XFG Language and its Profile for Modeling and Analysis of Energy-Aware Real-Time Behaviors

Kang, Eun-Young; Perrouin, Gilles; Schobbens, Pierre

Publication date:
2012

Document Version
Early version, also known as pre-print

XFG Language and its Profile for Modeling and Analysis of Energy-aware and real-timed behaviors

Eun-Young Kang, Gilles Perrouin, and Pierre-Yves Schobbens

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

Abstract. This report introduces a formal specification language XFG (extended function-block graphs), which can be used as IF (interchange format) for Timed Automata-based input modeling languages and model checkers. Section 1 informally represents a general introduction to XFG. The concrete E-BNF syntax rules 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. We propose a model-based approach to system engineering using XFG in Section 5 . Section 6 defines a UML profile for XFG language based on EAST-ADL and MARTE. Finally, Section 7 provides model-to-text transformations to convert the profiled models into XFG language.

1 eXtended Function-block Graphs: XFG

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 [6, 21], KRONOS [8, 7], and HYTECH [12, 11, 13], 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.

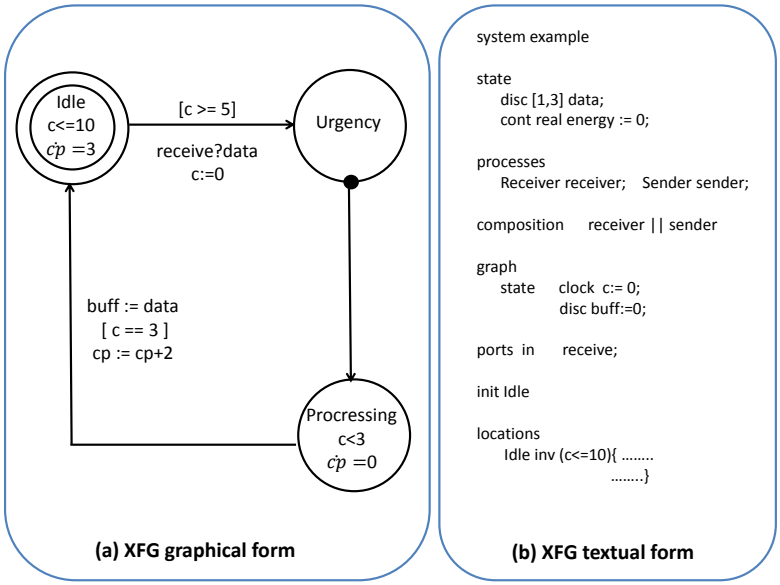


Fig. 1. A combination of XFG attributes: A receiver component as an XFG

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.

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.

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

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 $\dot{c}p$ 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 $\dot{c}p$ 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 ($\dot{c}p = 3$), 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`.

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

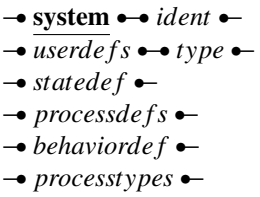
The terminal symbols present the ascii representation of the keywords of the XFG language. Each rule is labeled with its defining nonterminal symbol.

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*

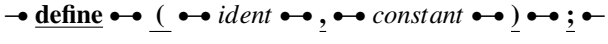


  - the system heading part, formed by the keyword **system** and a unique identifier.
  - *userdefs* : the user defined types.
  - *statedef* : the global state definition and all global variables.
  - *processdefs* : the process definitions, defining the processes in the system together with their types.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

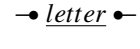
2.2 User Definition

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

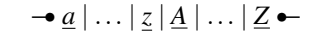
- *userdefs*



- *ident*



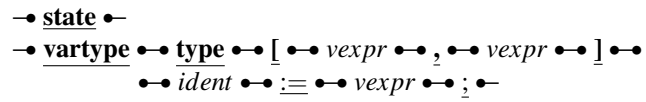
- *letter*



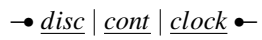
2.3 State Definition

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

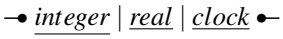
- *statedef*



- *vartype*

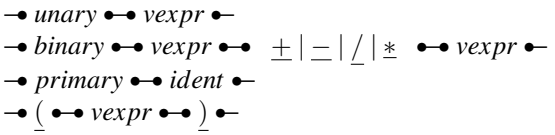


- *type*



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:

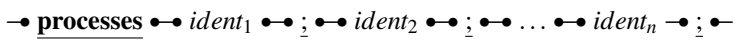
- *vexpr*



2.4 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*



2.5 Behavior Definition

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

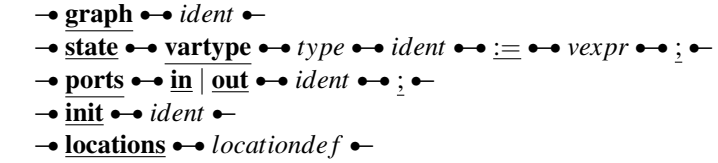
- *behaviordef*

2.6 Process Type

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

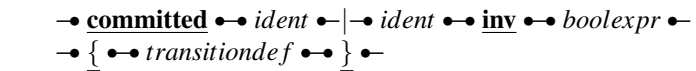
- a name.
- a state definition, defining the local variables. Variables have to be initialized.
- a set of communication ports. For each port, a direction is specified *in* or *out*, and its type (possibly empty).
- an initial location.
- a set of location definitions.

- *processtypes*

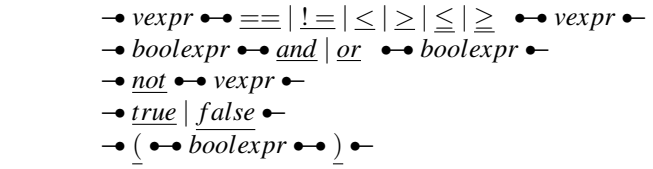


A *location* is defined by its name and a set of outgoing transitions. An invariant can be specified for a state, limiting the allowed data values for this location. Furthermore, a location can be declared committed, meaning that it has to be left immediately upon entering, without any other transitions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

- *locationdef*

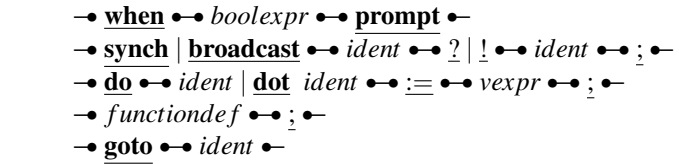


- *boolexpr*



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*



- *functiondef*

$$\multimap \textit{ident} \bullet\!\!-\!\!\bullet \underline{()} \bullet\!\!-\!\!\bullet \underline{\{} \bullet\!\!-$$
$$\multimap \underline{\textbf{if}} \bullet\!\!-\!\!\bullet \underline{(} \bullet\!\!-\!\!\bullet \textit{boolexpr} \bullet\!\!-\!\!\bullet \underline{)} \bullet\!\!-$$
$$\multimap \textit{ident} \bullet\!\!-\!\!\bullet \underline{:=} \bullet\!\!-\!\!\bullet \textit{vexpr} \bullet\!\!-\!\!\bullet \underline{;} \bullet\!\!-$$
$$\multimap \underline{\}} \bullet\!\!-$$

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*

- $Dtype : V \to \{disc, cont, clock\}$ *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,*
- $I : L \to Bexpr$ *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.*
– $CP : L \cup E \to \mathbb{R}^{\geq 0}$ *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* $\rho : V \to Val$ *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

$$[\![\_]\!]_\_ : Expr \to (V \to Val) \to Val$$

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* $H = \{h1, h2, \ldots,\}$ *be a set of communication (synchronization action) labels, and let* $\overline{H} = \{\overline{h_1}, \overline{h_2}, \ldots\}$ *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*

