Dynamic Parametrized Transition Systems

–

LTU & ENS Paris-Saclay

Ulysse REMOND

01/02/2020 – 02/07/2020

Contents

1 General view of the project

1.1 General context

a) A paradigm shift in the industry

Nowadays, the industry is shifting from mass automation to mass customization. As a consequence, it is becoming ever more commonplace to move and reconfigurate industrial systems on-the-fly. Additional components can be removed or added at any time. It is therefore becoming pivotal for industrial models to encompass this dynamism.

The most widely used international standard for industrial systems is the IEC 61499. It already underwent several major additions, to cater for the various needs of the industry. The next addition, Part 5 of the IEC 61499, will tackle dynamism and reconfiguration problems.

b) Formal verification

Research is currently being done on what kind of framework could be provided, that could ensure both interoperability, clarity and an easy formal verification.

Indeed, these requirements could be understood as mutually exclusive, as dynamism comes in many forms, and as formal verification methods are yet to be widely used by engineers. Dynamism comes in many forms:

- One type of dynamism is a physical dynamism, for which verification protocols using hybrid automata are being devised.
- Another type of dynamism is the state reconfiguration dynamism[Bra04], that makes machines able to reconfigurate on the fly, and is either triggered by an external event (ad-hoc dynamism[End94]), or when an internal set of conditions is met (programmed dynamism[End94], which is easier to model).
- Dynamism may also include architectural dynamism[DRBIP][Cim18],

Formal verification techniques include SMV to Petri nets transformations and unfoldings verifications. One technique developed in [Vya17] allows the development of a plant model from a specification, thus ensuring that these requirements are met.

1.2 The problem under study (Verification of Dynamic CPS)

As mass customization is becoming ubiquitous in the industry, plants must be flexible and agile, and plant models should mimic closely this flexibility.

Concretely speaking, plants are now revolving around wirelessly communicating, removable, reconfigurable on-the-fly, internet addressable distributed controllers. More often than not, this flexibility goes along product driven design patterns[Atm19].

My internship director, Valeriy Vyatkin, along with numerous authors have already studied broadly how to model dynamic cyber-physical systems on a computer. These works pave the way to a better simulation and a better understanding of dynamic CyberPhysical Systems (CPS). One of the suggested improvement of the approach of Vyatkin & Al. is to add a way to model the possible internal and architectural reconfigurations.

My work consists in finding several techniques to model this variability of plant models in an easily-verifiable way, and to possibly set up a framework for such a formal verification

1.3 My contribution

First of all, I reviewed of several papers on the subject. I understood key notions of function blocks, cyberphysical systems, plant models, agnosticity, and how IEC 61499 systems were currently verified.

Then, as explained in Part 3, I devised a toy Architecture Description Language, inspired by the work of A. Cimatti, to get to know better those tools. I extended a language modeling architectural changes to one, that is able to model an entire system, consisting of several machines, each with inputs and internal states.

After this, as shown in Part 4, I studied Hybrid automata, unrelated papers, and various Dynamical Software Architectures. Then, I reviewed other DSA (Dynamical Software Architectures).

Furthermore, as explicited in Part 5, I implemented a way to encompass function block variability with phantom function blocks in IEC 61499. The extension of Cimatti & Al.'s ADL described in part 3 can be used to represent this type of limited state dynamism (where there is only a finite number of possible internal states configurations). I designed a framework for testing such function blocks, but in fact all types of composite function blocks.

1.4 Notations

a) Abbreviations

The base unit of the IEC61499 is the function block, usually noted FB. Externally, FB come with signal interfaces (event and data). Internally, they can either be made of several other FB, in which case they are called Composite Function Blocks (CFB) or of one internal state machine, called Execution Control Chart, or ECC [Atm19].

