Advanced XFG language: Extending XFG language with Energy-Aware Timed Requirement Properties

Kang, Eun-Young; Schobbens, Pierre

Publication date:
2013

Document Version
Early version, also known as pre-print

Advanced XFG language: Extending XFG Language with Energy-Aware Timed Requirement Properties

Eun-Young Kang and Pierre-Yves Schobbens

PReCISE Research Centre
Computer Science Faculty, University of Namur, Belgium
{eun-young.kang,pierre-yves.schobbens}@fundp.ac.be

Abstract. This report presents an advanced XFG (extended function-block graphs) formal specification language, which can be used as interchange format for Timed Automata-based input modeling languages and model checkers. In particular, XFG is designed to provide support for modeling and analyzing energy-aware real-timed (ERT) systems. In comparison to our early version of XFG, an enriched XFG language is defined with the XFG temporal logic, consisting of timing and energy-constrained modalities: Section 1 informally represents a general introduction to the advanced XFG. The concrete E-BNF syntax rules complying with the extension are presented in Section 2. Section 3 defines complete syntax and semantics of the langauge. Section 4 gives a running example of the Brake-By-Wire (BBW) system and part of the XFG specification of the system. In section 5 we study how to obtain computer-aided analytical leverage through well established analysis tools. This study will provide a basis for automatic model transformations between XFG, and (Timed-Automata based) specification languages for model checkers.

1 Advanced XFG (Extended Function-block Graphs) Language

In regard to modeling and analysis support for ERT system behaviors, XFG allows the specification of executional constraints on system functions (A.K.A processes, block graphes, components, e.g., their internal state transitions) at high level. To facilitate the guarantee of system safety, correctness and performance, it is expected that XFG as an interchange modeling format language would form the basis for eliciting, validating, and consolidating various kinds of behavioral concerns. For example, such behavioral concerns can be related to requirements specifications and elicitation, the design of verification and validation cases. From a system design point of view, the behavioral issues of particular concern should include not only the executions of system functions and function structuring, but also the definition of system operational situations, requirements specifications and refinements.

With the previous versions of XFG, application specific behavioral concerns (e.g., the definitions of executional behaviors on an function process) can only be treated in textual or graphical implementations. This is considered as a point of extension, as the provision of precise specifications of the issues mentioned above is fundamental of many overall design decisions including requirements elicitation and refinements, function structuring, the obtainment of its analytical leverage through well established analysis methods and tools. Indeed, nevertheless the actual ERT behaviors of system functions are captured in XFG, there is still a need of explicitly annotating the application requirements with related bounds (e.g., invariants of data, timing and energy constraints).

To address such challenges, an advanced XFG language has been developed. We introduce the definitions of related key concepts in following sections. The aim of the enriched XFG is to enable a more precise specification of various behavioral concerns and to provide a gateway for supporting model transformations from XFG to well established analysis methods and tools for ensuring the analytical leverage.

1.1 System Specification Language

An XFG (eXtended Function-block Graphs) language is an extension of timed automata [2]. It is a formal specification interchange format language for modeling and analysis of energy-aware real-time (ERT) systems. The XFG format is a textual description language and it captures the axiomatic and operational specification of function aspects, and ERT behavior. The XFG language aims to establish interoperability of various tools by means of model transformations to and from XFG: The XFG is designed as an engineering language for formal specification and verification, serving as the Hybrid and Timed Automata (TA) [2, 1] based input modeling language for various model checkers such as UPPAAL series tool [4, 16], KRONOS [8, 7], and HYTECH [11, 10, 12], etc.

An XFG system consisting of a number of graphes (processes) provides a simple representation for high-level specification languages and is suitable for modeling interprocess communication by value or signal passing though data channels. The basic building blocks for an XFG system are presented by processes and two basic constructions of the process in XFG are locations and edges. The process in XFG system represents a single thread of execution. Interprocess communication is represented by the synchronous edges. They communicate by means of shared variables or by synchronous value passing.

The XFG process permits two-way synchronization communication (rendezvous-style) on complementary input and output actions, as well as broadcast actions. An edge labeled with a synchronization $l!v$ with another labeled $l?v$ or an arbitrary number of receivers $l?v$, where l is a synchronization channel name and v is a share variable. Any receiver can synchronize in the current state must do so. If there are no receivers, then the sender can still execute the $l!$ action, i.e. sending is never blocking.

The XFG language extends classical TA with energy consumption information both on locations and edges of an XFG process (which is seen as a timed automaton). The energy label on a location represents the rate of energy consumption (continuous energy consumption) per time unit for staying in that location. The energy label on an edge represents the discrete energy consumption for taking the edge. Thus, every run in the XFG process has a energy consumption, which is the accumulated energy (either energy rate or discrete energy consumption) along the run of every delay (continuous) and discrete edge. The energy consumption variable in the XFG process can be viewed as an hybrid variable[1], therefore the XFG processes are special cases of liner hybrid automata [1], in which all continuous variables are clocks, except the energy, which is never used for the executions in the XFG system.

[1] An hybrid variable is a variable which can have different slops on different locations

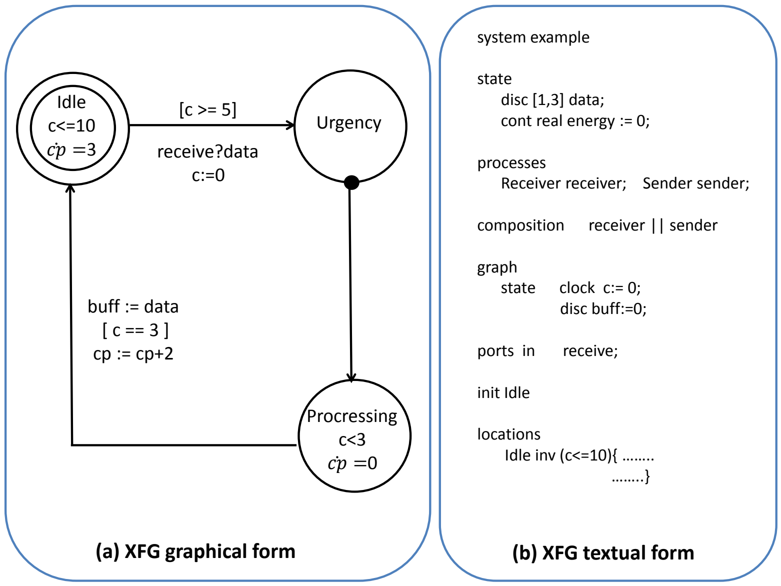

Fig. 1. A combination of XFG attributes: A receiver component as an XFG

Particular key features of XFG are that: 1. It provides a general form of urgency. Edges can either be urgent or non-urgent. Urgent edges marked with a small dot (see `Urgency` location in Figure 1) indicate that they have to be executed immediately once the location has been enabled without letting time pass. This form allows easy modeling of edge that triggers on data and time conditions; 2. It allows information for continuous or discrete consumption of resources, e.g., energy, on both locations and transitions. Locations are guarded with invariants, which forces control out of a location by taking an enabled edge as soon as the location of the process and the invariant are inconsistent.

Figure 1 shows a simple XFG graph representation of a single process in the XFG system, both in its graphical (Fig.1.a) and textual (Fig.1.b), that receives messages and puts the message into a buffer. A message is received (from another process, not visualized here) through the `receive` input action. It receives data between 5 and 10 seconds then immediately goes through the `Urgency` location. Because of the urgency semantics, the edge at the source location (`Urgency` location) will be taken without any delay. This edge is indicated by the keyword `prompt` in Listing 1.1 (in line 14) and by the black dot at the source of the edge in Figure.1.(a). Afterwards, it takes three seconds to process the message.

The message is subsequently placed in a buffer, modeled by the data element `buff`. The delay is enforced by the clock `c`. The system will leave the `Processing` location when the moment `c` becomes three, which is exactly three seconds after the location was entered. We put a constraint $c \geq 3$ on the edge from `Processing` to `Idle`.

```
1 Init
2     Idle
3
```

```
4   locations
5       Idle inv(c<=10){    when (c<=10 && c>=5)
6                                   do c:=0;
7                           goto Urgency
8
9                           when not(c>=5 && c<=10)
10                                  do dot energy := 3;
11                          goto Idle
12                      }
13
14  Urgency {    when true prompt
15                goto Processing
16           }
17
18  Processing inv(c<3){  when true
19                              do dot energy:=0;
20                         goto Processing
21
22                         when c==3
23                              do buff := data;
24                                 energy := energy+2;
25                         goto Idle
26                 }
```

Listing 1.1. XFG Process and its edges on Idle, Urgency and Process locations

The `receiver` process has a certain energy consumption, captured both in its graphical XFG (with $\dot{cp}$ or `cp` in Fig.1.a) and its textual representations (with `dot energy` or `energy` in Listing 1.1 in lines 10, 19, 24), where $\dot{cp}$ is a rate of energy consumption per time unit during a stay in the `Idle` location, whereas `cp` is a discrete energy consumption allocated on the edge as an update.

As soon as the `receiver` process is triggered (modeled by the `Idle` location) the value `energy` grows with rate 3 ($\dot{cp} = 3$), until the actual `receiver` is taking a place in the location `Processing`. With no continuous energy consumption in the location `Processing` (rate 0), it will be ready to receive data from other processes. In that case a two-units (discrete) energy is consumed on the edge from `Processing` to `Idle`.

1.2 Property Specification Language

In regard to specifying application requirements, which should comply with requirement constraints (i.e., application specific behavioral concerns), and validating the correctness of such application requirements (A.K.A application system properties), we introduce the XFG logic which consists of timing and energy constrained modalities.

The specification language for properties is a temporal logic based on CTL (Computational Tree Logic) and TCTL (Timed CTL) [6] and WCTL (Weighted Computation Tree Logic) [5]. TCTL variants are real-time extensions of CTL and WTCL extends CTL with cost constraints. These extensions are categorized by either augmenting temporal operators with time bounds or using a cost function:

- **Temporal operators with time bounds**: Temporal operators of CTL like $E[\phi_1 U \phi_2]$, $EF\phi$ and so on, can be equipped with a time constraint in a succinct way. For instance, the formula $A[\phi_1 U_{\leq 7} \phi_2]$ intuitively means that along way any path starting from the current state, the property ϕ_1 holds continuously until within 7 time-units ϕ_2 becomes valid. It can be defined by $z\ in\ A[(\phi_1 \wedge z \leq 7) U \phi_2]$. Alternatively, the formula $EF_{\leq 5}\phi$ means that along some path starting from the current state, the property ϕ becomes valid within 5 time-units, and is defined by $z\ in\ EF(z \leq 5 \wedge \phi)$.

In other words, $EF_{\leq c}\phi$ denotes that a ϕ-state is reachable within in c time-units. The dual expression, $AF_{\leq c}\phi$ is valid if all executions lead to a ϕ-state within c time-units.