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

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:*

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

$synch(vp_1, vp_2)$ 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*

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

$$\left.\begin{array}{l} \exists e_1 = \langle 1_i, g, h, u, l_1' \rangle \in E_i. \\ Ch(e_1) = \bot.H(e_1) = \bot \text{ and} \\ U(e_1) = urg \end{array}\right\} \Leftrightarrow \left\{\begin{array}{l} \exists e = \langle \langle 1_1, \ldots, l_n \rangle, g, h, u, \langle l_1', \ldots, 1_n' \rangle \rangle \in E. \\ (\forall k \in (\{1,\ldots,n\} \backslash \{i\}) \Rightarrow l_k = l_k') \text{ and} \\ H(e) = \bot \text{ and } U(e) = urg \end{array}\right.$$

$$\left.\begin{array}{l} \exists e_1 = \langle 1_i, g_1, h_1, u_1, l_i' \rangle \in E_i. \\ \exists e_2 = \langle 1_i, g_2, h_2, u_2, l_j' \rangle \in E_j. \\ synch(Ch(e_1), Ch(e_2)) \neq \bot. \\ H(e_1) \neq \bot \,.\, H(e_2) \neq \bot \text{ and} \\ U_1(e_1) \vee U_j(e_2) = urg \end{array}\right\} \Leftrightarrow \left\{\begin{array}{l} \exists e = \langle \langle 1_1, \ldots, l_n \rangle, g, h, u, \langle l_1', \ldots, 1_n' \rangle \rangle \in E. \\ (\forall k \in (\{1,\ldots,n\} \backslash \{i,j\}) \Rightarrow l_k = l_k') \text{ and} \\ g = g_1 \wedge g_2 \text{ and } h = h_1 \cup h_2 \text{ and} \\ u = u_1 \cup u_2 \cup synch(Ch(e_1), Ch(e_2)) \text{ and} \\ H(e) = H(e_1) \cup H(e_2) \text{ and } U(e) = urg \end{array}\right.$$

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

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 $I(\langle l',l'_1,\ldots,l'_m\rangle)$*

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 $P_1,\ldots,P_n$. 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 $\langle l_{1,0},\ldots,l_{n,0},\rho\rangle$ 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 $s=\langle l_1,\ldots,l_n,\rho\rangle\in S$, any $i\in\{1,\ldots,n\}$, 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 $s'=\langle l'_1,\ldots,l'_n,\rho'\rangle$ 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 $j\in\{1,\ldots,n\}$. 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 $s=\langle l_1,\ldots,l_n,\rho\rangle\in S$ and $\delta\in\mathbb{R}^{\geq 0}$, $\mathscr{T}$ contains a transition $\langle s,\delta,s'\rangle\in T$ where $s'=\langle l_1,\ldots,l_n,\rho[+\delta]\rangle$ 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*

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

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 $\pi = s_0 \xrightarrow{c^0} s_1 \xrightarrow{c^1} s_2 \ldots \xrightarrow{c^{n-1}} s_n$, 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 $\langle l_0,\ldots,l_n,\rho\rangle$ 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.

4 Running example: Brake-by-Wire System

Our running example is a Brake-by-Wire system (BWS), which is modeled in EAST-ADL [9] based on a use case provided by VOLVO using Papyrus UML [18] within the ATESST2 project [4]. 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 BWS consists of seven components (seen as function blocks): the BWS is illustrated as `FunctionAnalysisArchitecture` with ≪`analysisFunctionType`≫ in Figure 2. Each construction of the BWS 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.

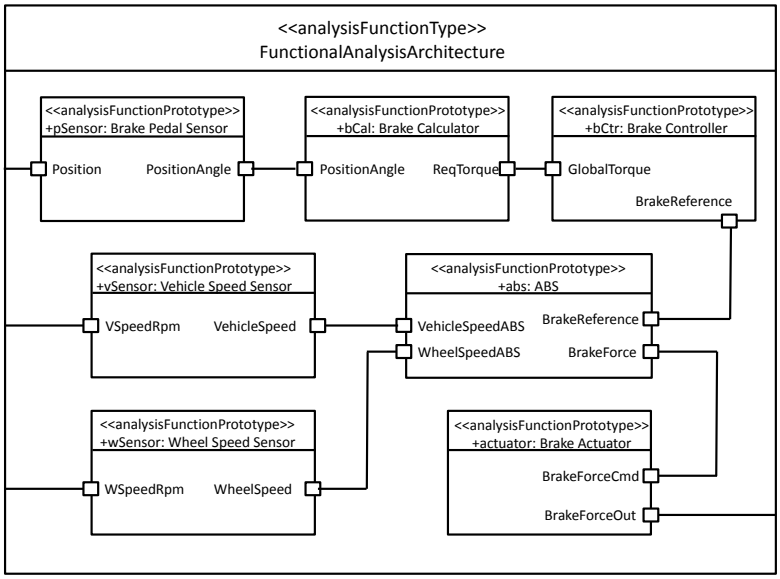


Fig. 2. Schematic view of the BBW system in EAST-ADL

- **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 and interactions between F_{PS} via ports and connectors (inter-behavior) are captured by synchronization actions between XFG processes in the XFG system (XFG global graph). For example, Figure 3 illustrates an XFG graphical representation, which simulates the intra-behavior of ABS F_P. The XFG textual format (XFG code) of ABS is denoted as `graph ABS` in Listing 1.2 (lines 46 – 153). Urgent edge marked with a small round blob (line 150 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 149 – 153). 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 59 – 60 respectively. A set of lo-

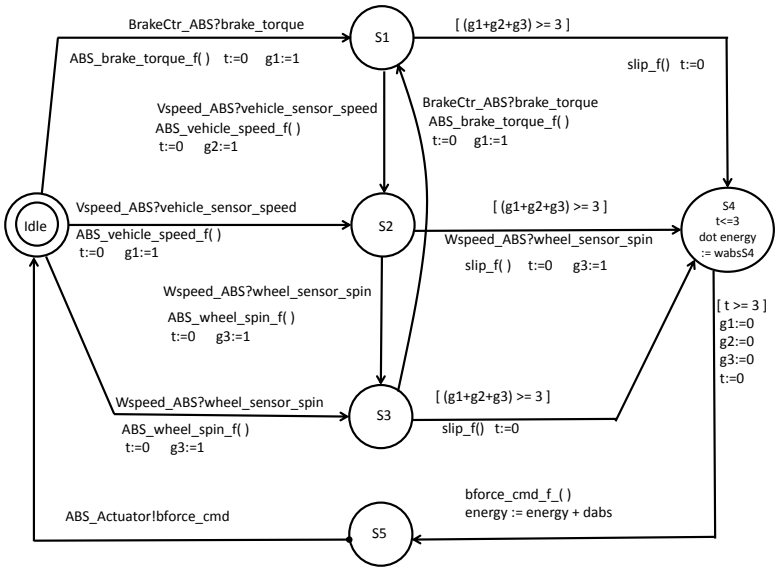


Fig. 3. XFG graphical representation of the ABS

cations {`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 (79, 95), (86, 109), and (72, 125) 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 16). The slip 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 17 – 20. This equation is illustrated as a function `slip_f()` and defined straightforwardly in lines 100, 116, and 130. 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 136 – 147). The required function for this computation is given as `bforce_cmd_f()` in Figure 3 and defined in line 145. Furthermore, the `ABS` has two energy consumption types:

- Consumption of continuous energy (`dot energy`) expressed by its derivative (`wabsS4`) that gives the rate on the location `S4` where the ABS process consumes energy at the rate of `wabsS4` per one time unit (line 138).
- 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 142).