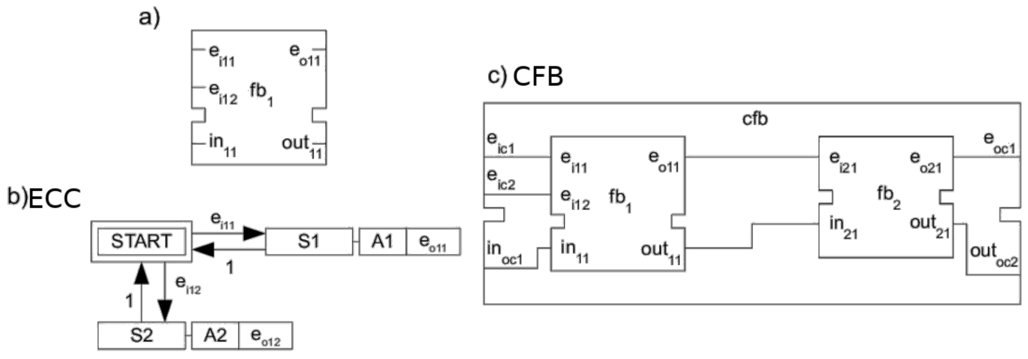


Figure 1: a) event (e) and data (in & out) interfaces, b) ECC c) CFB

Architecture Description Languages are abbreviated as ADL[Cim18].

b) Quantifiers

Consider the following formula :

$\exists_{I_1} \bar{i}_1,\ \exists_{I_2^C}\ \bar{i}_2,\ \alpha$

It means that there exists a subset of I_1, named $\bar{i}_1$, and a set $\bar{i}_2$ of elements of the same type as the ones in I_2, so that alpha is true. For instance, with $I_1 = \{0\}, I_2 = \{0, 2\}, \alpha = ((\bar{i}_1 = \bar{i}_2) \vee (3 \in \bar{i}_2))$, the formula above is true, for instance with $\bar{i}_1 = \emptyset$ and $\bar{i}_2 = \{3\}$. These abbreviations were used by [Cim18] in particular.

2 The human context

My internship took place from 1st of February to 2nd of July in the *Department of Computer Science, Electrical and Space Engineering* of the LTU (*Luleå Tekniska Universitet*). The LTU is located in Porsön, in the suburbs of Luleå, which is one of the northernmost ports in Sweden and the capital of Norrbotten county. More specifically, I worked in the *Dependable Communication and Computation Systems* laboratory, in the *Industrial Informatics and Automation* (IIA) team.

During my internship, I worked under the leadership of Valeriy Vyatkin, leader of the IIA team, with Sandeep Patil, senior lecturer who especially helped making my arrival as easy and efficient as possible, and with Dmitrii Drozdov, who gave me clear work in the midst of the pandemic.

My largest difficulty for me, by far, has been my total lack of contact with people in Sweden as soon as the coronavirus epidemic started. At the start of the pandemic, I began by only working on side project, such as one with my team member D. Drozdov - about implementing a connection configurator for connecting to (and for each member) of a swarm of small devices. After this, I even almost entirely stopped going out of my flat, maybe because I am not very social and something of an hypochondriac as well.

3 Dynamic Parametrized Architecture : A Case Study

I mainly studied the ADL created by Cimatti & Al [Cim18]. (originally created in order to specify and verify formally dynamic architectures), and then extended it to encompass more types of dynamism.

3.1 Parametrized Architecture

During the following, formulae are considered to be first order, quantifier-free, and without equality between sets.

Definition: An INDEX SET is a finite subset of $\mathbb{N}$

Definition: Given a set S, an index set I, S IS INDEXED BY I if and only if there exists a bijective mapping from I to S

Definition: An INDEX SET OF PARAMETERS is a variable whose domain is $P_f(\mathbb{N})$

Definition: An ARCHITECTURAL CONFIGURATION is a pair (C,E), where:

- C is a set of COMPONENTS
- $E \subseteq C \times C$ is a set of CONNECTIONS (between components)

Definition: An STRUCTURED *architectural configuration* is a pair $(\mathscr{C}, E)$ where :

- $\mathscr{C}$ is a finite set of disjoint sets of COMPONENTS indexed by index sets
- $E \subseteq \bigcup_{\substack{C_1 \in \mathscr{C} \\ C_2 \in \mathscr{C}}} C_1 \times C_2$ is a set of CONNECTIONS (still between components)

Example: $I_1 = \{1,2,3\}, I_2 = \{5\}, \mathscr{C} = \{C_1, C_2, C_3\}$, where $C_1 = \{C_{1,i}\}_{i \in I_1}$,$C_2 = \{C_{2,i}\}_{i \in I_1}$, $C_3 = \{C_{3,i}\}_{i \in I_2}$

I_1 and I_2 are two index sets. C_1 and C_2 are indexed by I_1.