- **Cost function**: In a consideration of the energy consumption on the XFG system, i.e., a cost of resource such as memory, CPU, etc., we can analyze if a certain resource consumption on all possible behaviors of the system within the available resource provided. A cost constraint or cost function is the accumulated resource consumption. For instance, the formula $AF_{cost \leq min}\phi$ means that for all execution paths, the ϕ-state is eventually reached within *min* cost. The dual expression $EF_{cost \leq max}\phi$ denotes that there is a path in which the ϕ-state is reached within a maximum resource cost *max*.

In our CTL in XFG language setting, called CTL_{XFG} (or CTL as a simple way), we use a similar approach mentioned above to our property specification in XFG language (namely XFG property) by adapting both time- and cost-bounds constraints. Indeed we use the reset operator followed by a path quantifier. The reset operator is expressed by assignments to clocks, energy consumption functions, and variables of the XFG property. In our CTL variant $(z := 0)\&(EF(z \leq 5 \wedge \phi))$ expresses that $z\ in\ EF(z \leq 5 \wedge \phi)$ and $EF_{\leq 5}$. Furthermore, $(cost := 0)\&(EF(cost \leq max \wedge \phi))$ denotes $EF_{cost \leq max}\phi$.

Notice that it is not allowed to write a bounded response time property like ”there exist some unknown time t such that if ϕ_1 holds, then before time t property ϕ_2 holds”. For instance,

$$\exists\, t.\ z\ in\ (AG_{\geq 0}(\phi_1 \Rightarrow AF(z < t \wedge \phi_2)))$$

This formula can be expressed in our XFG property as below. However, such quantifications over time makes analysis undecidable. Therefore, in our XFG property a clock constraint should be defined by an assignment with its time-bound t, which ranges over $\mathbb{R}$:

$$AG(\phi_1\ \Rightarrow\ (z := 0)\&AF(z < t \wedge \phi_2))$$

The core syntax defines two temporal operators, AU and EU. The formula $A\phi_1 U\phi_2$ is satisfied in a state if for all paths starting from the state, the property ϕ_1 holds continuously until ϕ_2 holds. $E\phi_1 U\phi_2$ is satisfied if there at least one such computation path exists. From those two main operators, we have several derived operators:

- $EG\phi$: There is a path in which every state satisfies ϕ
- $EF\phi$: There is a path on which a state satisfies ϕ
- $AG\phi$: For all paths, along way any path starting from the current state, every state satisfies ϕ
- $AF\phi$: For all paths, along way any path starting from the current state, some state satisfies ϕ

Our atomic proposition have two types and such expressions are defined:

- Boolean value expressions that specify conditions on values of variables and clocks of the system (states) and the property;

– Allocation expressions that specify conditions on locations of the system (states) and denoted as *Proc#loc*, where *Proc* is an identifier of a single XFG process and *loc* is an identifier of the current location where the XFG process is allocated. In other words, a single XFG process of the XFG system is currently allocated at the location *loc*. An allocation expression evaluates whether or not for a single XFG process in an XFG system control is at some specified location. Thus, *Proc#loc* will be true if for process graph *Proc* control is at location *loc*.

Finally, the enriched XFG textual presentation of the example is given in Listing 1.2, where the specification of properties and the XFG of sender process are implemented. This advanced XFG is combined with the XFG of receiver shown in Listing 1.1 and constitutes a whole XFG system example where the sender and receiver processes communicate through the synchronization port channels. (The graphical details are omitted)

```
system example

% variables used in property are defined here
property variables
    clock c1, clock c2;
    disc int data, buffer ;


% property specification is given here
properties

EF(sender.data == receiver.buffer) % data correspondence -- positive

AG((sender#Idle and ( 5<=c1 and c1<=10)) imply (receiver#Processing)
% location correspondence -- positive

AG(c1:=0&EF(c1>1)) % non-zenoness -- positive

state disc [1,3] data;
      cont real energy :=0;

% define processes (executable block graph)
processes
      Receiver receiver;
      Sender sender;

composition sender || receiver

block graph sender

state clock c1 :=0;
      disc int data:=0;

ports out send;

Init
    Idle

locations
    Idle inv(c<=10){
                  when (c1<=10 && c1>=5 && data<=3)
                  synch send!data;
                          do c1:=0;
                             data:=0;

                        goto Idle

```

```
                    when not(c1>=5 && c1<=10 && data<=3 )
                      do dot energy := 2;
                         data := data+1;
                    goto Idle
                }
```

Listing 1.2. Advanced XFG with Property Specification

2 The concrete XFG syntax, notation, and grammar

The concrete E-BNF grammar rules are presented in syntax charts. End symbols are presented with under-bars such as

—• <u>*terminal*1</u> •—• <u>*terminal*2</u> •—

The terminal symbols present the ascii representation of the keywords of the XFG language. Each rule is labeled with its defining nonterminal symbol.

—• *nonterminal* •—

The start symbol of the grammar is the nonterminal ”system”. A *system* specification in XFG is composed of the six main parts, *System Definition, User Definition, State Definition, Process Definition, Behavior Definition* and *Process Type Definition*. Each part will be investigated more detail in the following sections.

2.1 System Definition

The *system definition* clause specifies the global variables of the system. All variables have to be initialized:

- *system definition*

  —• <u>**system**</u> •—• *ident* •—•
  •—• *userdefs* •—• *type* •—•
  •—• *propertyvariables*
  •—• *properties* •—•
  •—• *statedef* •—•
  •—• *processdefs* •—•
  •—• *behaviordef* •—•
  •—• *processtypes* •—

  – the system heading part, formed by the keyword **system** and a unique identifier.
  – *userdefs* : the user defined types.
  – *propertyvariables* : the specification variables used in the property specification.
  – *properties* : the properties specifications.
  – *statedef* : the global state definition and all global variables.
  – *processdefs* : the process definitions, defining the processes in the system together with their types.
  – *behaviordef* : the behavior definition, defining how the processes in the system are composed.
  – *processtypes* : the process type definitions, defining the structure of the process.

2.2 User Definition

The *user definition* clause presents one kind of user-defined type, constant type:

- *userdefs*

—• **define** •—• (•—• *ident* •—• , •—• *constant* •—•) •—• ; •—

- *ident*

—• *letter* •—

- *letter*

—• *a* •—
—• … •—
—• *z* •—
—• *A* •—
—• … •—
—• *Z* •—

2.3 Property Variables

The *property variables* clause defines the property specification variables where the variables can be used in the properties. Thus, these variables are not part of the system:

- *propertyvariables*

—• **property variables** •—• *type* •—• *ident* •—• ; •—

2.4 Properties

The *properties* clause specified the properties to be verified:

- *properties*

—• **properties** •—• *property* •—• ; •—

- *property*

—• *property* •—• **and** •—• *property* •—
—• *property* •—• **or** •—• *property* •—
—• **not** •—• *property* •—
—• **EG** •—• *property* •—
—• **EF** •—• *property* •—
—• **AG** •—• *property* •—
—• **AF** •—• *property* •—
—• (•—• *property* •—• **EU** •—• *property* •—•) •—
—• (•—• *property* •—• **AU** •—• *property* •—•) •—
—• (•—• *ident* •—• **:=** •—• *vexpr* •—•) •—• **&** •—• *property* •—
—• *ident* •—• **#** •—• *ident* •—
—• *relation* •—
—• (•—• *property* •—•) •—

For the model checkers the allowed value expressions could be more limited than what is specified below. Only expressions of the following form are allowed: $x - y \sim c$, $x \sim y$ and $x \sim c$, where $\sim \in \{<, \leq, >, \geq, ==\}$, x and y are variables and c is a constant. The *state reference* is only used in properties to refer to local variables:

- *vexpr*

  —• *unary* •—• *vexpr* •—
  —• *binary* •—• *vexpr* •—• + •—• *vexpr* •—
  —• *binary* •—• *vexpr* •—• − •—• *vexpr* •—
  —• *binary* •—• *vexpr* •—• / •—• *vexpr* •—
  —• *binary* •—• *vexpr* •—• * •—• *vexpr* •—
  —• *primary* •—• *ident* •—
  —• *state reference* •—• *ident* •—• **#** •—• *ident* •—
  —• (•—• *vexpr* •—•) •—

- *boolexpr*

  —• *relation* •—• *relation* •—
  —• *combination* •—• *boolexpr* •—• *and* •—• *boolexpr* •—
  —• *combination* •—• *boolexpr* •—• *or* •—• *boolexpr* •—
  —• *negation* •—• *not* •—• *vexpr* •—
  —• *constant* •—• *true* •—
  —• *constant* •—• *false* •—
  —• (•—• *boolexpr* •—•) •—

- *relation*

  —• *vexpr* •—• == •—• *vexpr* •—
  —• *vexpr* •—• != •—• *vexpr* •—
  —• *vexpr* •—• < •—• *vexpr* •—
  —• *vexpr* •—• > •—• *vexpr* •—
  —• *vexpr* •—• ≤ •—• *vexpr* •—
  —• *vexpr* •—• ≥ •—• *vexpr* •—

2.5 State Definition

The *state definition* clause specifies the global variables of the system. All variables have to be initialized:

- *statedef*

  —• **state** •—
  —• **vartype** •—• **type** •—• **[** •—• *vexpr* •—• **,** •—• *vexpr* •—• **]** •—•
  •—• *ident* •—• := •—• *vexpr* •—• ; •—

- *vartype*

  —• *disc* •—
  —• *cont* •—
  —• *clock* •—

- *type*

  —• *integer* •—
  —• *real* •—
  —• *clock* •—

2.6 Process Definition

The *process definition* clause states the processes of the system. Each process is defined by a unique identifier and a process type. It is not allowed to use a dot ('.') in an identifier in the process definition. However the dot can be used in the transition definition as a special continuous variable, i.e., cost rate in terms of the clock. In this case that transition should be the delay transition.

- *processdefs*

$$\to \underline{\textbf{processes}} \to ident_1 \to \underline{;} \to ident_2 \to \underline{;} \to \ldots \to ident_n \to \underline{;} \to$$

2.7 Behavior Definition

The *behavior definition* clause specifies how the processes, which are specified in the previous clause, communicate. This is done using parallel composition:

- *behaviordef*

$$\to \underline{\textbf{composition}} \to ident_1 \to \underline{\|} \to ident_2 \to \underline{\|} \to \ldots \to ident_n \to$$

2.8 Process Type

The *process type* clause defines the structure of the processes. A process (type) is defined by elements:

- a name.
- a state definition, defining the local variables. Variables have to be initialized.
- a set of communication ports. For each port, a direction is specified *in* or *out*, and its type (possibly empty).
- an initial location.
- a set of location definitions.