3. *Sending out the computed commands* to the Brake Actuator via the synchronization channel `ABS_Actuator!`, which is defined in line 61. 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. In particular, the XFG language can provide a sound basis for modeling interdisciplinary (intra- and inter-block behaviors) semantics of systems in EAST-ADL. Nevertheless, this language is not widespread among engineers. For this reason, we will first model the intra- and inter-behaviors of systems in EAST-ADL at the UML level then automatically translate the UML model into the analyzable XFG model by model transformation. In this way, developers can use familiar notations, while benefiting from formal specification and verification. Details will be investigated in the following sections.

```
1  % This is the Brake by Wire System
2  system BWS
3
4  %global constants, variables assignment are added here
5  define(radius, 10);   % define wheel radius
6  define(wabsS4, 3);    % define weighted energy of ABS
7  define(wact, 3);      % define weighted energy of Actuator
8  define(dabs, 2);      % define discrete energy of ABS
9  define(dact, 2);      % define discrete energy of Actuator
10
11
12 state
13     clock time:=0 ;
14     cont real[0,20] energy:=0;
15
16     cont real [0,2] slip ;
17     cont real [1,41] wheel_spin ;
18     cont real [1,41] wheel_sensor_spin ;
19     cont real [1,121] vehicle_speed ;
20     cont real [1,121] vehicle_sensor_speed ;
21     cont real [1,30] bforce_cmd ;
22     cont real [1,46] pedal_pos ;
23     cont real [1,46] pedal_sensor_pos ;
24
25     disc int [1,3] brake_torque ;
26     disc int [1,3] bforce_cmd2 ;
27
28 % define processes here (function block)
29 processes
30     Pedal_Sensor      Psensor;
31     Brake_Calculator  Bcal;
32     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 ABS process type here
graph ABS

% define local variable assignments
state
    clock t:=0 ;
    disc int g1:=0;
    disc int g2:=0;
    disc int g3:=0;
    disc brake_torque := 0;
    cont real abs_vehicle_speed := 0;
    cont real abs_wheel_spin := 0;

% all the inout and output ports are defined here
ports
    in Vspeed_ABS, Wspeed_ABS, BrakeCtr_ABS;
    out ABS_Actuator;

% define initial state
init
    Idle

% define locations
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
}

% invariant is defined if it is necessary
S4 inv (t <= 3) {
    when true
        do dot energy := wabsS4;
    goto S4

    when  not (t <= 3)
        do energy := energy + 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;
    goto Idle
}

% Actuator process is defined

graph Actuator

state
    clock c;
    cont real get_torque = 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 &&  get_torque >=21)
                    bforce_cmd2 := 3  ;     //strong force

                 if (get_torque <=21 && get_torque >=11)
                    bforce_cmd2 := 2  ;  // medium force

                 if (get_torque <=11 && get_torque >=1)
                    bforce_cmd2 := 1 ; //weak force
                    };
           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 1.2. XFG Textual Specification

5 A Model-based Approach to System Engineering

5.1 Motivations

While the XFG language presented in the previous sections can be used directly to model ERT systems, it may not suit systems engineers who are used to more high-level and visual notations. Various system engineering notations have been provided these last years, often as extensions (*profiles*) of the Unified Modeling Language (UML) [20] such as SysML [16], MARTE [17] and EAST-ADL [15], the latter two being the basis of our modeling approach. Due to the fact that the UML became the *lingua franca* of modeling and is an OMG standard since 1997, several mature environment for specifying UML models and profiles have been developed and these environments are now increasingly adopted by system designers. Since UML does not own a formal semantics, such models needs to be translated into analyzable specifications such as XFG and then verified using a dedicated model-checker. This translation is often performed manually by verification experts. Yet, this process is tedious an error-prone as the size of the models increases. To combine the benefits of UML with formal verification we offer means to: *i)* model ERT systems using existing UML system engineering notations slightly extended (i.e. an UML profile for XFG) to describe ERT-specific concerns and

ii) an automated model-transformation producing XFG systems in their textual representation.

5.2 EAST-ADL

EAST-ADL (Electronics, Architecture and Software Technology – Architecture Description Language) [15] is an ADL dedicated to automotive electronic systems resulting from several European projects [3, 14]. The language provides support for architectural specifications at different abstraction levels: the highest abstraction level, *Vehicle (Feature) level*, characterizes a vehicle by means of its features and requirements. The *Analysis level*, where the Functional Analysis Architecture (FAA) is specified, decomposes the model into function units, called *AnalysisFunctions* (AF), that communicate through ports. The model at this level is used for the analysis of control requirements, timing constraints, data consistency between interfaces, hazard identification, etc. The *Design level* seems similar but *DesignFunctions* must correspond to the elements of the final implementation: hardware, operating system, middleware, network, software modules. At *Implementation level*, the component architecture is represented using the AUTOSAR standard [5]. EAST-ADL has extensions for environment modeling, requirements specification, timing (borrowed from TADL [19]), dependability and V & V at every abstraction level. This modular approach separates the definition of functional and non-functional aspects (timing, dependability, etc.).

If EAST-ADL provides excellent support to model the requirements and architecture of ERT systems at various levels of abstraction, dedicated behavioral means are more limited. Our XFG profile mainly extends the UML state machine construct to model precisely such behavior. The XFG profile is progressively detailed in Section 6 and summarized in Appendix A. The transformation of XFG profiled models is described in Section 7.

6 Modelling ERT Behaviors with the XFG profile, EAST-ADL and MARTE

6.1 Structural Models

We rely on the EAST-ADL specification and UML profile [15] to model the high-level architecture of the system. In the following, we focus on EAST-ADL analysis level but the modeling and transformation approach is easily extensible to other levels. In the EAST-ADL profile, analysis functions are modeled using « `AnalysisFunctionType` » . To refer to such function types within an architecture UML parts (EAST-ADL « `AnalysisFunctionPrototype` ») are used. These parts exhibit function types' ports (EAST-ADL « `FunctionFlowPort` »), which are connected to other function types with connectors (EAST-ADL « `FunctionConnector` »). To ease communication amongst parts, we require that each port is attached to only one connector. An excerpt of the architecture of the BWS system discussed in previous sections is presented Figure 4.

