 0, & \text{if } c \in R \\ u(c), & \text{otherwise} \end{cases}
\]
The last (semantic) operators are:

\[
\begin{align*}
\text{holdConstr} & : 2^{\mathcal{T}C} \times (C \rightarrow \mathbb{Q}) \rightarrow \{\text{true, false}\} \\
\text{buildEnv} & : \mathbb{V} \rightarrow \mathbb{Env} \\
\text{mergeEnv} & : \mathbb{Env} \times \mathbb{Env} \rightarrow \mathbb{Env} \\
\text{filterEnv} & : \mathbb{Env} \times \mathbb{AP} \rightarrow \mathbb{Env} \\
\text{buildProg} & : \mathbb{AP} \rightarrow \text{Program}.
\end{align*}
\]

holdConstr, taken a set of constraints and a clock, returns whether the clock satisfies the constraints. buildEnv takes the current values of variables in \(\mathbb{V}\) and creates an environment for the interpreter of the concrete language. mergeEnv augment the definition of the first argument with the mappings defined by the second one. If any variable is set by both the environments, it will assume the value of the second one. filterEnv, given an environment \(\mathbb{e}\) and an action pointer \(a\), will filter out all the variables of \(\mathbb{e}\) not included in \(\iota(a)\). Finally, buildProg takes an action pointer and translates it into a syntactically correct program in the concrete language.

buildProg produces a different code depending on the type of action pointer received as an argument. As an example, given an action \(a\) such that \(a\).type = method, then:

\[
\text{buildProg}(a) = \begin{align*}
\text{require_once } \langle\langle \text{filename} \rangle\rangle; \\
\text{$_\mathbb{env}$} = \text{$_\mathbb{automaton}$}(\text{\textquoteleft env\textquoteleft}); \\
\text{$_\mathbb{obj}$} = \text{$_\mathbb{automaton}$}(\text{\textquoteleft obj\textquoteleft})[\text{\textquoteleft \langle \text{name} \rangle \textquoteleft}]; \\
\text{if } (\text{$_\mathbb{obj}$} == \text{null}) \\
\{ \\
\text{$_\mathbb{automaton}$}(\text{\textquoteleft obj\textquoteleft})[\text{\textquoteleft \langle \text{name} \rangle \textquoteleft}] = \\
\text{$_\mathbb{obj}$} = \text{new } (a\text{.class})(); \\
\} \\
\text{$_\mathbb{obj}$}->\langle \text{name} \rangle(\text{$_\mathbb{env}$}); \\
\end{align*}
\]

buildEnv is responsible to produce an environment containing a special variable \(\text{$_\mathbb{automaton}$}[\text{\textquoteleft env\textquoteleft}]\). In brief, such variable denotes an array containing two more arrays: \(\text{$_\mathbb{automaton}$}[\text{\textquoteleft obj\textquoteleft}]\) that gathers all the automaton variables, and \(\text{$_\mathbb{automaton}$}[\text{\textquoteleft obj\textquoteleft}]\) needed to collect all the objects initialized by those action pointers having type method.

**Definition 5 (Abstract machine).** Given a generic program:

\[
\langle\langle \Sigma, S, \hat{s}, F, C, TI, \tau \rangle, \langle V, AP, \iota, \alpha, \mu, \omega \rangle\rangle,
\]

we call abstract machine a tuple

\[
\langle s, u, e \rangle
\]

such that \(s \in \Sigma, u : C \rightarrow \mathbb{Q}\) and \(e : \mathbb{Env}\).

**Definition 6 (XAL operational semantics).** Given a generic program:

\[
\langle\langle \Sigma, S, \hat{s}, F, C, TI, \tau \rangle, \langle V, AP, \iota, \alpha, \mu, \omega \rangle\rangle,
\]

then its execution begins from the initial state \(\langle \hat{s}, u_0, \text{buildEnv}(V) \rangle\), where \(u_0\) sets all the clock variables to zero. The abstract state machine then proceeds executing the following transition rules:

\[
\begin{align*}
\text{(delay)} \\
\langle s, u, E \rangle \xrightarrow{d} \langle s, u', E \rangle & \quad \text{where:} \\
& d \in \mathbb{Q}^+ \\
& \text{holdConstr}(TI(s), u) = \text{true} \\
& u' = \text{timeInc}(u, d) \\
& \text{holdConstr}(TI(s), u') = \text{true}
\end{align*}
\]