- *processtypes*

$$\to \underline{\textbf{block graph}} \to ident \to$$
$$\to \underline{\textbf{state}} \to \underline{\textbf{vartype}} \to type \to ident \to \underline{:=} \to vexpr \to \underline{;} \to$$
$$\to \underline{\textbf{ports}} \to \underline{\textbf{in}} \to ident \to \underline{;} \to$$
$$\to \underline{\textbf{ports}} \to \underline{\textbf{out}} \to ident \to \underline{;} \to$$
$$\to \underline{\textbf{init}} \to ident \to$$
$$\to \underline{\textbf{locations}} \to locationdef \to$$

A *location* is defined by its name and a set of outgoing transitions. An invariant can be specified for a state, limiting the allowed data values for this location. Furthermore, a location can be declared committed, meaning that it has to be left immediately upon entering, without any other transitions interfering:

- *locationdef*

$$\to \underline{\textbf{committed}} \to ident \to$$
$$\to ident \to \underline{\textbf{inv}} \to boolexpr \to$$
$$\to \underline{\{} \to transitiondef \to \underline{\}} \to$$

A *transition* is defined by a guard. It defines when a transition is enabled, an optional state update, i.e., a set of assignments to local and global state variables, an optional communication definition, and a destination location. The *prompt* keyword defines transitions to be urgent:

- *transitiondef*

–• **when** •–• *boolexpr* •–• **prompt** •–
–• **synch** •–• *ident* •–• ? •–• *ident* •–• ; •–
–• **synch** •–• *ident* •–• ! •–• *vexpr* •–• ; •–
–• **broadcast** •–• *ident* •–• ! •–• *vexpr* •–• ; •–
–• **do** •–• *ident* •–• := •–• *vexpr* •–• ; •–
–• **dot** •–• *ident* •–• := •–• *vexpr* •–• ; •–
–• *functiondef* •–• ; •–
–• **goto** •–• *ident* •–

- *functiondef*

–• *ident* •–• () •–• { •–
–• **if** •–• (•–• *boolexpr* •–•) •–
–• *ident* •–• := •–• *vexpr* •–• ; •–
–• } •–

3 XFG complete syntax and semantics

3.1 Core syntax and semantics: XFG process

We define first a core syntax for an XFG system, on which the dynamic semantics is based. In core syntax, an XFG system is defined as a single, global graph. Also at core syntax level we do not worry about static semantics issues like type correctness. In the following, we use some abstract syntax domains that are assumed to be provided by the data model:

Definition 1. *A* data language *provides the following syntactic domains:*

- *V : a finite set of variables,*
- *$V_c \subseteq V$: a subset of clock variables,*
- *Expr: value expressions (over the set V of variables),*
- *Bexpr $\subseteq$ Expr: the subset of Boolean expressions.*

An XFG consists of a fixed, finite number of processes. The control part of any process is described as a finite state machine. The full state space is given by a set of variables (which can be local to the process or shared between processes), communication channels, clocks, and energy consumption functions. Edges of an XFG process can be marked as urgent, implying that they should be taken as soon as they are enabled. Processes of an XFG are executing asynchronously in parallel. They communicate by means of shared variables or by synchronous value passing. The XFG process permits two-way synchronization communication (rendezvous-style) on complementary input and output actions, as well as broadcast actions.

Definition 2. *An XFG process is a tuple* $\langle Dtype, Init, L, l_0, I, E, H, U, CP\rangle$ *where*

- $Dtype : V \rightarrow \{disc, cont, clock\}$ *assigns to each variable a dynamic type: discrete, continuous, or clock. The sets* V_{disc}*,* V_{cont}*, and* V_c *are defined as* $V_t = \{v \in V \mid Dtype(v) = t\}$ *for* $t \in \{disc, cont, clock\}$*,*
- $Init \in Bexpr$ *indicates the initial condition for the process. A set of dotted variables* $\dot{V} \in V_{disc}$ *represents different rates of increasing energy,*
- L *is a finite set of locations,*
- $l_0 \in L$ *is the initial location,*
- $I : L \rightarrow Bexpr$ *assigns an invariant to each location,*
- H *is a finite set of synchronizing action labels,*
- $E \subseteq L \times Bexpr \times 2^{V\times Expr} \times H \times L$ *is a set of edges, represented as tuples* $\langle l, g, h, u, l'\rangle$ *where*
  - $l \in L$ *is the source location,*
  - $g \in Bexpr$ *is the guard,*
  - $h \in H$ *is a label for synchronization* $\{h!x, h?x | \{x\} \subseteq Expr, \{v\} \subseteq V\}$*, where* x *and* v *are either empty or sequences of expressions or variables,*
  - $u \subseteq V \times Expr$ *is an update,*
  - $l' \in L$ *is the destination location.*

  Note that an assignment is defined as a set of pairs $\langle v,x\rangle$ *where* v *is a variable and* x *is an expression whose value is to be assigned to the variable. Each variable should appear at most once in the update set.*
- $U \subseteq E$ *identifies the subset of urgent edges.*
- $CP : L \cup E \rightarrow \mathbb{R}^{\geq 0}$ *assigns to each location and edge an energy consumption*

The semantics of the XFG process is defined in terms of *timed structures*.

Definition 3. *A* timed structure *is a tuple* $\langle S, S_0, T\rangle$ *where*

- S *is a set of states,*
- $S_0 \subseteq S$ *is the subset of initial states, and*
- $T \subseteq S \times (\mathbb{R}^{\geq 0} \cup \{\mu\}) \times S$ *is a transition relation.*

A run *of a timed structure is an infinite sequence*

$$\pi = s_0 \xrightarrow{\lambda_0} s_1 \xrightarrow{\lambda_1} s_2 \ldots$$

where $s_0 \in S_0$ *is an initial state and* $\langle s_i, \lambda_i, s_{i+1}\rangle \in T$ *is a transition for all* $i \in \mathbb{N}$*.*

Timed structures distinguish two kinds of transitions: time-passing transitions are labeled by a non-negative real number that represents the amount of time that has elapsed during this transition. Discrete transitions model state changes and have a special label μ. To define the dynamic semantics of XFG, the following *evaluation function* is needed.

Definition 4. *We assume a universe Val of values that includes the set* $\mathbb{R}^{\geq 0}$ *of non-negative real numbers and the Boolean values tt and ff. A valuation is a mapping* $\rho : V \rightarrow Val$ *from variables to values such that* $\rho(c) \in \mathbb{R}^{\geq 0}$ *for all* $c \in V_c$. *For a valuation* ρ *and* $\delta \in \mathbb{R}^{\geq 0}$ *we write* $\rho[+\delta]$ *to denote the environment that increases each clock in* V_c *by* δ*:*

$$\rho[+\delta](v) = \begin{cases} \rho(v) + \delta & \text{if } v \in V_c \\ \rho(v) & \text{otherwise} \end{cases}$$

We assume given an evaluation function

$$[\![\_]\!]_{\_} : Expr \rightarrow (V \rightarrow Val) \rightarrow Val$$

that associates a value $[\![x]\!]_\rho$ *with any expression* $x \in Expr$ *and valuation* ρ. *We require that* $[\![x]\!]_\rho \in \{tt, ff\}$ *for all* $x \in Bexpr$.

Definition 5. *Operational semantics of an XFG process is given as a timed transition system* $\langle S, s_0, T \rangle$ *where*

- $S = \langle l, \rho \rangle \in L \times \rho[+\delta](v)$
- $s_0 = \langle l_0, \rho_0 \rangle$
- $T \subseteq S \times (E \cup \mathbb{R}^{\geq 0}) \times S$ *such that:*
  - *For any* $e = \langle l, g, h, u, l' \rangle \in E$ *and* $\{\langle l, \rho \rangle . \langle l \rho'[u] \rangle\} \subseteq S$*:* $\langle l, \rho \rangle \xrightarrow{e} \langle l', \rho'[u] \rangle$
  - *For any* $\delta \geq 0$ *and any* $\{\langle l, \rho \rangle, \langle l', \rho'[+\delta] \rangle\} \subseteq S$*:* $\langle l, \rho \rangle \xrightarrow{\delta} \langle l, \rho[+\delta] \rangle$
  - *To each such transition step, we associate an energy consumption defined by*

$$\begin{cases} CP(\langle l, \rho \rangle \xrightarrow{e} \langle l', \rho'[u] \rangle) = CP(e) \\ CP(\langle l, \rho \rangle \xrightarrow{\delta} \langle l, \rho[+\delta] \rangle) = CP(l) \cdot \rho[+\delta] \end{cases}$$

A run π of the XFG process is a finite of infinite sequence of steps with no time-stuttering. The energy consumption of π denoted $CP(\pi)$ is the accumulated consumption of steps along the run. An XFG system is a finite set of XFG processes. With any XFG we associate a timed structure, allowing continuous and discrete energy consumption, whose states are given by the active locations of the XFG and the valuations of the underlying variables. Detail syntax and semantics will be defined in the next section.

3.2 Complete syntax and semantics: XFG system

The aforementioned semantics gives a meaning to a global XFG system consisting of a set of XFG's single graphs (processes) together with a set of shared data variables and a st of communication channels between the individual XFG's. To define the communication channels, the concept of value passing expression is defined.

Definition 6. *Let* $H = \{h1, h2, \dots, \}$ *be a set of communication (synchronization action) labels, and let* $\overline{H} = \{\overline{h_1}, \overline{h_2}, \dots\}$ *denote a set of complementary labels. A* value passing expression *is a tuple* $\langle ch, ia, oa \rangle$ *where*

- $ch \in H \cup \overline{H}$ *identifies a communication channel,*
- $ia \in V_\tau$ *is a possible empty tuple of variables, and*

– $oa \in Expr_\tau$ *is a possible empty tuple value expressions over* V.

Let VP *denote the set of possible value passing expressions, and* VP_V *those that range over variable set* V. *Two communication labels are referred to as complementary, if one is an overlined version of the other, i.e.* h *and* $\overline{h}$ *are complementary.*

In concrete syntax a value passing expression $\langle\langle v_1, v_2, \ldots\rangle, \langle x_1, x_2, \ldots\rangle\rangle$ is written as $h?v_1?v_2, \ldots, !x_1!x_2$, where $v_1, v_2, \ldots$ denotes variables, and $x_1, x_2, \ldots$ denote value expressions. Mostly, value expressions only transfer single value or no value at all. In the latter case, they become pure synchronization. Value passing expressions come with a notion of direction, implemented by label names. Only value passing expressions with complementary label can be matched for actual communication.