Each function type have a set of attributes (UML properties). To model properties' types, we make use of the MARTE [17] profile, that provides facilities to describe non-functional properties (subtypes of MARTE « `NFP_Type` »). MARTE is also used to define clocks. Since energy is treated differently from other variables, the XFG profile provides the stereotype « `XFGEnergy` » to identify such a variable and ease its specific

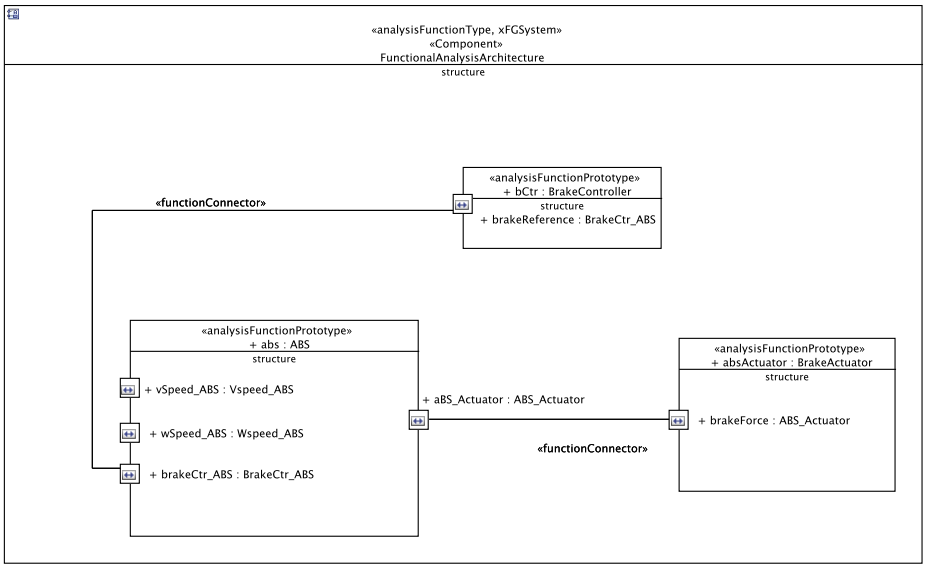


Fig. 4. Composite Structure Diagram of the BWS (excerpt)

translation. More generally, we will describe how these types are translated into XFG attributes in Section 7.

6.2 Behavioral Models

The XFG UML profile offers to model the behavior of automotive systems in terms of state machines. EAST-ADL behaviors can also be given in terms of activity diagrams [10] but UML state machines are semantically closer to XFG, based on timed automata. Nevertheless, we will make use of activities to send information through XFG channels.

The general design philosophy underlying XFG-profiled state machines is to mix the graphical notation of UML with XFG textual expressions to describe effects associated by the firing of transitions, time constraints or assignments. Thus, XFG plays the role of an ERT action language for UML state machines. Figure 5 depicts the state machine for the ABS function detailed in XFG in the preceding sections.

In the following we explain how to use each UML construct in the context of the XFG Profile, a summary of the profile and its intuitive semantics will be given in Appendix A.

Modelling States. Regarding state modeling, we focus on clock constraints and energy consumption. In our profiled state machines, states do not own any behavior (*Entry*, *Do* or *Exit* activities are thus forbidden) : behavior is associated to transitions as we will explain below. Clock constraints are modeled as XFG expressions as part of the states' invariants. The XFG UML profile provides a specific stereotype to model continuous energy consumption in states: « `XFGContEnergy` » . This stereotype allows the de-

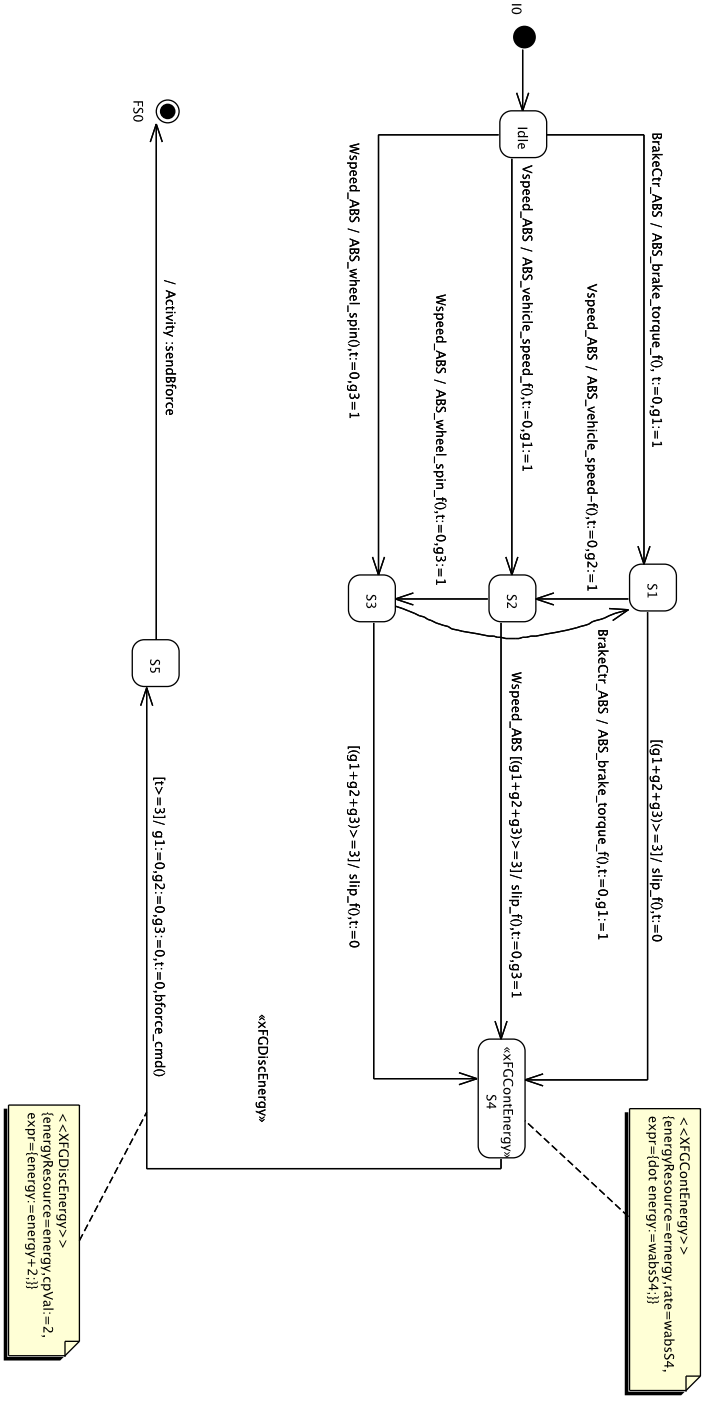


Fig. 5. State Machine for the ABS Component using the XFG Profile

signer to specify a consumption rate, the clock this consumption rate it applies to and the energy property to be updated. The application of such stereotype is depicted in the UML comment (a rectangle with a folded corner) associated to state S4 in Figure 5.

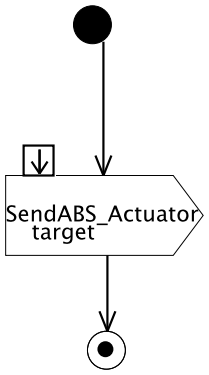


Fig. 6. Activity Diagram for Sending Brake Force to Actuator

Modelling Transitions The XFG UML profile models behavior over transitions. This behavior can concern the enclosing EAST-ADL function type only (*intra-behavior*) or ensure synchronization amongst function types of the considered system (*inter-behavior*):

- **Inter-behavioral information reception using triggers.** In UML triggers are events enabling a transition. In our profiled state machines, events for information reception are UML `ReceiveSignalEvent` where the associated signal carry a property modeling the information to be received. Each trigger is associated to a « `FunctionFlowPort` » port, allowing to identify from which function type the information is issued. This approach is equivalent to the reception of a value through a channel ($h?v$) in the XFG language,
- **Guards** Guards provide fine-grained control over the firing of the transition. If the guard is true the transition is enabled, disabled otherwise. In the context of the XFG profile we model guards using the XFG concrete (textual) syntax. These constraints can refer to local variables or clocks,
- **Intra-behavior effects** Effects correspond to behaviours realized when a triggering event has been received and the guard is true. We model intra-behavior as an UML `OpaqueBehaviour` having a textual XFG expression. The only difference is that we use comas to separate statements and that semicolons are omitted.
- **Inter-behavior effects using Activities.** For sending information to other function types, we use activities to model signal sending as illustrated on the transition going to the final state of the state machine Fig. 5. Such an activity diagram is depicted Figure 6. It shows how we send to the information to `BrakeActuator` using

a SendSignalAction. As for reception, the targeted function type is identified by specifying the EAST-ADL « FunctionFlowPort » port using the attribute onPort in the SendSignalAction element.

– **Discrete energy consumption** Behaviors realized on transitions consume energy. However this energy is discrete and is not dependent on time. As for continuous cases, we provide a stereotype: « XFGDiscEnergy » . This stereotype allows to specify a fixed amount of energy consumed by the transition as illustrated Fig. 5.

7 Transforming XFG UML models into XFG textual format

7.1 Model-to-Text Transformations with Acceleo MTL

Our approach relies on generative techniques to transform UML models using the XFG, MARTE and EAST-ADL profiles into the XFG format. Model-to-Text transformations (M2T) are an appropriate means to support it. We have chosen Acceleo[2], a free implementation of OMG's MOF Model to Text Language (MTL). Roughly, an MTL program consists of transformation rules (called *templates*), which are organized in *modules*. A template is usually formed both by immutable text and by expressions enclosed by square brackets. When applied on an actual model, these expressions are substituted by the result of their evaluation. The language offers usual constructs such as for, if and variable definition (let). Navigation amongst model elements is performed using the Object Constraint Language (OCL) syntax.

7.2 Mapping Structural Elements

Transforming EAST-ADL/MARTE Attributes. To describe attributes of EAST-ADL function types, we use the "annotated stereotype"[3] approach where predefined datatypes defined in MARTE and EAST-ADL UML profiles are applied at the model level on user-defined types. Since the XFG grammar only offers Integer and Real as possible types for variables we will focus on these ones. We envision to add built-in types in the XFG profile in the future. While the approach is similar to use MARTE or EAST-ADL predefined datatypes, MARTE provides a richer set of constructs to define and constrain datatypes. Such sophistication is currently not needed for primitive types, therefore, for the sake of simplicity we will exemplify our mapping using EAST-ADL « EAInteger » and « EFloat » stereotypes instead. Furthermore, we assume that discrete variables are always of type integer and continuous ones of type real. Listing 1.3 represent how XFG variables are handled. This basically consists in a conditional branching between properties stereotypes « EAFloat » and « EAInteger » , « Clock » properties and energy ones (stereotyped by « XFGEnergy » since energy has a special status in XFG). Lines 7-15 and 17 show calls to other templates which are detailed in the remaining of the listing.