\[
\begin{align*}
\text{(initial step)} \\
\langle s_0, u_0, E \rangle \rightarrow \langle s_0, u_0, E' \rangle & \quad \text{where:} \\
& \text{holdConstr}(TI(s_0), u_0) = \text{true} \\
& (\text{void, } E_1) := \text{eval(buildProg}(\alpha(s_0)), \\
& \text{filterEnv}(E, \alpha(s_0))) \\
& E' := \text{mergeEnv}(E, E_1)
\end{align*}
\]

\[
\begin{align*}
\text{(step)} \\
\langle s, u, E \rangle \rightarrow \langle s', u', E' \rangle & \quad \text{where:} \\
& (\sigma, E_1) := \text{eval(buildProg}(\mu(s)), \\
& \text{filterEnv}(E, \mu(s))) \\
& E_2 := \text{mergeEnv}(E, E_1) \\
& \exists g, r, s' : s \xrightarrow{g,s,r} s' \in \tau \\
& \text{holdConstr}(g, u) = \text{true} \\
& u' := \text{clockReset}(u, r) \\
& \text{holdConstr}(TI(s'), u') = \text{true} \\
& (\text{void, } E_3) := \text{eval(buildProg}(\alpha(s')), \\
& \text{filterEnv}(E_2, \alpha(s'))) \\
& E' := \text{mergeEnv}(E_2, E_3)
\end{align*}
\]

5. The structure of the interpreter

We would like to give a more practical view of our work. As already mentioned, in Figure 1 we have represented the flow of execution of the interpreter through a XAL automaton. We just underline how the programmer only needs to describe the automaton through a XAL file and a bounch of PHP functions or methods.

The presentation layer is needed when some state is marked to be interactive. In such case, indeed, the state of the automaton is stored for the session and the handler of the presentation layer (e.g. a web server) is invoked. When the user will fill the interface with its input, the responsible for the presentation layer would restore the session (with
the stored state of the automaton) where the state has been “enriched” with the user input. The metric will then analyze the enriched state and choose the next symbol to be emitted.

Since we wrote our interpreter in PHP itself (i.e. the same language we chose as concrete language), we exploited the reflexive capabilities of PHP. This can be done, obviously, with any language allowing to execute code generated at run-time. Otherwise the designer of the interpreter would be required also to develop a layer responsible for generating code and invoking the concrete language’s tools (interpreter, linker, ...).

From what concerns the real-time features, we just mention that we set the clock to have a tick per microsecond, exploiting the granularity of the function microtime.

### 6. Activating a phone line: implementation

In Figure 2 we represented the XAL automaton we devised to implement the application of our case study. We adopted the following conventions: the rounded rectangles are states, each one containing its name. We assume that all the states are interactive with the only exception of LogDelay that appears in bold-italic style. Every edge of the automaton must be associated with a label, i.e. a symbol emitted by the metric function of the source state. An edge may also specify (in square brackets) conditions over some time variables or directives that set back to zero other time variables. This means, as we explained in Section 4, that the step denoted by the edge should be taken only if (just before the step is taken) the time constraints are satisfied. If not any time constraints appear, the edge is time independent. For example, from state OK two (time-independent) steps can be taken while from Completed we can take three different (time-dependent) steps, depending on whether the symbols Visible or NotVisible are emitted, and whether the transition takes place within thirty days or not.

In Figure 3, we report a fragment of the XAL program of this case study. We focused on the description of the state SendRequest, on the four transitions leaving such state and on the transition leaving the state Active. This portion of the automaton, indeed, gathers most of the characteristics of the language: the association between states and action/metric functions, the clock reset directives and the time-dependent transition rules.

We are developing a set of tools (e.g. a plugin for the Eclipse IDE) to rapidly and visually create, modify, and interpret XAL source files. When the tools will be fully developed, the programmer should draw the diagram in Figure 2, specifying which action functions and metrics should be used by each state. The source code in Figure 3 will be automatically generated. Next, the methods and services specified as action functions must be coded. Finally, in order to
let the users to invoke the web application, an index.php file is needed, whose only task is to load the XAL interpreter specifying the XAL source file to be executed.

7. Final Discussion

The idea of defining a program as an automaton is, of course, not new. Not considering the foundational works of Turing on computable functions [22] and the whole theory of automata [16], the first actual language structuring programs as automata we met were Harel’s Statecharts [11, 14, 13] and Shalyo at al. SWITCH technology [20, 10].