Definition 7. *An XFG system is a tuple* $\mathscr{X} = \langle GV, GInit, G, Ch, GCP\rangle$, *where*

- $GV = GV_c \cup GV_{cont} \cup GV_{disc}$ *is a set of global variables, where each* $GV_c, GV_{cont}, GV_{disc}$ *is a set of global clock, continuous, and discrete variables,*
- $GInit$ *defines the initial condition of* $\mathscr{X}$,
- $G = \langle P_1, \ldots, P_n\rangle$ *is a tuple of* $\mathscr{X}$,
- $Ch : EE \to (VP \cup \{\bot\})$, *where* $EE = \bigcup_{P \in G}^{n} E$ *provided that* $Ch(e) \in \{\bot, VP_v\}$ *for each* $P \in G$ *and* $e \in E$,
- $GCP : LL \cup EE \to \mathbb{R}^{\geq 0}$, *where* LL *is a location vector, is a function mapping location vectors or* EE *to energy consumptions,*
- *For each* $e, e' \in EE$ *with* $Ch(e) = \langle l, \langle v_1, \ldots, v_n\rangle, \langle x_1, \ldots, x_m\rangle\rangle$ *and* $Ch(e') = \langle l', \langle v'_1, \ldots, v'_{n'}\rangle, \langle x'_1, \ldots, x'_{m'}\rangle\rangle$, *if* l *and* l' *are complementary then* $n = m'$ *and* $m = n'$ *and* $\forall\, i \in \{1, \ldots, n\}.\mathbb{T}_V[\![v_i]\!] = [\![x'_i]\!]$ *and* $\forall\, i \in \{1, \ldots, m\}.\mathbb{T}_V[\![v'_i]\!] = [\![x_i]\!]$ *where*
  - $\mathbb{T}_V[\![\_]\!] : (Expr \cup V) \to \mathscr{P}(Val)$ *is an evaluation function associates a type with each value expression and variable where* $\mathscr{P}(Val)$ *denotes the powerset of* Val,
  - *Types are interpreted as sets of possible values and we assume type correctness of value expressions:* $\forall x \in Expr.\forall \rho \in (V \to Val).[\![x]\!]_\rho \in \mathbb{T}_V[\![x]\!]$

Thus an XFG system is defined by a global state GV, a set G of single XFG's, and a function Ch assigning value passing expressions to some of the edges of the XFG processes. If $Ch(e) = \bot$ then no value passing is associated with e. The final constraint in the definition only serves to ensure that value expressions with matching labels have matching types. We assume that the identifiers used for locations and local variables are globally unique.

Let an XFG system $\mathscr{X}$, this $\mathscr{X}$ can be extended with an additional automaton that does not communicate with the XFG processes in $\mathscr{X}$. This simple form of extension is formalized below.

Definition 8. *Given an XFG system* $\mathscr{X} = \langle GV, GInit, \langle P_1, \ldots, P_n\rangle, Ch, GCP\rangle$, *and an XFG process* P, *the extension of* $\mathscr{X}$ *with* $P = \langle Dtype, Init, L, l_0, I, E, H, U, CP\rangle$ *is defined to result in the XFG system* $\mathscr{X}' = \langle GV, GInit, \langle P_1, \ldots, P_n, P\rangle, Ch', GCP\rangle$ *where*

$$Ch'(e) = \begin{cases} \bot & \textit{if } e \in E \\ Ch(e) & \textit{otherwise} \end{cases}$$

If $l = \langle l_1, \ldots, l_n \rangle$ *is a location of the global graph corresponding to* $\mathscr{X}$*, and* l' *is a location of P, then we let* $l + l'$ *denote the location* $\langle 1_1, \ldots, l_n, l' \rangle$ *of the global graph corresponding* $\mathscr{X}'$*.*

Definition 9. *Let* $vp_1 = \langle l, \langle v_1, \ldots, v_n \rangle, \langle x_1, \ldots, x_m \rangle \rangle \in VP_V$ *and let* $vp_2 = \langle l', \langle v'_1, \ldots, v'_{n'} \rangle, \langle x'_1, \ldots, x'_{m'} \rangle \rangle \in VP_{V'}$*. Then the function* $synch(vp_1, vp_2) \in (\mathscr{P}((V \cup V') \times Expr_{V \cup V'}) \cup \{\bot\})$ *is defined as follows:*

$$synch(vp_1, vp_2) = \begin{cases} \bigcup\limits_{i \in \{1,\ldots,n\}} \langle v_i, x'_i \rangle \cup \bigcup\limits_{i \in \{1,\ldots,m\}} \langle v'_i, x_i \rangle & \text{if } l \text{ and } l' \text{ are complementary} \\ \bot & \text{otherwise} \end{cases}$$

$synch(vp_1, vp_2)$ returns $\bot$ if vp_1 and vp_2 do not match, which is the case if the synchronization labels are not complementary. If the two value passing expressions match, then an update is produced that is the result of combining the two expressions. Note that in that case it follows from Definition 7, that $n = m'$ and $m = n'$.

The most common operator for composing hybrid and TA is parallel composition. There are no compatibility requirements for the parallel composition of XFG process (seen as automata): Any pair of XFG process can be composed by the parallel composition operator. The parallel composition operator synchronizes on all external actions that the arguments share and allows interleaving of any other actions (under the condition that they maintain the consistency of the other process). The external variables that are shared by the argument processes need to have the same values. The formal semantics of the operator is defined in a structured operational semantics style below.

Definition 10. *Given an XFG system* $\mathscr{X} = \langle GV, GInit, \langle P_1, \ldots, P_n \rangle, Ch, GCP \rangle$ *with a single XFG process* $P_i = \langle V_i, Init_i, L_i, l_{0i}, I_i, E_i, H_i, U_i, CP_i \rangle$*, the global graph corresponding to* $\mathscr{X}$ *is an XFG* $= \langle V, Init, L, l_0, I, E, H, U, CP \rangle$ *where*

- $V = \bigcup\limits_{i \in \{1,\ldots,n\}} V_i \cup GV$,
- $\forall v \in V.Init = \begin{cases} Init_i & \text{if } v \in V_i \quad i \in \{1, \ldots, n\} \\ GInit & \text{if } v \in GV \end{cases}$
- $L = \prod\limits_{i=1}^{n} L_i$
- $l_0 = \langle l_{10}, \ldots, l_{n0} \rangle$
- $\forall \langle 1_1, \ldots, 1_n \rangle \in L.I(\langle 1_1, \ldots, l_0 \rangle) = \bigwedge\limits_{i \in \{1,\ldots,l_n\}} I_i(l_i)$
- *E,H and U are defined as follows: For any* $i, j \in \{1, \ldots, n\}$*, and for any* $urg \in Bexpr$*,*

$$\left.\begin{array}{l} \exists e_1 = \langle 1_i, g, h, u, l'_1 \rangle \in E_i. \\ Ch(e_1) = \bot.H(e_1) = \bot \text{ and} \\ U(e_1) = urg \end{array}\right\} \Leftrightarrow \begin{cases} \exists e = \langle \langle 1_1, \ldots, l_n \rangle, g, h, u, \langle l'_1, \ldots, 1'_n \rangle \rangle \in E. \\ (\forall k \in (\{1, \ldots, n\} \backslash \{i\}) \Rightarrow l_k = l'_k) \text{ and} \\ H(e) = \bot \text{ and } U(e) = urg \end{cases}$$

$$\left.\begin{array}{l} \exists e_1 = \langle 1_i, g_1, h_1, u_1, l'_i \rangle \in E_i. \\ \exists e_2 = \langle 1_i, g_2, h_2, u_2, l'_j \rangle \in E_j. \\ synch(Ch(e_1), Ch(e_2)) \neq \bot. \\ H(e_1) \neq \bot \;.\; H(e_2) \neq \bot \text{ and} \\ U_1(e_1) \vee U_j(e_2) = urg \end{array}\right\} \Leftrightarrow \left\{\begin{array}{l} \exists e = \langle \langle 1_1, \ldots, l_n \rangle, g, h, u, \langle l'_1, \ldots, 1'_n \rangle \rangle \in E. \\ (\forall k \in (\{1, \ldots, n\} \backslash \{i, j\}) \Rightarrow l_k = l'_k) \text{ and} \\ g = g_1 \wedge g_2 \text{ and } h = h_1 \cup h_2 \text{ and} \\ u = u_1 \cup u_2 \cup synch(Ch(e_1), Ch(e_2)) \text{ and} \\ H(e) = H(e_1) \cup H(e_2) \text{ and } U(e) = urg \end{array}\right.$$

- $CP = \langle CP_1, \ldots, CP_n, GCP \rangle$

The definition of E, H, and U need additional explanation: An edge in the global graph (XFG system) originated either from one edge of one of the constituent graphs (processes) or from two matching edges from two different graphes (processes) as a consequence of synchronization. In the first case, the original edge must not have a value passing expression associated with it, since edges with a value passing expression are require to synchronize.

The resulting global edge is then given the guard, the synchronization (with an empty condition) and the urgency attribute from the local edge. In case the edge is the result of a synchronization, the two value passing expressions must have matched. Then the guard of the global edge is the conjunction of those of the local edges. The synchronization action label of the global edge is a combination of the synchronization action labels of the local edges. The update of the global edge is a combination of the updates of the local edges and the update that results from the synchronization. The global edge is urgent, if either one of the local edge is.

In case there is one sender-graph (process), which has an edge labeled with $h!v$, can synchronize with an arbitrary number of receiver-graphs (processes) having $h?v$, where h is a synchronization channel name and v is a share variable, then any receiver can synchronize in the current state must do so. If there are no receivers, then the sender can still execute the $l!$ action, i.e. sending is never blocking. This broadcasting type of synchronization is defined as follows.

Definition 11. *Assume an order $P_1, \ldots, P_n$ of processes given by the order of the processes in the XFG system $\mathscr{X}$. We have a transition $\langle l, l_1, \ldots, l_m, \rho \rangle \xrightarrow{\mu^*} \langle l', l'_1, \ldots, l'_m, \rho' \rangle$ (see Definition 12) if there is an edge $e = \langle l, l' \rangle$ and m edges $e_i = \langle l_i, l'_i \rangle$ for $1 \leq i \leq m$ such that*

- *$e, e_1, \ldots, e_m$ are in different processes,*
- *$e_1, \ldots, e_m$ are ordered according to the process ordering $P_1, \ldots, P_n$,*
- *e has a synchronization label $h! = \{h!x | \{x\} \subseteq Expr, h \in H\}$ and $e_1, \ldots, e_m$ have synchronization labels $h? = \{h?v | \{v\} \subseteq V\}$, where h is a broadcasting channel,*
- *ρ satisfied the guards of $e, e_1, \ldots, e_m$,*
- *For all location $l \in \langle l, l_1, \ldots, l_m \rangle$ not a source of one of the edges $e, e_1, \ldots, e_n$, all edges from l either do not have a synchronization label $h?$ or ρ does not satisfy the guard on the edge,*
- *ρ' is obtained from ρ by first executing the updated label given on e and then the updated labels given in e_i for increasing order of i,*
- *ρ' satisfies $I(\langle l', l'_1, \ldots, 1'_m \rangle)$*

In the following we define the operational semantics of the XFG system consisting of a set of XFG processes.