```
1 [template public genVariables(props:Set(Property)) post(trim())]
2 % define local variable assignments
3 state
4 [for (prop:Property | props )]
5       [if (prop.type.getAppliedStereotypes()->notEmpty() and prop.type.
            getAppliedStereotypes()->select(st|
```

[2] http://www.eclipse.org/acceleo/

[3] See part 2 of the MARTE tutorial for examples: http://www.omg.org/omgmarte/Documents/tutorial/part2.pdf

```
6           st.oclAsType(Stereotype).name='EAInteger' or st.oclAsType(Stereotype).
                name='EAFloat')->notEmpty())]
7       [genEAVariable(prop).trim()/]
8     [else]
9     [if (prop.getAppliedStereotypes()->notEmpty() and prop.getAppliedStereotypes()
          ->select(st|
10          st.oclAsType(Stereotype).name='Clock')->notEmpty())]
11    [genClock(prop)/]
12    [/if]
13    [if (prop.type.getAppliedStereotypes()->notEmpty() and prop.type.
          getAppliedStereotypes()->select(st|
14          st.oclAsType(Stereotype).name='XFGEnergy')->notEmpty())]
15    [genEnergy(prop)/]
16    [/if]
17    [genVariable(prop)/]
18    [/if]
19  [/for]
20  [/template]
21
22
23  [template public genClock(clock : Property) post(trim())]
24    clock [clock.name/]:=[if (clock.default->isEmpty())]['0;'/][else][clock.default
          /][/if]
25  [/template]
26
27  [template public genEnergy(energy : Property) post(trim())]
28    cont [energy.name/]:=[if (energy.default->isEmpty())]['0;'/][else][energy.
          default/][/if]
29  [/template]
30
31
32  [template public genVariable(prop:Property) post(trim())]
33  [if (prop.type.name='Integer')]
34      disc int [prop.name/][if (prop.default->notEmpty())]:=[prop.default/];[/if]
35  [/if]
36  [/template]
37
38
39  [template public genEAVariable(prop:Property) post(trim())]
40  [if (prop.type.getAppliedStereotypes()->notEmpty() and prop.type.
       getAppliedStereotypes()->select(st| st.oclAsType(Stereotype).name='EAInteger'
       )->notEmpty())]
41    [let eaType: Stereotype = prop.type.getAppliedStereotypes()->select(st| st.
          oclAsType(Stereotype).name='EAInteger')->any(true)]
42    disc int  ['['/] [prop.type.getValue(eaType, 'min') /],[prop.type.getValue(
          eaType, 'max') /][']'/] [prop.name/] [if (prop.default->isEmpty())] [';'/]
          [else]:=[prop.default/]; [/if]
43    [/let]
44  [else]
45    [if (prop.type.getAppliedStereotypes()->notEmpty() and prop.type.
          getAppliedStereotypes()->select(st| st.oclAsType(Stereotype).name='EAFloat'
          )->notEmpty())]
46      [let eaType: Stereotype = prop.type.getAppliedStereotypes()->select(st| st.
            oclAsType(Stereotype).name='EAFloat')->any(true)]
47    cont real  ['['/] [prop.type.getValue(eaType, 'min') /] , [prop.type.getValue(
          eaType, 'max') /] [']'/] [prop.name/] [if (prop.default->isEmpty())] [';'/]
           [else]:=[prop.default/]; [/if]
48    [/let]
49    [/if]
50  [/if]
51  [/template]
```

Listing 1.3. Generating XFG Attributes

Acceleo templates `genClock()` and `genEnergy` are straightforward. They append to the immutable string 'clock' or 'cont' (since energy is a continuous variable) the name of the UML property and its default value or '0' if there is no specified de-

fault value. Applied on a model these templates generate variable declarations such as `clock t:=0;` or `cont energy:=0;`. `genEAVariable` (lines 39-51) is more complicated as it also retrieves the lower and upper bound values provided by the modeler while she applied the stereotype. Thus a typical output of such a template is a declaration like this one: `disc int [ 1,3] brake_torque :=0;`. Other properties are considered as integer variables and processed with the default template `genVariable`.

7.3 Translating EAST-ADL Architecture into an XFG System.

As depicted Figure 4, an EAST-ADL architecture is divided in parts (whose associated function types can be recursively decomposed). To identify in a convenient way the top-level component, which delimits the boundaries of the system, we have introduced the stereotype « `XFGSystem` » . In our case, EAST-ADL “FunctionalAnalysisArchitecture" plays this role. Therefore the transformation of the UML model into its equivalent XFG specification starts by looking for the model element applying « `XFGSystem` » .

```
1  [let sysComp : Component = model.allOwnedElements()->select(e:Element| e.
       oclIsTypeOf(Component) and e.getAppliedStereotypes()->select(st:Stereotype|st
       .name='XFGSystem')->notEmpty())->any(true).oclAsType(Component)]
2  [file (sysComp.name+'.xfg', false, 'UTF-8')]
3  [genSystem(sysComp)/]
4  [genVariables(sysComp.ownedAttribute->reject(att|att.oclIsTypeOf(Port) or att.type
       ->isEmpty()))/]
5  [genProcessDecl(sysComp)/]
6  [genCompositions(sysComp)/]
7
8  ....
9
10 [template public genSystem(sys:Component) post(trim())]
11 %System Declaration
12 System [sys.name/]
13 [/template]
14
15 [template public genProcessDecl(comp : Component) post(trim())]
16 %define processes here (function block)
17 processes
18 [let parts:OrderedSet(Property) = comp.ownedAttribute->select(att:Property|att.
       type->notEmpty() and att.getAppliedStereotypes()->select(st:Stereotype|st.
       name='AnalysisFunctionProtoType')->notEmpty())]
19 [for (p:Property|parts)]
20 [p.type.name/] [p.name/];
21 [/for]
22 [/let]
23 [/template]
24
25 [template public genCompositions(sysComp:Component) post(trim())]
26 % define process composition behavior
27 composition
28 [for (c:Connector|sysComp.ownedConnector)]
29 [for (cend:ConnectorEnd|c.end)]
30   [if (cend<>c.end->at(c.end->size()))]
31     [cend.partWithPort.name+'||'/] [else] [cend.partWithPort.name+';'/]
32 [/if]
33 [/for]
34 [/for]
35 [/template]
```

Listing 1.4. Generating XFG System Elements

Listing 1.4 illustrates the templates generating declarations at XFG system level. Lines 1-6 exhibits calls to the templates in the main module. Lines 1-2 describes how the top-level component is identified using « `XFGSystem` » stereotype and create a

file (whose extension is ".xfg") and named after the top-level component. Line 2 calls the template for generating system declaration (detailed lines 10-13): appending the name of top-level component to the `System` keyword. Then comes the call to variables generation (explained above), for which we do not consider UML ports or untyped UML properties. This followed by templates handling process declaration. To generate such process declaration (lines 15-23) we identify the UML parts stereotyped by EAST-ADL « `AnalysisFunctionPrototype` » . Their types, (EAST-ADL « `AnalysisFunctionTypes` ») will be translated to XFG process. We then append the name of the process (mapped from the type of the part considered), that will be detailed in within `Graph` declaration, followed by an identifier (the part name). Finally, lines 25-35 details how we generate parallel composition of processes by analyzing EAST-ADL « `FunctionConnectors` » connectors: for each of such connectors we extract the name of the correct function type attached to the ports linked by the connector.

7.4 Mapping Behavioral Elements

Now we have we have translated structural elements and XFG system declarations, we describe here how XFG processes are generated from UML state machines, as illustrated Listing 1.5.