With respect to Statecharts, XAL has less syntactic constructs, since we chose to be as simple as possible in this first definition of the language. The main differences with Statecharts, though, concern synchronism: the first one is that Statecharts semantics[14, 13] provide both synchronous and asynchronous behaviors. With the latter, you can naturally devise applications reacting to environment changes and to events. On the contrary, XAL is a synchronous language since it has not been designed for reactive systems. The second main difference is a consequence of the first: in Statecharts temporal constraints are used to generate events to which the application can immediately react. On the contrary, on XAL temporal constraints can only activate or deactivate some transitions, at a certain time, but they do not induce the automaton to leave the current state for a new one. We chose these two behaviors in order to stick on timed-automata definition and use the many verification and analysis tools designed for them.

The added value of our work with respect to the above ones is that XAL is designed to be the core of a web-, automata- and time- based developing framework. With it, we try to fill a gap we found in the web developing scenario summarizing our experience with the web services and applications we devised and deployed so far. Indeed, most of the times they turned out to be state-based applications. Until now, we realized about 40 applications with XAL [6], most of them dealing with service activation processes (like the example in Section 2), while others monitor the status of some network overall the country, acting in some “clever” way if some fault is discovered.

Concerning web applications, several authors used automata and timed automata. Nevertheless, most of them deal with web application verification (e.g., see [9, 15, 21]). The work closest in spirit to ours is [7], even though they use automata in order to express user behavior patterns, designing highly customizable web applications. In a XAL program, instead, the automaton describes the behavior of the application itself.

Furthermore, with respect to most of the previously mentioned works, XAL is original since combines the design of the automaton with a concrete programming language. It is a “two-level” language rather than a modeling language upon a programming language. It allows the programmer to design the application by stepwise refinements of states and links, until it finds its automaton is expressive enough to be directly filled with the concrete pieces of code implementing the behaviors of each state.

This is an important difference also with respect to the more common web developing frameworks, such as .NET, J2EE, PHP, . . . The latters actually compel the developer to translate the application state-based model in their object-based model. Thus the complexity of the software components will hide, modification after modification, the initial state-based model that described the application. Another key difference between XAL and the above mentioned platforms is about writing time-aware web applications. As long as we know, if you want to create a timed application with them, you either choose to implement your own scheduler which enforces your time constraints, or you rely on tools for real-time programming. Since the lasts are mainly targeted for embedded devices, they are of course not suitable for web applications.

At now, we are terminating the formal definition of XAL using the Maude tool [8]. The executable rewriting logic framework of the latter would give us a more solid formal (still executable) basis for the language. Finally, we are evaluating how to extend the framework to include concurrency and cooperation of automata.

References


<?xml version="1.0" encoding="UTF-8"?>
<Automaton ...
<GlobalState>...

<Clocks>
  <Variable Name="cOrder" />
  <Variable Name="cSend" />
</Clocks>

<ActionPool>
  ...
  <Action Id="SendRequestInterface" Type="object">
    <Inputs>...</Inputs>
    <System Path="./DeliveryWeb.php" Class="DeliveryWeb"
      Name="SendRequestInterface" />
  </Action>
  ...
</ActionPool>

<States>
  ...
  <State Interactive="true" Id="SendRequest"
    IdAction="SendRequestInterface"
    IdMetric="SendRequestMetric" />
  ...
</States>

<InitialState IdState="NewOrder"/>

<Transitions>
  ...
  <Transition IdInputState="SendRequest"
    MetricValue="CarrierK0"
    IdOutputState="KO" >
    <ClockConstraint Exp="cSend &lt;= 1 day" />
  </Transition>
  <Transition IdInputState="SendRequest"
    MetricValue="AtWork"
    IdOutputState="WorkInProgress" >
    <ClockConstraint Exp="cSend &lt;= 1 day" />
  </Transition>
  <Transition IdInputState="SendRequest"
    MetricValue="AtWork"
    IdOutputState="LogDelay" >
    <ClockConstraint Exp="cSend &gt; 1 day" />
  </Transition>
  <Transition IdInputState="SendRequest"
    MetricValue="CarrierK0"
    IdOutputState="LogDelay" >
    <ClockConstraint Exp="cSend &gt; 1 day" />
  </Transition>
  <Transition IdInputState="SendRequest"
    MetricValue="CarrierK0"
    IdOutputState="LogDelay" >
    <ClockConstraint Exp="cSend &gt; 1 day" />
  </Transition>
  <Transition IdInputState="Active"
    MetricValue="NewOrder"
    IdOutputState="NewOrder" >
    <ClockReset ClockVar="cOrder" />
  </Transition>
  ...
</Transitions>
</Automaton>

Figure 3. The xml of the case study