Definition 12. *Let $\mathscr{X}$ be an XFG with processes $P_1, \dots, P_n$. The timed structure $\mathscr{T} = \langle S, S_0, T \rangle$ generated by $\mathscr{X}$ is the transition structure such that*

- *S_0 consists of all tuples $\langle l_{1,0}, \dots, l_{n,0}, \rho \rangle$ where $l_{i,0}$ is the initial location of process P_i and $[\![Init_i]\!]_\rho = tt$ for the initial conditions $Init_i$ of all processes P_i.*
- *For any state $s = \langle l_1, \dots, l_n, \rho \rangle \in S$, any $i \in \{1, \dots, n\}$, and any edge $\langle l_i, g, h, u, l'_i \rangle \in E_i$ of process P_i such that $[\![g]\!]_\rho = tt$, $\mathscr{T}$ contains a transition $\langle s, \mu^*, s' \rangle \in T$ where $s' = \langle l'_1, \dots, l'_n, \rho' \rangle$ and $l'_j = l_j$ for $j \neq i$, and where*

$$\rho'(v) = \begin{cases} [\![e]\!]_\rho & \text{if } \langle v, e \rangle \in u \\ \rho(v) & \text{otherwise} \end{cases}$$

  provided that $[\![I(l'_j)]\!]_{\rho'} = tt$ for all $j \in \{1, \dots, n\}$. A set of pairs $\langle v, e \rangle$ is an assignment where v is a variable and e is an expression whose value is to be assigned to the variable.
- *For a state $s = \langle l_1, \dots, l_n, \rho \rangle \in S$ and $\delta \in \mathbb{R}^{\geq 0}$, $\mathscr{T}$ contains a transition $\langle s, \delta, s' \rangle \in T$ where $s' = \langle l_1, \dots, l_n, \rho[+\delta] \rangle$ provided that for all $0 \leq \varepsilon \leq \delta$, the location invariants evaluate to true, i.e. $[\![I(l_i)]\!]_{\rho[+\varepsilon]} = tt$, and that for all $0 \leq \varepsilon < \delta$, the guards of any urgent edge $\langle l_i, g, h, u, l'_i \rangle$ leaving an active location l_i of state s evaluate to false, i.e. $[\![g]\!]_{\rho[+\varepsilon]} = ff$.*
- *To each such transition step, an energy consumption is associated by*

$$\begin{cases} GCP(\langle l_0, \dots, l_n, \rho \rangle \xrightarrow{\mu^*} \langle l'_1, \dots, l'_n, \rho'[u] \rangle) = GCP(\mu^*) \\ GCP(\langle l_0, \dots, l_n, \rho \rangle \xrightarrow{\delta} \langle l_0, \dots, l_n, \rho[+\varepsilon] \rangle) = GCP(\langle l_0, \dots, l_n \rangle) \cdot \rho[+\varepsilon] \end{cases}$$

Discrete transitions correspond to edges of one of the XFG processes. They require the guard of the edge to evaluate to true in the source state. The destination state is obtained by activating the target location of the edge and by applying the updates associated with the edge. Time-passing transitions uniformly update all clock variables; time is not allowed to elapse beyond any value that activates some urgent edge of an XFG process. In either case, the invariants of all active locations have to be maintained.

A run of $\mathscr{X}$ is a path in the underlying transition system. Given a run $\pi = s_0 \xrightarrow{c^0} s_1 \xrightarrow{c^1} s_2 \dots \xrightarrow{c^{n-1}} s_n$, its ith-energy consumption is $GCP_i(\pi) = \sum_{j=0}^{n-1} c_i^j$. A position along a run π is an occurrence of a state $\langle l_0, \dots, l_n, \rho \rangle$ along π. Let Δ be such a position, then $\pi[\Delta]$ denotes the corresponding state, whereas $\pi \leq \Delta$ denotes the finite prefix of π ending at position Δ.

3.3 Property Specification Language

For the definition of the property specification language, a mapping which relates the compositional control state to the current global state. Let XFG_{Proc}_ID be a set of identifiers of XFG processes in the XFG system (also named XFG_ID), XFG_LocID be a set of identifiers of locations of XFG processes. We define the mapping $Allocation : LocID \rightarrow (XFG\_ID \rightarrow LocID)$ that holds for each global location a mapping from XFG processes to (local) locations. For location expressions, some new structures are needed.

Definition 13. *Given an XFG system* $\mathscr{X} = \langle GV, GInit, \langle Proc_1, \ldots, Proc_n\rangle, Ch, GCP\rangle$ *with a single XFG process* $Proc_i = \langle V_i,\ Init_i,\ L_i,\ l_{0i},\ I_i,\ E_i,\ H_i,\ U_i,\ CP_i\rangle$*. A location expression of* $\mathscr{X}$ *is a construct* $Proc_i\#loc$*, where* $Proc_i \in \{Proc_1, \ldots, Proc_n\}$ *and* $loc \in L_i$*. Let* $LE_{\mathscr{X}}$ *be the set of possible expressions for* $\mathscr{X}$*. Given an* $\mathscr{X}$*, we define the evaluation function for location expressions* $\mathscr{L}_{\mathscr{X}}[\![\_]\!] : LE_{\mathscr{X}} \to (\prod_{i=1}^{n} L_i \to B)$ *as follows:*

$$\mathscr{L}_{\mathscr{X}}[\![Proc\#loc]\!]\ (\langle l_0, \ldots, l_n\rangle) = \begin{cases} true & \text{if } \exists i \in \{1, \ldots, n\}.Proc_i = Proc \text{ and } l_i = loc \\ false & \text{otherwise} \end{cases}$$

A location expression evaluates if a single XFG process in an XFG system control is at some specified location. Thus, *Proc*#*loc* will be true if the single XFG process *Proc* is at location *loc*.

Definition 14. *Let* $\mathscr{X}$ *be an XFG system, i.e., a global XFG system, consists of a set of XFG processes. Then a property of XFG specification, called XFG property, is a tuple* $\langle\ V_P,\ \phi\rangle$*, where* V_P *is a set of property variables, having a subset* $V_{Pc} \subseteq V_P$ *of clock variables and a subset* $V_{Pp} \subseteq (V_P \setminus V_{Pc})$ *of parameters, and* ϕ *is a CTL [6] formula augmented with either time or energy constraints. Valid XFG property CTL formulae are defined by the following syntax:*

$$\phi\ ::= a \mid Proc\#loc \mid Proc.v \mid \neg\phi \mid \phi \vee \phi \mid E\phi U_{\alpha\sim\beta}\phi \mid A\phi U_{\alpha\sim\beta}\phi \mid (v := e)\&\phi$$

where

- $a \in Bexpr_{(V_P \cup GV)}$ *is a Boolean value expression ranging over both variables from the system and those of the property,*
- *Proc*#*loc* $\in LE_{\mathscr{X}}$ *expresses a location of the XFG process identified by Proc, where* $Proc \in XFG\_ID$*, and* $loc \in XFG\_LocID$*,*
- *Proc.v expresses a variable of the XFG process in which Proc identifies a block graph from the system and v a variable in Proc where* $v \in V_P$*,*
- *A and E are the universal and existential quantifiers,* $U_{\alpha\sim\beta}$ *is the "until" temporal modality,*
- α *is either a clock variable* $c \in V_{Pc}$ *or an energy consumption function CP,*
- β *ranges over* $\mathbb{R}$*,*
- $\sim \in \{<, \leq, \geq, >\}$*,*
- $v \in V_P$ *and* $e \in Expr_{V_P \cup GV}$*, then* $v := e$ *denotes an update of a property variable. Note that any property specification variable* $v \in V_P$ *occurs in a Boolean value expression or in the right-hand side of an update, it has to be in the scope of a property update that binds v to some value.*

In XFG property CTL update operators have precedence over temporal operators and temporal operators have precedence over Boolean operators. In concrete syntax we write property updates defined y a single assignment as $(v := e)\&\phi$*, while non-singleton property updates are written as* $\{v_1 := e_1, v_2 := e_2\}\&\phi$

The transformation of derived CTL operators to core syntax is defined by the following rules:

- $EF\ \phi\ =\ true\ EU\ \phi$
- $AG\ \phi\ =\ \neg EF\ \neg\phi$
- $AF\ \phi\ =\ true\ AU\ \phi$
- $EG\ \phi\ =\ \neg AF\ \neg\phi$
- $AX\ \phi\ =\ \neg EX\ \neg\phi$
- $\phi_1 \wedge \phi_2 = (\neg\phi_1) \vee (\neg\phi_2)$
- $\phi_1 \Rightarrow \phi_2 = (\neg\phi_1) \vee \phi_2$

Given an XFG system $\mathscr{X}$, let the extension of $\mathscr{X}$ with V_{Pc} and V_{Pp} denote an advanced XFG system $\mathscr{X}'$, derived from $\mathscr{X}$ by adding V_{Pp} to the set of variables, and V_{Pc} to the set of clocks. An XFG system $\mathscr{X}$ satisfies a property specification $\langle V_P, \phi\rangle$ if the initial state s_0 of the transition system defined by the extension of $\mathscr{X}$ with V_P satisfies ϕ, written as $s_0 \models \phi$. The satisfaction relation is inductively defined in definition 15.

The definition below gives the semantics for XFG CTL property, which is similar to those found in literature and partly based on [6] and [5].