```
1  [let atts : OrderedSet(Property) = sysComp.ownedAttribute->select(e:Property| e.
       type->notEmpty() and e.type.oclIsTypeOf(Component) and e.
       getAppliedStereotypes()->select(st:Stereotype|st.name='
       AnalysisFunctionPrototype')->notEmpty())]
2  [for (c: Component | atts.type.oclAsType(Component))]
3    [genGraph(c)/]
4    [genSyncDecl(sysComp,c)/]
5    [genVariables(c.ownedAttribute->reject(att|att.oclIsTypeOf(Port) or att.type->
         isEmpty()))/]
6    [for (st:StateMachine | c.ownedBehavior->select(b:Behavior|b.oclIsKindOf(
         StateMachine)))]
7      [generateInitialLocation(st)/]
8      [generateLocationsDef(st)/]
9    [/for]
10 [/for]
11 [/let]
12
13 [template public genGraph(comp:Component) post(trim())]
14
15 % [comp.name/] process is defined
16 graph [comp.name/]
17
18 [/template]
19
20 [template public genSyncDecl(sysComp:Component,c:Component) post(trim())]
21   % all the input and output ports are defined here
22   ports
23      [for (p:Port| c.ownedPort)]
24     [if (p.getValue(p.getAppliedStereotype('EAST-ADL2::Structure::FunctionModeling
           ::FunctionFlowPort'), 'direction')->notEmpty() and p.end->notEmpty())]
25     [let cName : String =  p.end->select(cend:ConnectorEnd|cend->notEmpty() and
           cend.owner->notEmpty())->any(true).owner.oclAsType(Connector).name]
26         [p.getValue(p.getAppliedStereotype('EAST-ADL2::Structure::FunctionModeling
               ::FunctionFlowPort'), 'direction')/] [if (cName->notEmpty())][cName
               /][/if];
27     [/let]
28     [/if]
29       [/for]
30 [/template]
31
32 [template public generateInitialLocation(st: StateMachine)]
```

```
33
34 [let initTrans : Transition = st.region->any(true).oclAsType(Region).transition->
35           select(t:Transition|t.source.oclIsKindOf(Pseudostate) and t.source.
                 oclAsType(Pseudostate).kind = PseudostateKind::initial)->any(true)]
36 % define initial location
37 init
38  [initTrans.target.name/]
39 [/let]
40 [/template]
41
42
43 [template public generateLocationsDef(st: StateMachine) post (trim())]
44 % locations definition
45 locations
46 [for (state:State| st.region->any(true).oclAsType(Region).subvertex->select(sta:
       Vertex|sta.oclIsTypeOf(State)).oclAsType(State))]
47       [genLocation(state)/]
48 [/for]
49 [for (term:Pseudostate| st.region->any(true).oclAsType(Region).subvertex->select(
       sta:Vertex|sta.oclIsKindOf(Pseudostate) and sta.oclAsType(Pseudostate).kind =
        PseudostateKind::terminate).oclAsType(Pseudostate))]
50       [generateTerminate(term)/]
51 [/for]
52 [let fst:FinalState = st.region->any(true).oclAsType(Region).subvertex->select(sta
       :Vertex|sta.oclIsKindOf(FinalState))->any(true).oclAsType(FinalState)]
53     [genFinalLocation(fst)/]
54 [/let]
55 [/template]
56
57
58 [template public generateTerminate(term : Pseudostate)]
59   [state.name/] {}
60 [/template]
```

Listing 1.5. Generating XFG Processes

Processes' channels and variables generation. The full definition of each process, starts with the `Graph` keyword as shown lines 13-18. Then comes the definition of synchronisation channels (in/out `ports` in XFG). Such ports serves for inter-behavior communication amongst processes and are translated from EAST-ADL ports and connectors. Lines 20-30 illustrates the generation algorithm: First, we consider only UML ports that are stereotyped by « `FunctionFlowPort` » for which a direction has been indicated and that are related to an UML connector (lines 23-24). Then, we append to the direction the name of the UML connector that is mapped to a channel in XFG. The generation of local process variables is the same as for "system" variables.

Profiled State Machines Translation. The core of the behavioral translation is illustrated in the scope of the `for` declaration line 6. The general approach is to translate states as locations and transition as `when ... do ... goto` blocks defined within locations. In the following, we describe how each element is translated according to the visual rules depicted Figure 7.

Generating Initial Location. As illustrated by visual rule R1, the initial state in XFG is mapped from the first state (and not the initial pseudostate in UML) of the state machine. This mapping rule is motivated by the fact that UML constrains the initial pseudostate (one unique transition, on which neither trigger nor guard can be added), which does not exist in XFG. Lines 32-40 of Listing 1.5 demonstrate how to identify

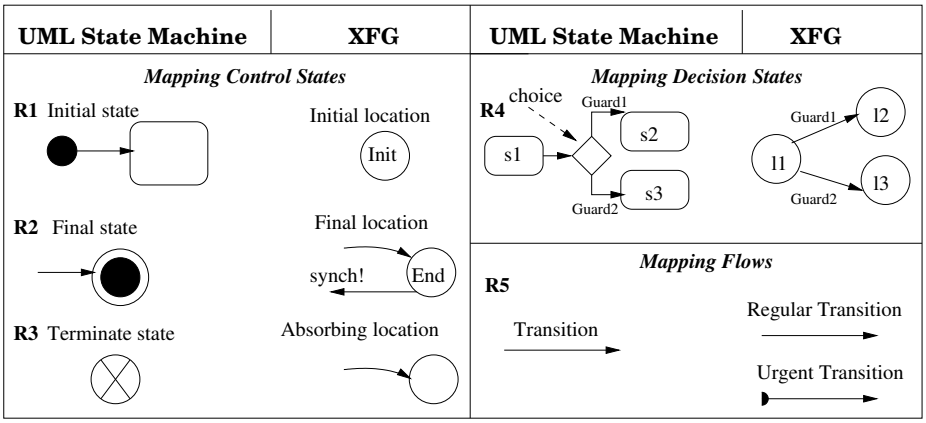


Fig. 7. State Machine Mapping Rules

the "start" state by identify the state whose incoming transition as the initial pseudostate as its source.

Generating final location and sending values through communication channels. The last transition of the UML state machine is related to inter-behavior synchronization of state machines. It consists of an effect modeled using an activity diagram as illustrated Figure 6. Mapping rule R2 illustrates conceptually how to transform this behavior in XFG and Listing 1.6 details the transformation in Acceleo. The identification of the final state in the UML state machine is performed lines 52-54 of Listing 1.5. From the final state, the `genFinalLocation` template generates a location and creates a non-guarded urgent transition (`when true prompt`) and call the template for generating sending value behavior defined lines 16-23. It proceeds by retrieving the UML `SendSignalAction` element (line 18) and identifying the correct communication channel (line 19), i.e. the UML EAST-ADL « `FunctionConnector` » connected to the « `FunctionFlowPort` » port referred by the `SendSignalAction` element. Once done, we append to the channel name the symbol for sending values `!` and the name of the variable contained by the signal associated to `SendSignalAction`. Finally, this final transition target the initial location of the XFG process as illustrated lines 8-10.