With $E = \{(C_{1,i}, C_{2,i}), i \in I_1\}$ as a set of edges, $(\mathscr{C}, E)$ is a valid system of parameters (assuming that the no $C_{i,j}$ are equal. To ensure that the sets of components are disjoint, one will map them towards distinct symbols, with so-called system of parameters.

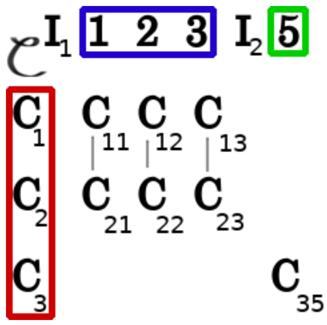


Figure 2: A possible representation of the structured architectural configuration

Definition: A SYSTEM OF PARAMETERS is a pair $(\mathfrak{I}, \mathscr{V})$, where:

- $\mathfrak{I}$ is a finite set of symbols for index set parameters
- $\mathscr{V}$ is a finite set of symbols $\{v \mid v \in \mathscr{V}\}$ where each $v \in \mathscr{V}$ is associated with :
  - an index set parameter symbol $I_v \in \mathfrak{I}$
  - a sort $sort_v \in \{bool, int\}$

Definition: An ASSIGNMENT μ to a system of parameters $(\mathfrak{I}, \mathscr{V})$ is a pair such that:

- $\mu = (\mu_{\mathfrak{I}}, \{\mu_V\}_{V \in \mathscr{V}})$
- $\mu_{\mathfrak{I}} : \mathfrak{I} \to P_f(\mathbb{N})$
- $\forall\ V \in \mathscr{V}, \mu_v : \mu_{\mathfrak{I}}(I_V) \to dom(sort_V)$ (in practice, $dom(bool) = (B)$, and $dom(int) = (Z)$)

Definition: A PARAMETRIZED ARCHITECTURE is a tuple $A = (\mathfrak{I}, \mathscr{V}, (P), \psi, \phi)$, where:

- $(\mathfrak{I}, \mathscr{V})$ is a system of parameters
- $\mathscr{P}$ is a finite set of parametrized and indexed sets of components (each set $p \in \mathscr{P}$ is associated with an index set $I_p \in \mathfrak{I}$)
- $\psi = \{\psi_p(x)\}_{p \in \mathscr{P}}$ is a set of formulae *(component guards)* over $\mathfrak{I}$, $\mathscr{V}$ and a free variable x
- $\phi = \{\phi_p(x, y)\}_{p \in \mathscr{P}}$ is a set of formulae *(connection guards)* over $\mathfrak{I}$, $\mathscr{V}$ and two free variables x, y

Definition: Given a parametrized architecture A and an assignment μ to the system of parameters $(\mathfrak{I}, \mathscr{V})$, the INSTANCIATED CONFIGURATION of A defined by the assignment μ is given by $\mu(A) := (C_\mu, E_\mu)$, where:

- $C_\mu = \{C_{\mu,p} \mid p \in \mathscr{P}\}$, where $C_{\mu,p}$ is defined by:

$$C_{\mu,p} = \{c_{p,i} \mid i \in \mu_{\mathfrak{I}}(I_p) \text{ and } [\![\psi_p(i/x)]\!]_\mu = \top\}$$

- $\forall C_{\mu,p}, C_{\mu,q} \in C_\mu, \forall$ component instances $c_{p,i} \in C_{\mu,p}, c_{q,j} \in C_{\mu,q}$,

$$(c_{p,i}, c_{q,j}) \in E_\mu \iff [\![\phi_{p,q}(i/x, j/y)]\!]_\mu = \top$$

(C_μ, E_μ) is a structured architectural configuration.