Definition 15. *Let $M = \langle S, S_0, T\rangle$ be a transition system generated by an XFG system $\mathscr{X}$. Let $\langle V_P, \phi\rangle$ be an XFG property specification, and let $\langle l,\rho\rangle \in S$ be a state from M. Then we write $\langle l,\rho\rangle \models \phi$ to denote that ϕ is satisfied at a state $\langle l,\rho\rangle$ of M. Then $\langle l,\rho\rangle \models \phi$ if and only if $\langle l,\rho,\theta\rangle \models \phi$, where θ is an arbitrary initial valuation for the property variables and the satisfaction relation $\langle l,\rho,\theta\rangle \models \phi$ is inductively defined as follows:*

$$
\begin{array}{lcl}
\langle l,\rho,\theta\rangle \models true & & \\
\langle l,\rho,\theta\rangle \models a & \Longleftrightarrow & [\![a]\!]_{(\rho,\theta)} = true \\
\langle l,\rho,\theta\rangle \models Proc\#loc & \Longleftrightarrow & L_{\mathscr{X}}[\![Proc\#loc]\!](l) = true \\
 & & (\equiv\ (Allocation(l)(Proc) = loc)) \\
\langle l,\rho,\theta\rangle \models Proc.v & \Longleftrightarrow & [\![Proc.v]\!]_{(\rho,\theta)} = true \\
\langle l,\rho,\theta\rangle \models \neg\phi & \Longleftrightarrow & not \langle l,\rho,\theta\rangle \models \phi \\
\langle l,\rho,\theta\rangle \models \phi_1 \vee \phi_2 & \Longleftrightarrow & \langle l,\rho,\theta\rangle \models \phi_1\ or\ \langle l,\rho,\theta\rangle \models \phi_2 \\
\langle l,\rho,\theta\rangle \models A\phi_1 U_{\alpha\sim\beta}\phi_2 & \Longleftrightarrow & for\ all\ runs\ \pi \in \prod Proc\ starting\ from\ \langle l,\rho,\theta\rangle, \\
 & & \pi \models \phi_1 U_{\alpha\sim\beta}\phi_2 \\
\langle l,\rho,\theta\rangle \models E\phi_1 U_{\alpha\sim\beta}\phi_2 & \Longleftrightarrow & for\ at\ least\ one\ run\ \pi \in \prod Proc\ starting \\
 & & from\ \langle l,\rho,\theta\rangle,\ \pi \models \phi_1 U_{\alpha\sim\beta}\phi_2 \\
\langle l,\rho,\theta\rangle \models (v := e)\&\phi & \Longleftrightarrow & \langle l,\rho',\theta'\rangle \models \phi\ with\ (\rho',\theta') = [\![v := e]\!]_{(\rho,\theta)} = true \\
 & & v{:=}e\ denotes\ a\ property\ update \\
\pi \models \phi_1 U_{\alpha\sim\beta}\phi_2 & \Longleftrightarrow & there\ exists\ \xi > 0\ position\ along\ \pi\ such\ that \\
 & & \pi[\xi] \models \phi_2\ with\ \theta[+\delta(\xi)], \\
 & & for\ all\ positions\ \xi' \geq 0\ before\ \xi\ on\ \pi, \\
 & & \pi[\xi'] \models \phi_1\ and\ \theta[+\delta(\xi')], \\
 & & and\ \alpha \sim \beta,\ where\ \alpha = CP[\pi \leq \xi]
\end{array}
$$

In order to be able to deal with the property variables on XFG CTL, an additional valuation θ is introduced for hold values for these variables. Having such a property variable valuation, the semantics of the update operator becomes evident. Given θ, $(v := e)\&\phi$ is satisfied in case ϕ is satisfied given an updated property valuation in which the update $v := e$ has been applied.

The semantics of $\phi_1 U \phi_2$ can be explained more: it has to ensure that the property specification clocks are increased with the amount of time that elapses. Whenever a subproperty (ϕ_1 or ϕ_2) is evaluated against a state along a path then θ has to be updated such that the clocks in θ reflect the time that has elapsed. This is described by adding $\delta(\xi)$.

4 Running example: Brake-by-Wire System

Our running example is a Brake-by-Wire (BBW) system, which is modeled in EAST-ADL [9] based on a use case provided by VOLVO using Papyrus UML [15] within the ATESST2 project [3]. Figure 2 depicts a simplified schematic view of the BBW system with Anti-lock Braking System (ABS) function, where no mechanical connection exists between the brake pedal and the actuators applied to the four wheels.

The BBW consists of seven components (seen as function blocks): the BBW is illustrated as `FunctionAnalysisArchitecture` with ≪`analysisFunctionType`≫ in Figure 2. Each construction of the BBW has ≪`analysisFunctionProtptype`≫. One component can communicate with the others through ports and connectors. Hereafter, the `FunctionAnalysisArchitecture` associated with ≪`analysis Function Type`≫ will be called F_T and the function blocks associated with ≪`analysis Function Protptype`≫ will be called F_P.

- **Brake Pedal Sensor** (`pSensor`) : The position of the brake pedal is measured by this sensor and information derived from it is the basis for computing the applied brake force.
- **Brake Calculator** (`bCal`): Based on each brake pedal position, a desired torque (force) command is sent to the Brake Controller, i.e., each pedal angle is converted to its corresponding torque and the desired global torque is calculated based on the received torque. Afterwards, the calculated global torque is transferred to the Brake Controller.
- **Brake Controller** (`bCtr`): This F_P computes the desired torque required for each wheel based on the value received from the Brake Calculator, and it sends the computed torque to the ABS at each wheel.
- **ABS** (`abs`) : This F_P controls the braking to prevent the locking of wheels to avoid skidding. It calculates ABS commands based on the referenced brake torque (from the Brake Controller) and inputs from Vehicle Speed Sensor and Wheel Speed Sensor.
- **Vehicle Speed Sensor** (`vSensor`): The speed of the vehicle is measured and transferred to the ABS
- **Wheel Speed Sensor** (`wSensor`): The speed of the wheel is measured and transferred to the ABS.
- **Brake Actuator** (`actuator`): This F_P performs the actual braking by applying the brake pad to the brake disc, i.e., brake force is translated into voltage.

Each behavior inside F_P (called intra-behavior), is visualized in an XFG process. Moreover, the interactions between F_{PS} through the ports and connectors (namely inter-behavior) are captured by synchronization actions between XFG processes in the XFG system (XFG global graph). In addition to modeling such executional intra- and inter-behaviors of F_Ps in XFGs, the application requirements properties comprised of timing

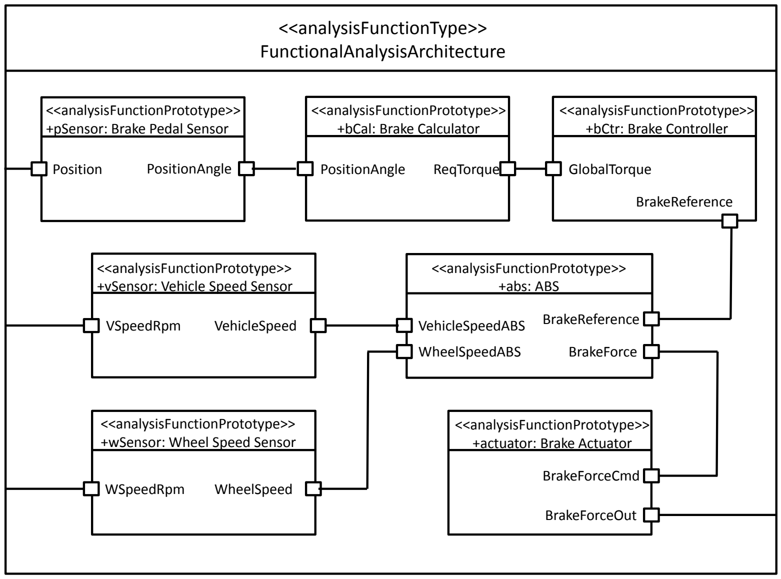

Fig. 2. Schematic view of the BBW system in EAST-ADL

and energy constrained modalities that are formally specified in the advanced XFG language.

For example, Figure 3 illustrates an XFG graphical representation, which simulates the intra-behavior of ABS F_P. The XFG textual implementation (XFG code) of ABS is denoted as `block graph ABS` in Listing 1.3 (lines 185 – 293). Urgent edge marked with a small round blob (line 289 with `prompt` keyword) on the `S5` location describes that no time unit is allowed on the synchronization action, in particular regarding the value passing through the synchronization channel (lines 289 – 292). Furthermore, the application requirements BBW system, in particular ABS, Brake Controller, and Actuator, must satisfy are specified (lines 19 – 79). The ABS has three modes of being:

1. *Receiving data* from Vehicle Speed Sensor, Wheel Speed Sensor, and Brake Controller processes through each synchronization channel `Vspeed_ABS?`, `Wspeed_ABS?`, and `BrakeCtr_ABS?`, which are defined in lines 199 – 200 respectively. A set of locations {`Idle`, `S1`, `S2`, `S3`} and edges between them which are associated with relevant channels model the "receiving data" behaviors. The required functions for receiving data are illustrated as `ABS_vehicle_speed_f()`, `ABS_wheel_spin_f()`, and `ABS_Brake_torque_f()` in Figure 3. They are defined in lines (217, 233), (224, 246), and (210, 263) respectively as assigning the received data to the local variables of `ABS`.
2. *Computing required commands* based on the received data from the three processes. A location `S4` and the incoming edges to the `S4` model "calculating slip value". The `ABS` controls the wheel braking in order to prevent locking the wheel, based on the slip value (a variable for this value is defined as continuous type in line 89). The slip

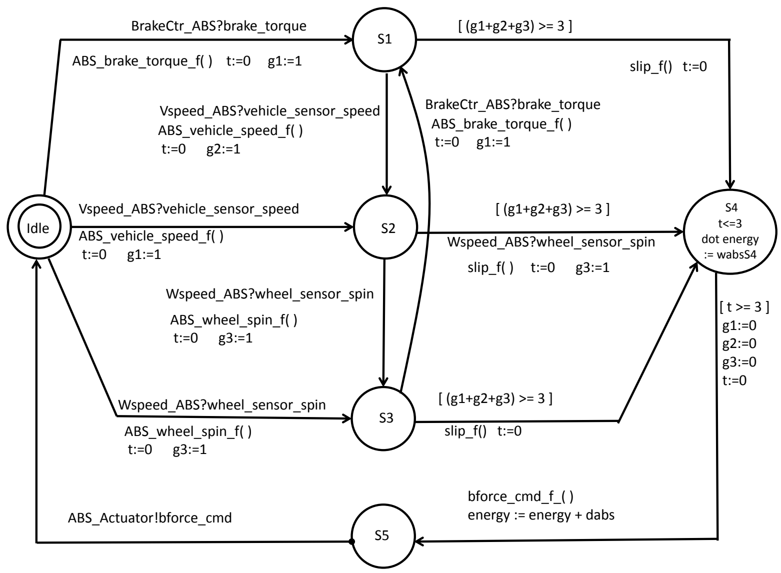

Fig. 3. XFG graphical representation of the ABS

value is calculated by the equation, $slip = (v - wr)/v$ where v is the vehicle speed, w is the wheel speed, and r is the wheel radius which are defined in lines 90 – 93. This equation is illustrated as a function `slip_f()` and defined straightforwardly in lines 238, 254, and 268. The friction coefficient of the wheel has a nonlinear relationship with *slip*:

– When *slip* increases from zero, the friction coefficient also increases and the value reaches the peak when *slip* is around 0.2. After that, further increase in *slip* reduces the friction coefficient. For this reason, if *slip* is greater than 0.2 the Brake Actuator is released and no brake is applied, otherwise the requested brake torque is used.

The required ABS commands for the Brake Actuator are controlled (computed) by the variable *slip*. Thus, from the location `S4`, based on the current *slip* value (`slip`), the ABS braking force command (`bforce_cmd`) is computed during a given clock constraint $t \leq 3$ (lines 273 – 286). The required function for this computation is given as `bforce_cmd_f()` in Figure 3 and defined in line 284. Furthermore, the `ABS` has two energy consumption types:

– Consumption of continuous energy (`dot energy`) expressed by its derivative (`wabs`) that gives the rate on the location `S4` where the ABS process consumes energy at the rate of `wabs` per one time unit (line 275 – 276).
– Consumption of discrete energy allocated on the edge from the location `S4` to the `S5` that is expressed as a usual update, e.g., `energy += dabs` where `dabs` is a discrete type integer (line 280 – 281).