```
1  [template public genFinalLocation(fst:FinalState) post (trim())]
2  [fst.name/] {
3
4  when true prompt
5  [let synTrans : Transition = fst.incoming->select(t:Transition|t.effect.
       oclIsTypeOf(Activity))->any(true)]
6   synch [genSendSynch(synTrans)/]
7  [/let]
8  [let initTrans : Transition = fst.container.stateMachine.region->any(true).
       transition->
9            select(t:Transition|t.source.oclIsKindOf(Pseudostate) and t.source.
                 oclAsType(Pseudostate).kind = PseudostateKind::initial)->any(true)]
10 goto [initTrans.target.name/]
11 [/let]
```

```
12 }
13 [/template]
14
15
16 [template genSendSynch(trans:Transition) post (trim())]
17
18 [let sync: SendSignalAction = trans.effect.oclAsType(Activity).node->select(n:
       ActivityNode|n.oclIsTypeOf(SendSignalAction))->any(true) ]
19 [let syncChannel : String = sync.onPort.end->select(cend:ConnectorEnd|cend->
       notEmpty() and cend.owner->notEmpty())->any(true).owner.oclAsType(Connector).
       name ]
20 [syncChannel/]![sync.signal.attribute->any(true).name/]
21 [/let]
22 [/let]
23 [/template]
```

Listing 1.6. Generating Final Location and Sending Values to Other Processes

Generating Absorbing Locations. Regarding the generation of absorbing locations (Rule R3), the left-hand side of the rule is covered by the `for` statement lines 49-51 of Listing 1.5, which identifies the UML transition having `terminate` pseudostate as its target while the right-hand side of the rule, lines 55-57 illustrate the generation of this absorbing location.

Translating UML Choices. Mapping Rule R4 describes how a choice is translated to XFG. In fact, there is no direct translation of the concept UML choice in XFG, choice behavior is directly supported by transitions in XFG. Thus, we translate paths containing choices as regular XFG transitions where the source location is mapped from the source state of the choice. Since the source of such transition is not the same as for regular UML transitions, we need to treat as a special as case as demonstrated in lines 3-4 (excerpt of the template handling transitions) in Listing 1.7. Then `genChoice` template generates XFG transition with the particularity that they cannot receive values via communication channels ($h?v$) since we model value reception in UML using triggers on transition, which is not allowed for transition outgoing pseudostates (see [20] pp 582).

```
1  [template public genTransition(trans:Transition) post (trim())]
2
3  [if (trans.target.oclIsTypeOf(Pseudostate) and (trans.target.oclAsType(Pseudostate
       ).kind = PseudostateKind::choice) )]
4   [genChoice(trans.target.oclAsType(Pseudostate))/]
5
6   ...
7
8  [template public genChoice(choice:Pseudostate) post (trim())]
9
10 [for (trans:Transition | choice.outgoing)]
11 when [genGuard(trans.guard)/] [genPrompt(trans)/]
12 [if trans.effect->notEmpty()]
13 do [genEffect(trans.effect)/]
14 [/if]
15 goto [trans.target.name/]
16 [/for]
17 [/template]
```

Listing 1.7. Handling Choice

Translating UML State Machine Transitions Finally, mapping Rule R5 handles regular transitions. Full Acceleo code for managing transition is provided in Listing 1.8. Lines 1-4 have been explained above so we focus on lines 5-12:

– **Mapping signal reception to value reception from communication channels.** . Value reception is modeled in the trigger of the UML transition. Template `genReceiveSynch` (lines 14-18) generated the XFG statements for value reception in a similar way to what we have explained for sending values,
– **Guards.** UML Guards are straightforwardly mapped into XFG guards since the language for the UML constraint is already XFG. This is illustrated by template `genGuard` performs this translation,
– **Urgent transitions.** We have introduced the stereotype « `XFGUrgent` » to model the fact that a transition has to be taken as soon it is enabled. Template `genPrompt` appends the keyword `prompt` when the stereotype is applied,
– **From UML transition effects to XFG updates.** For transition effects not involving any value sending, we use UML `OpaqueExpression` consisting of XFG statements. Template `genEffect` (lines 37-48) describes how such opaque behaviors are translated: it basically consists in splitting the transition in text in XFg statements. For operation calls, such as `slip_f()`, we extract the associated XFG statements from the method associated to the operation defined in the owning « `FunctionType` » as illustrated by template `getOperation` lines 52-58,
– **Discrete energy consumption.** Regarding discrete energy consumption, we retrieve the expression `expr` which is defined in stereotype « `XFGDiscEnergy` » as detailed in template `genDiscEnergy` lines 61-67.

```
1  [template public genTransition(trans:Transition) post (trim())]
2  [if (trans.target.oclIsTypeOf(Pseudostate) and (trans.target.oclAsType(Pseudostate
       ).kind = PseudostateKind::choice) )]
3   [genChoice(trans.target.oclAsType(Pseudostate))/]
4  [else]
5  [if (trans.trigger->notEmpty())]
6  synch [genSynch(trans.trigger->any(true))/];
7  [/if]
8  when [genGuard(trans.guard)/] [genPrompt(trans)/]
9     [genEffect(trans.effect)/]
10 goto [trans.target.name/]
11 [/if]
12 [/template]
13
14 [template public genReceiveSynch(trigger : Trigger) post (trim())]
15 [let syncChannel : String = trigger.port->any(true).end->select(cend:ConnectorEnd|
       cend->notEmpty() and cend.owner->notEmpty())->any(true).owner.oclAsType(
       Connector).name ]
16   [syncChannel/]?[trigger.event.oclAsType(ReceiveSignalEvent).signal.attribute->
         any(true).name/];
17 [/let]
18 [/template]
19
20 [template public genGuard(guard : Constraint) post (trim())]
21   [if (guard->notEmpty() and guard.specification.oclIsTypeOf(OpaqueExpression))]
22       [guard.specification.oclAsType(OpaqueExpression)._body/]
23   [else]
24       true
25   [/if]
26 [/template]
27
28
29 [template public genPrompt(trans:Transition) post (trim())]
30 [if trans.getAppliedStereotypes()->notEmpty() and trans.getAppliedStereotypes()->
       select(st| st.oclAsType(Stereotype).name='XFGUrgent')->notEmpty()]
31   prompt
32 [/if]
```