Example: The pneumatic cylinder from [Vya17] can be represented as $A = (\mathfrak{I}, \mathscr{V}, (P), \psi, \phi)$, where:

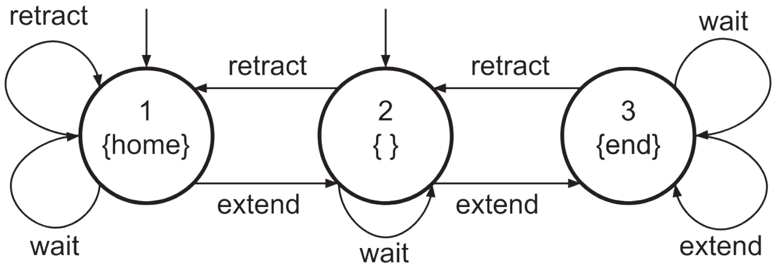


Figure 3: The pneumatic cylinder

- $\mathfrak{I} = \{C\}$
- $\mathscr{V} = \{home, end\}$, and both are associated with C and the sort *bool*

(Let's note that C, *home* and *end* are **symbols**. At this point, one knows very little about the architecture of the system under scrutiny.)

- $\mathscr{P} = \{p\}$, $I_p = C$
- $\psi = \{\psi_p(x)\}$, $\psi_p(x) = \top$
- $\phi = \{\phi_{p,p}(x)\}$, $\phi_{p,p}(x, y) = (\neg home(x) \vee \neg end(y)) \wedge (\neg home(y) \vee \neg end(x))$

There is only one machine, C, that can be in one of as many states as the assignment will specify - what [Cim18] and the definitions above describe as components are will be in this report treated as states. In each of these states, the variables *home* and *end* may be true or false. What is described as a component in the definition can in fact be seen as a mere state in the examples, as in the figure 3.

Let's now give an assignment to the system of parameters $(\mathfrak{I}, \mathscr{V})$:
$\mu_{\mathfrak{I}} \equiv C \mapsto \{1, 2, 3\}$
$\mu_{home} \equiv 1 \mapsto true, 2 \mapsto false, 3 \mapsto false$
$\mu_{end} \equiv 1 \mapsto false, 2 \mapsto false, 3 \mapsto true$

The instantiated configuration $\mu(A)$ is:

$C_\mu = \{C_{\mu,p}\}$, where $C_{\mu,p} = \{C_{p,1}, C_{p,2}, C_{p,3}\}$.
$E_\mu = \{(C_{p,1}, C_{p,1}), (C_{p,1}, C_{p,2}), (C_{p,2}, C_{p,1}), (C_{p,2}, C_{p,2}), (C_{p,2}, C_{p,3}), (C_{p,3}, C_{p,2}), (C_{p,3}, C_{p,3})\}$

At this point, we know how to describe a static system. At any point of its execution, the architecture of the system can be expressed as an INSTANTIATED CONFIGURATION (like $\mu(A)$). It evolves according to certain rules, as described below, but at each step, the configuration itself is valid with regard to the pattern set up by the PARAMETRIZED ARCHITECTURE A. Only the assignment μ will change.

In [Cim18], they describe TRANSITION FORMULAE, that safeguard how the instantiation may evolve, but I will not need such a complex and complete tool in this report, so the τ in the following definition will remain pretty obscure.

3.2 Architectural Variability of Parametrized Architectures

The architecture of the connectors between components may vary.

Definition: A DYNAMIC PARAMETRIZED ARCHITECTURE is a tuple (A, ι, κ, τ), where:

- $A = (\mathfrak{I}, \mathscr{V}, \mathscr{P}, \psi, \phi)$ is a parametrized architecture
- ι is a formula over $(\mathfrak{I}, \mathscr{V})$ specifying the set of INITIAL ASSIGNMENTS. ι can be written as :$\forall_{I_1} \bar{i}_1$, $\forall_{I_2^C} \bar{i}_2$, α, where no $=$ and $\subseteq$ between index sets appear under a negation in α.
- κ is a formula over $(\mathfrak{I}, \mathscr{V})$ specifying the INVARIANT. κ is a quantifier-free formula without operations $=$ and $\subseteq$ between index sets.
- τ is a disjunction of reconfiguration formulae over $(\mathfrak{I}, \mathscr{V})$ specifying the RECONFIGURATION TRANSITIONS. Notably, τ is still a first-order formula, but its nature will not be discussed in this report. What matters is that it is able to add or remove indices to index sets in $\mathfrak{I}$, and assign new variables to one or several indices.

For the sake of clarity and brevity of this report, let's not rewrite his definition of transition formulae[Cim18]. He defines it as a disjunction of a stronger version of what I describe as a reconfiguration formula.