3. *Sending out the computed commands* to the Brake Actuator via the synchronization channel `ABS_Actuator!`, which is defined in line 201. The location `S5` and its outgoing edge, which returns to the initial location `Idle`, model the ”sending data” behavior.

An interchange format XFG expressed in structured operational semantics for formal modeling and analysis of ERT system is introduced based on the hybrid and timed automata theory. The XFG language can provide a sound basis for modeling interdisciplinary (intra- and inter-block behaviors) semantics of systems in EAST-ADL in particular. In our early studies [14, 13], we first modeled the intra- and inter-behaviors of a system in EAST-ADL at the UML level then automatically translated the UML model into the XFG model by model transformation. In this way, developers can use familiar notations, while benefiting from formal specification and verification.

To enable an automatic and visualizing analysis of the XFG models derived from EAST-ADL models, we study how to obtain computer-aided analytical leverage through well established analysis tools. This study will provide a basis for automatic model transformations between EAST-ADL, XFG, and (Timed-Automata based) specification languages for model checkers. Our objective model checkers are `Uppaal` series tools, KRONOS and HYTECH. Details will be investigated in the following section.

```
% This is the Brake by Wire System
system BBW

%global constants, variables assignment are added here
define(radius, 10);   % define wheel radius
define(wabsS4, 3);    % define weighted energy of ABS
define(wgbc, 5);      % define weighted energy of Global Brake Controller
define(wact, 3);      % define weighted energy of Actuator
define(dabs, 2);      % define discrete energy of ABS
define(dgbc, 2);      % define discrete energy of GBC
define(dact, 2);      % define discrete energy of Actuator
define(rt, 100) % define end-to-end reaction time
define(sr, 10)             % define slip rate bound
define(mgbc, 100)     % define maximum energy assigned for GBC
define(mabs, 150)     % define maximum energy assigned for ABS
define(min, 500)      % define minimum energy consumed for the entire system BBW

% define property variables
property variables
    clock t, c, clk, tl;
    cont real get_torque, slip, bforce_cmd;
    cont real energy, cost_abs, cost_gbc;


% define properties (mainly related to ABS)
AG(abs#S5 imply ((clk:=0)&AF(clk < rt and actuator#S2)))
% End-To-End reaction time property between ABS and Actuator


AG((abs#S1 or abs#S2 or abs#S3) and abs.slip>sr)
   imply (actuator#S1 and actuator.bforce_cmd2==1))
% In case ABS slip rate exceeds its given bound (sr),
% the brake actuator is on the release mode and no brake is applied.


AG(abs#S5 imply (0<= abs.t and abs.t <=3)
% ABS local execution time property


AG((energy:=0)&EF(energy<=min and abs#S5))
```

```
% total accumulated energy of which the BBW consumes until
% it reaches to the ABS's location S5.
% it can be verified by uppaal-cora reachability analysis


AG(abs#S5 imply (abs.cost_abs <= mabs))
% ABS local energy consumption property


AG(abs#S4 imply ((cost_abs:=0)&EF(cost_abs <= mabs-dabs)))
% local energy consumption of which ABS is at the location S4


AG(Bctr#S1 imply (Bctr.cost_gbc <= mgbc))
% GBC local energy consumption property


%lead-to properties: location/data correspondence check
AG(abs.S1 or abs.S2 or abs.S3) imply AF(abs.S5))
% whenever the abs has received a signal from any of the GBC, wheel sensor,
% vehicle Sensor, it eventually sends the torque command to the actuator

AG(Bctr.Idle imply AF(Bctr.S2))
% whenever the GBC has received a signal from the pedal sensor,
% it eventually sends the computed torque to the ABS

AG(Bctr.brake_torque == 0 imply AF(Bctr.brake_torque != 0))
% each pedal angle data is eventually converted to the brake torque

AG(abs.slip == 0 imply AF(slip != 0))
% the slip rate is eventually computed based on
% the each brake torque received from the GBC

%data correspondence check
EF(abs.bforce_cmd == actuator.get_torque)
% the abs sends out a value of its torque command then
% the value should be received by the actuator

EF(Bctr.brake_torque == abs.abs_brake_torque)
% the GBC sends out a value of its brake torque then
% the value should be received by the ABS



state
    clock time:=0 ;
    cont real energy:=0;

    cont real [0,20] slip ;
    cont real [1,41] wheel_spin ;
    cont real [1,41] wheel_sensor_spin ;
    cont real [1,121] vehicle_speed ;
    cont real [1,121] vehicle_sensor_speed ;
    cont real [1,30] bforce_cmd ;
    cont real [1,46] pedal_pos ;
    cont real [1,46] pedal_sensor_pos ;

    disc int [1,3] brake_torque ;
    disc int [0,3] bforce_cmd2 ;

% define processes here (function block)
processes
    Pedal_Sensor      Psensor;
    Brake_Calculator  Bcal;
    Brake_Controller  Bctr;
    WheelSpeed_Sensor   Wsensor;
    VehicleSpeed_Sensor    Vsensor;
    ABS  abs;
```

```
    Actuator   actuator;


% define process composition behavior
composition
  Psensor || Bcal || Wsensor || Vsensor || abs || actuator || Bctr


% each function block process is defined here


% define Global Brake Controller process type here
block graph Bctr

% define local variable assignments
state
    clock tl:=0 ;
    cont real cost_gbc := 0;
    disc int request_torque := 0;
    disc int brake_torque :=0 ;

% all the input and output ports are defined here
ports
    in Psensor_BrakeCtr;
    out BrakeCtr_ABS;

% define initial state
init
    Idle

% define locations
locations Idle {
    when true
    synch Psensor_BrakeCtr?pedal_sensor_pos;
    do request_torque := pedal_sensor_pos;
       energy := energy+dgbc;
           %accumulated discrete energy consumption for the whole BBW system
       cost := cost+dgbc;
           %local discrete energy consumption for the GBC
       tl:=0;
    goto  S1
}

% invariant is defined if it is necessary
S1 inv (tl <= 5) {
    when not(tl>=3 && tl<=5)
      do dot energy := wgbc;
           %accumulated continuous energy consumption for the whole BBW system
         dot cost_gbc := wgbc;
           %local continuous energy consumption for the GBC
    goto  S1

    when  (tl>=3 && tl<=5 )
        do  out_torque()
         {
             if  (request_torque <=15 &&  request_torque >=0)
                 brake_torque := 1 ;

             if (request_torque <=30 && request_torque >=15)
                brake_torque := 2  ;

               if (request_torque <=45 && request_torque >=30)
                brake_torque := 3 ;
        }
            tl:=0;
    goto  S2
}
```

```
S2{
    when true prompt
    synch BrakeCtr_ABS!brake_torque;
    do cost_gbc := 0;
    goto Idle
}

% define ABS process type here
block graph ABS

state
    clock t:=0 ;
    disc int g1:=0;
    disc int g2:=0;
    disc int g3:=0;
    disc abs_brake_torque := 0;
    cont real cost_abs := 0;
    cont real abs_vehicle_speed := 0;
    cont real abs_wheel_spin := 0;
    cont real bforce_cmd : =0 ;
    cont real slip := 0;

ports
    in Vspeed_ABS, Wspeed_ABS, BrakeCtr_ABS;
    out ABS_Actuator;

init
    Idle

locations Idle {
    when true
    synch BrakeCtr_ABS?brake_torque;
        do  g1:=1;
            abs_brake_torque := brake_torque;
            t:=0;
    goto  S1

    when true
    synch Vspeed_ABS?vehicle_sensor_speed;
        do  g2:=1;
            abs_vehicle_speed := vehicle_sensor_speed;
            t:=0;
    goto  S2

    when true
    synch Wspeed_ABS?wheel_sensor_spin;
        do  g3:=1;
            abs_wheel_spin := wheel_sensor_spin;
            t:=0;
    goto  S3
}

S1 {
    when not (g1+g2+g3 >= 3)
    synch Vspeed_ABS?vehicle_sensor_speed;
         do g2:=1;
            abs_vehicle_speed := vehicle_sensor_speed;
            t:=0;
    goto  S2

    when (g1+g2+g3 >= 3)
        do  slip := (abs_vehicle_speed-abs_wheel_spin*radius)/abs_vehicle_speed;
            t:=0;
    goto S4
}

S2 {
    when  not (g1+g2+g3 >= 3)
```

```
    synch Wspeed_ABS?wheel_sensor_spin;
        do  g3:=1;
            abs_wheel_spin := wheel_sensor_spin;
            t:=0;
    goto  S3

     when (g1+g2+g3 >= 3)
     synch Wspeed_ABS?wheel_sensor_spin;
        do  g3:=1;
            slip := (abs_vehicle_speed-abs_wheel_spin*radius)/abs_vehicle_speed;
            t:=0;
    goto S4
}

S3 {
    when true
    synch BrakeCtr_ABS?brake_torque;
        do  g1:=1;
            abs_brake_torque := brake_torque;
            t:=0;
    goto  S1

    when (g1+g2+g3 >= 3)
        do  slip := (abs_vehicle_speed-abs_wheel_spin*radius)/abs_vehicle_speed;
            t:=0;
    goto S4
}

S4 inv (t <= 3) {
    when true
        do dot energy := wabs;
           dot cost_abs := wabs;
    goto S4

    when  not (t <= 3)
        do energy := energy + dabs;
           cost_abs := cost_abs + dabs;
           t:=0;
           g1:=0; g2:=0; g3:=0;
           bforce_cmd := slip * abs_brake_torque;
    goto S5
}

S5 {
    when true prompt
    synch ABS_Actuator!bforce_cmd;
         do cost_abs : =0;
    goto Idle
}

% Actuator process is defined
block graph Actuator

state
    clock c;
    cont real get_torque = 0;
    disc bfprce_cmd2 := 0;

ports
    in ABS_Actuator;
    out Actuator_Wdynamic;

init
    Idle

locations

Idle {
```

```
    when true
    synch ABS_Actuator?bforce_cmd;
        do c:=0;
           get_torque := bforce_cmd;
    goto S1
}

S1 inv (c <= 10) {
   when (c >=2 && c <= 10)
        do dot energy := wact;
           actuator_torque_f() {

                 if (get_torque >=31)
                    bforce_cmd2 := 1  ;

                 if (get_torque <=30 && get_torque >=15)
                    bforce_cmd2 := 2  ;

                 if (get_torque <=15 && get_torque >=1)
                    bforce_cmd2 := 3 ;
                    };
           c:=0;
   goto S2

   when not (c>=2 and c<=10)
        do energy := energy + dact;
   goto S1
}

S2{
    when true prompt
    synch Actuator_Wdynamic!bforce_cmd2;
    goto Idle
}
```

Listing 1.3. XFG Textual Specification

5 Verification

The ERT behaviors in EAST-ADL can be specified in XFGs based on *Behavior constraints* with the addition of resource-usage information. To enable computer-aided graphical modeling, formal analysis of the ERT behaviors, and even automatic code generation, the XFGs are represented in Uppaal-Cora PTAs[2] by *semantic anchoring*.