```
33 [/template]
34
35
36
37 [template public genEffect(effect : Behavior) post(trim())]
38   [if (effect->notEmpty() and effect.oclIsTypeOf(OpaqueBehavior))] do
39     [let strs:Sequence(String) = effect.oclAsType(OpaqueBehavior)._body.tokenize('
           ,')]
40     [for (str:String | strs)]
41         [if (str.contains('()'))] [getOperationBody(str,effect)/]
42       [else]
43       [str.trim()/];
44       [/if]
45     [/for]
46     [/let]
47   [/if]
48 [/template]
49
50
51
52 [template public getOperationBody(str:String,b:Behavior) post(trim())]
53  [let opNam: String =  str.tokenize('(')->first().trim()]
54 [if (b.redefinitionContext.owner.oclAsType(Component).ownedOperation->select(op:
       Operation|op.name.equalsIgnoreCase(opNam))->notEmpty())]
55  [b.redefinitionContext.owner.oclAsType(Component).ownedOperation->select(op:
       Operation| op.name.equalsIgnoreCase(opNam))->any(true).method.oclAsType(
       OpaqueBehavior)._body/]
56 [/if]
57 [/let]
58 [/template]
59
60
61 [template public genDiscEnergy(trans:Transition) post(trim())]
62 [if trans.getAppliedStereotypes()->select(st:Stereotype|st.name='XFGDiscEnergy')->
       notEmpty()]
63   [let st: Stereotype = trans.getAppliedStereotype('XFG::XFGDiscEnergy')]
64   [trans.getValue(st, 'expr')/]
65   [/let]
66 [/if]
67 [/template]
```

Listing 1.8. Handling Transitions

Transforming State machine states into XFG locations. The last elements the transformation handles are UML states, which are transformed to location. Besides transitions covered above, we need to handle invariants and continuous energy consumption. The general template for handling states is shown Listing 1.9:

- **"Committed" States.** To model situations in which the state has to be left immediately upon entering, we use « `XFGCommitted` » . Lines 26-30 checks for the presence of the stereotype applied to the state and appends the keyword `committed` before the XFG location name,
- **State invariants.** State invariants are translated into location invariants. The translation algorithm is the same as for guards in transitions. This algorithm is shown lines 13-15,
- **Translating continuous energy consumption.** We use « `XFGContEnergy` » stereotype to model continuous energy consumption on states. This is translated into an internal, non-guarded, XFG transition where the update is the dotted consumption of the energy variable modelled by an equation entered in the `'expr'` tagged value of « `XFGContEnergy` » . The translation is defined lines 18-24.

```
[template public genLocation(state : State) post (trim())]
[committed(state)/] [state.name/] [if state.stateInvariant->notEmpty()][
    genStateInvariant(state.stateInvariant)/][/if]{
              [for (trans : Transition | state.outgoing->select(t:Transition| not
                   t.target.oclIsTypeOf(FinalState)))]
         [genTransition(trans)/]
              [/for]
        [if (state.getAppliedStereotypes()->notEmpty() and state.
            getAppliedStereotypes()->select(st| st.oclAsType(Stereotype).name='
            XFGContEnergy')->notEmpty())]
         [genContEnergyCP(state)/]
        [/if]
       [/]
}
[/template]

[template public genStateInvariant(cons:Constraint) post(trim())]
inv ([cons.specification.oclAsType(OpaqueExpression)._body/])
[/template]


[template public genContEnergyCP(state:State) post (trim())]
[let st : Stereotype = state.getAppliedStereotypes()->select(st| st.oclAsType(
    Stereotype).name='XFGContEnergy')->any(true)]
when true do
 [state.getValue(st,'expr')/]
goto [state.name/]
[/let]
[/template]

[template public committed(state:State) post(trim())]
[if state.getAppliedStereotypes()->select(st:Stereotype|st.name='XFGCommitted')->
    notEmpty()]
committed
[/if]
[/template]
```

Listing 1.9. Handling Locations

A Appendix: The XFG Profile

In this section, we summarize the XFG Profile. The domain diagram of the profile is given Figure 8.

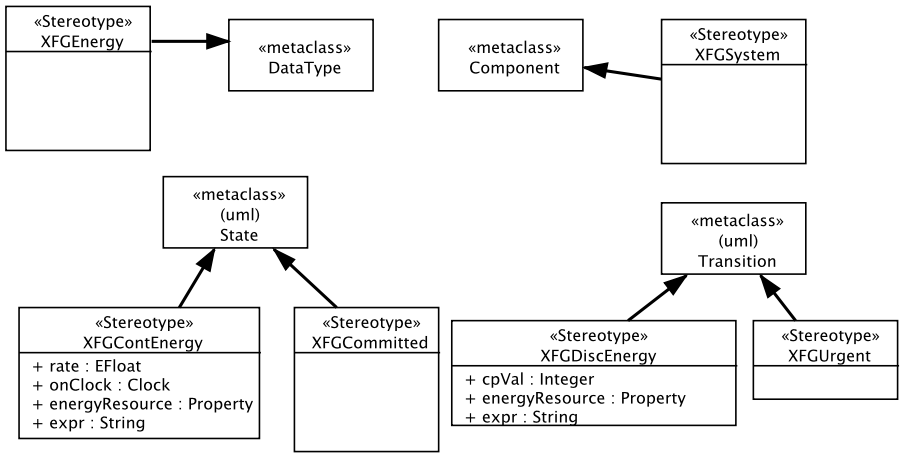


Fig. 8. Domain Diagram of XFG Profile

The following table summarizes the stereotypes, their application context and constraints.

Name	Base UML Metaclass	Tagged Properties	Description	Constraints
« XFGSystem »	Component	None	Models the top-level EAST-ADL function type to be considered for XFG generation.	There cannot be more than one application of this stereotype in a model.
« XFGEnergy »	Datatype	None	Models the special energy variable to be generated in XFG textual format	none
« XFGContEnergy »	State	rate : energy consumption rate, onClock: the clock referred to, energyResource: the energy resource to be updated, expr : the XFG code statement to be inserted in the XFG specification	Models the continuous conception of energy by multiplying the rate by the clock value	None
« XFGCommitted »	State	None	Models "committed" states, left immediately upon entering	none
« XFGDiscEnergy »	Transition	cpVal : amount of energy consumed, energyResouce: the energy resource to be updated, expr : the XFG code statement to be inserted in the XFG specification	Discrete consumption of Energy	None
« XFGUrgent »	Transition	None	Denotes urgent transitions, taken as soon there are enabled	none

Table 1. XFG Profile Stereotypes

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. ATESST. Atesst eu project website. `http://www.atesst.org/`.
4. Advancing Traffic Efficiency and Safety through Software Technology Phase 2 (European project), 2010. http://www.atesst.org.
5. AUTomotive Open System Architecture, 2010. http://www.autosar.org.
6. Gerd Berhmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing uppaal over 15 years. *Software - Practice and Experience*, December 2010.
7. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III*, volume 1066 of *LNCS*. 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.
9. EAST-ADL Consortium. East-adl domain model specification v2.1.9. Technical report, Maenad European Project, 2011.
10. L. Feng, D.J. Chen, H. Lönn, and M. Törngren. Verifying system behaviors in east-adl2 with the spin model checker. In *Mechatronics and Automation (ICMA), 2010 International Conference on*, pages 144–149. IEEE, 2010.
11. 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.
12. T.A. Henzinger and P.-H. Ho. HyTech: The cornell hybrid technology tool. volume 1019 of *LNCS*, pages 29–43. Springer-Verlag, 1995.
13. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to hytech, 1996.
14. MAENAD. MAENAD Project webiste. `http://www.maenad.eu/`.
15. MAENAD Project. East-adl domain model specification (version 2.1.9). Technical report, EAST-EEA, 2011.
16. OMG/INCOSE. Systems modeling language version 1.2. Technical report, OMG, June 2010.
17. OMG, UML profile for MARTE, 2011. http://www.omgwiki.org/marte/.
18. Open Source Tool for Graphical UML2 Modeling, 2010. http://www.papyrusuml.org.
19. TIMing MOdel, 2010. http://www.timmo-2-use.org/timmo/index.htm.
20. UML 2.4, 2011. http://www.omg.org/spec/UML/2.4.1/.
21. UPPAAL CORA, 2012. http://people.cs.aau.dk/ adavid/cora/language.html.