Application requirements on the system expressed in XFG CTL properties are represented as Uppaal-Cora CTL statements, and that can be verified by Uppaal-Cora. We have been developing conversion algorithms transform the all prototypes in XFG language to Uppaal-Cora process type declarations. In case an additional observation is required to dispatch an XFG process, the dispatcher XFG simply sends a triggering signal to the channel *trigger*. The transformation procedure also created the Uppaal-Cora trigger process that initializes the active objects of all processes types and issues triggers to them, as well as the application requirements are converted to the Uppaal-Cora CTL expressions.

The XFG system $\mathscr{X}$ consisting of those XFGs is considered as a network of PTA and expressed as a composition of the PTAs: We consider two types of resource consumption analysis:

[2] Uppaal-Cora is a branch of the Uppaal tool for cost optimal reachability analysis. For more details we refer the reader to www.uppaal.org

– *Feasibility analysis* to verify if the accumulated resource consumption on the actual behavior execution of F_P meets the available resource provided by the platform, and its corresponding formula:

  - $AF_{cp \leq min}\ Proc\#loc$

  which states that for all execution runs, the *loc* location in the XFG process is eventually reached within *min* resource-usage where cp is an energy consumption function;

– *Optimization of resource consumption analysis* to compute an optimal resource-aware run for the overall consumption of resources that formalized:

  - $EF_{cp \leq max}\ Proc\#loc$

  which denotes there is a run which the *loc* location in the XFG process is reached within a maximum resource-usage *max*

In our present experiment, we assume memory is a critical resource needs to be checked, and our particular concern is to analyze the optimal resource-consumption reachability problem for computing the minimum memory-usage on a corresponding run generated with the help of Uppaal-Cora, i.e., one can identify a sequence of event occurrences of BBW that costs the minimum memory. As an example, we find an optimal memory-usage sequence satisfying the property:

– $EF(ABS\#abs\_cal)$

which means that the torque command calculated based on the computed slip rate value is eventually sent to the actuator. This property is equivalent to the XFG property specification expressed in Listing 1.3 (line 40). The execution run is found by Uppaal-Cora is presented in Figure 5, and the best solution for its memory-usage (the lowest memory consumption) has been presented as 1870.

According to the safety concern, in addition to the optimal resource reachability analysis, the quality requirements are also formalized like the following which are established as valid over the BBW system network PTA model:

1. The brake pedal is activated, the actuator reacts timely under its given time bound (rt) as a failsafe. This is equivalent to checking the BP's F_P is invoked, it should not reach the $fail$ location of the observer XFG (*Obs*), which violates the bounded response time condition, while the actuator is executing.

– $AG(Psensor\#pedal \Rightarrow ((clk := 0) \& EF(clk \leq rt \wedge ACT\#active)))$
  $\equiv AG(Psensor\#pedal \Rightarrow (\neg Obs\#fail \wedge ACT\#active))$

  The *Obs* XFG contains an observer clock constraint as an invariant. This observer restricts the time bound of response time. By applying this observer XFG in our experiment, we successfully evaluate end-to-end response time properties (see line 26 in Listing 1.3) in a way that the fail location, which violates the bounded time condition, is never reached from any location of the main actual system model.

2. In case the ABS slip rate variable exceeds its given bound (*sr*), the brake actuator is on released mode (*rmode*) and no brake is applied ($bcmd == 1$), which is equivalent to the XFG property specification in Listing 1.3 (line 30 – 31)

   – $AG((ABS\#abs\_cal \land (slipRate > sr)) \Rightarrow (ACT\#rmode \land$ bcmd==1))

3. Execution time property: each F_P and its corresponding XFG should execute within its given local execution time, $lower \leq clock \leq upper$ (equivalent to the XFG property on line 36)

   – $AG(ABS\#cal \Rightarrow (lower \leq ABS.clock \leq upper))$

4. Lead-to property: whenever the ABS has received a signal from the GBC, it eventually sends the torque command to the actuator. Other lead-to properties (line 59 – 70 in Listing 1.3) are also successfully evaluated.

   – $AG(ABS\#get\_torque \Rightarrow AF(ABS\#abs\_cal)$

5. Data correspondent check: the ABS sends out a value of its torque command then, the value should be received by the actuator. (line 75 – 79 in Listing 1.3)

   – $EF(ABS.bforce\_cmd == ACT.get\_torque)$

Search order is breadth first and uses conservative space optimization for such safety properties checking. Verifying properties takes an average of around 2 seconds per verified property on an Intel T9600 2.80 GHz processor.

References

1. R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229. Springer-Verlag, 1993.
2. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. Advancing Traffic Efficiency and Safety through Software Technology Phase 2 (European project), 2010. http://www.atesst.org.
4. Gerd Berhmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing uppaal over 15 years. *Software - Practice and Experience*, December 2010.
5. Thomas Brihaye and Cois Raskin. Model-checking for weighted timed automata. In *In Proceeding of FORMATS-FTRTFTŠ04, LNCS 3253*, pages 277–292. Springer, 2004.
6. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. The MIT Press, 2000.
7. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III*, volume 1066 of *LNCS*, page ... Springer-Verlag, 1996.
8. C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proceedings of the 7th International Conference on Formal Description Techniques*, pages 227–242. Chapman and Hall, 1995.

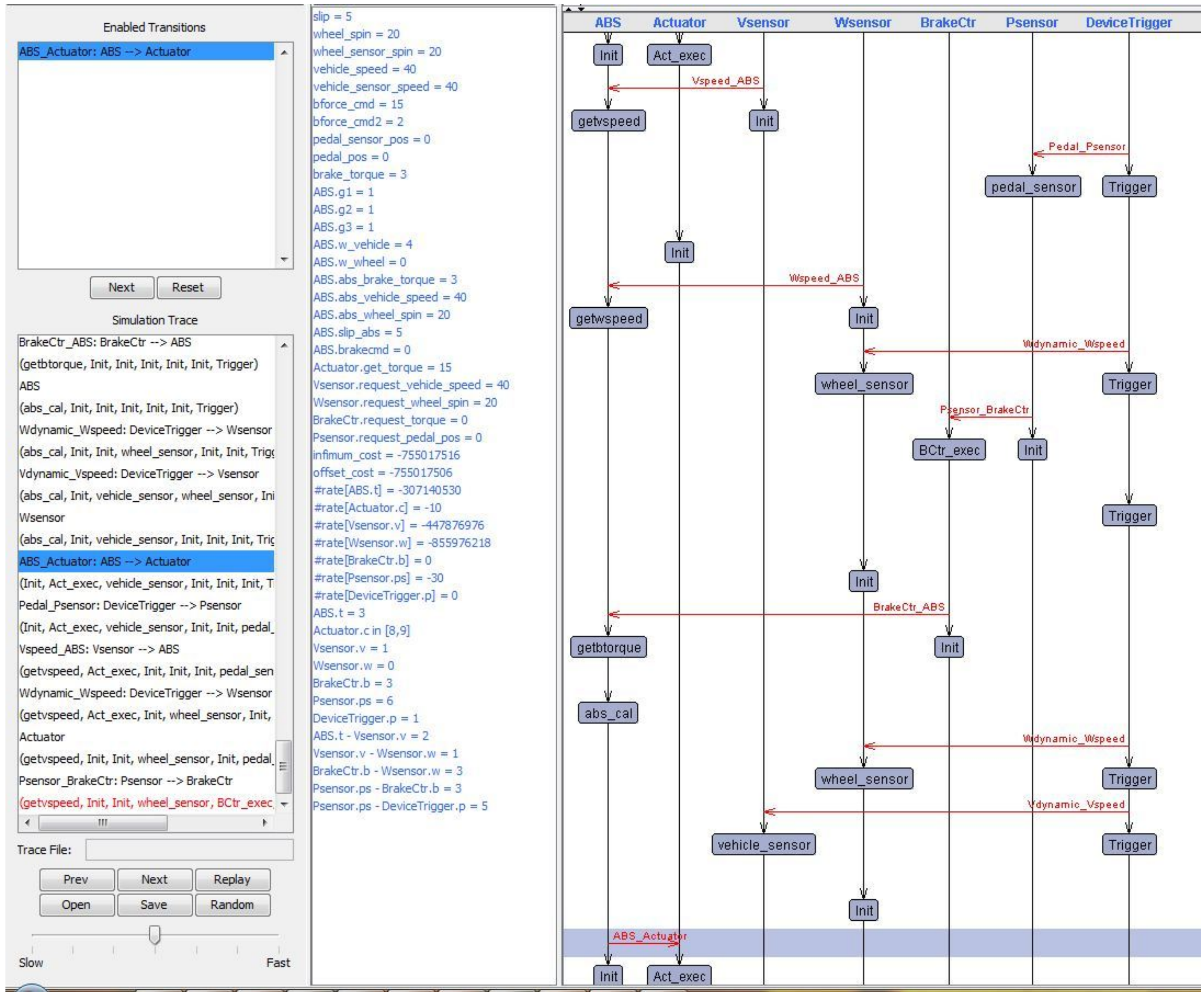

Fig. 4. Optimal sequence of event occurrences in BBW

9. EAST-ADL Consortium. East-adl domain model specification v2.1.9. Technical report, Maenad European Project, 2011.
10. T.A. Henzinger and P.-H. Ho. algorithmic analysis on nonlinear hybrid systems. In *Proceedings 7th International Conference on Computer Aided Verification, CAV'95*, volume 939 of *LNCS*, pages 225–238. Springer-Verlag, 1995.
11. T.A. Henzinger and P.-H. Ho. HyTech: The cornell hybrid technology tool. volume 1019 of *LNCS*, pages 29–43. Springer-Verlag, 1995.
12. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to hytech, 1996.
13. Eun-Young Kang, Gilles Perrouin, and Pierre-Yves Schobbens. Towards formal energy and time aware behaviors in east-adl: An mde approach. In *QSIC*, pages 124–127, 2012.
14. Eun-Young Kang, Gilles Perrouin, and Pierre-Yves Schobbens. XFG language and its profile for modeling and analysis of energy-aware and real-timed behaviors. Technical report, PReCISE Research Centre, Belgium, 2012.
15. Open Source Tool for Graphical UML2 Modeling, 2010. http://www.papyrusuml.org.
16. UPPAAL CORA, 2012. http://people.cs.aau.dk/∼adavid/cora/language.html.