Definition: Given a DPA as previously, the SET OF INITIAL CONFIGURATIONS is :
$I(A) = \{\mu(A) : \mu$ is an assignment to $(\mathfrak{I}, \mathscr{V})$ such that $[\![\iota]\!]_\mu = [\![\kappa]\!]_\mu = \top\}$

Definition: A configuration $\rho(A)$ is DIRECTLY REACHABLE from a configuration $\mu(A)$ iff :
$[\![\iota]\!]_{\mu,\rho} = [\![\kappa]\!]_\rho = \top$

Same example of the pneumatic cylinder: with the A and μ defined before, one can add appropriate formulae:

$\iota = \forall_C\ i, C = \{1, 2, 3\} \wedge \neg home(2) \wedge \neg end(2) \wedge end(3) \wedge (\neg end(i) \vee \neg home(i))$

$\kappa = home(1) \wedge \neg end(1)$

$\tau = \tau_+ \vee \tau_-$, where :

$\tau_+ = \exists j \notin C, (C' = C \cup \{j\} \wedge end(j) \wedge (\forall i \in C, \neg end(i)) \wedge \gamma_\beta)$ is a formula to add a state, and τ_- would be the formula to remove a state. Note that γ_β is defined as in [Cim18] and takes care of the frame conditions.

In this example, only one type of state is defined ($\mathscr{C}$ is a singleton), and it is not easy to define other components. This way to describe the system does not scale easily, as $\mathfrak{I}$ settles the size of the internal machines of the components, and not the number of components. This type of approach may be useful if a bounded number of components are to be used, each having a wildly unpredictable behavior. However, usually, there is only a limited amount of possible behaviors, the problem being to add and remove lots of components efficiently.

3.3 Integrated variability of the transitions & the "machine vision"

We can define another architecture $B = (\mathfrak{I}, \mathscr{V}, \mathscr{P}, \psi, \phi)$ for the pneumatic cylinder:

$\mathfrak{I} = \{I\}$ (set of symbols for sets of indices) sets the number of components.

$\mathscr{V} = \{inactive\}$ (set of symbols for variables) sets the variables, that are useful to change the behavior of a component.

$\mathscr{P} = \{H, M, E\}$ (set of indexed sets) sets the type of states in the internal state machines. The index for all those variables is I

$\psi = \{\psi_\tau(x)\}_{\tau \in \mathscr{P}}$ (set of formulae for components) selects which components exist.

$\phi = \{\phi_{\sigma\tau}(x)\}_{\tau,\sigma \in \mathscr{P}}$ (set of formulae for connections) selects which connections exist.

In the current paradigm, I only want to simulate addition of new components, and not modification of internal state machines. It seems only fitting to say that each machine will correspond to a unique index. This is why this paradigm is called "**machine vision**", as each index matches with one machine. As a consequence, I will set :

$\forall \tau \in \mathscr{P}$, $\psi_\tau(x) = \top$

$\forall \sigma, \tau \in \mathscr{P}$, $\phi_{\sigma,\tau}(x,y) = x = y \wedge \neg(((\sigma = H) \wedge (\tau = E)) \vee ((\sigma = E) \wedge (\tau = H)))$

One could also choose to disable transitions between different states in inactive components, in that case, the connection formulae would be $\phi'_{\sigma,\tau}$, such that:

$\forall \sigma, \tau \in \mathscr{P}$, $\phi'_{\sigma,\tau}(x,y) = \phi_{\sigma,\tau}(x,y) \wedge (\neg inactive(x) \vee \sigma = \tau)$

I then noticed that it would in fact be possible to simulate transitions with the variables V. For instance, we could define a $\mathscr{V}' = \{wait, retract, extend\}$ (with still all sorts set as bool). The transition formula would then be:

$\forall \sigma, \tau \in \mathscr{P}$, $\phi''_{\sigma,\tau}(x,y) = x = y \wedge (\phi_{wait}(x) \vee \phi_{retract}(x) \vee \phi_{extend}(x))$, where

$\phi_{wait}(x) = (wait(x) \wedge \sigma = \tau)$

$\phi_{retract}(x) = (retract(x) \wedge ((\sigma = H \wedge \tau = H) \vee (\sigma = M \wedge \tau = H) \vee (\sigma = E \wedge \tau = M))$

$\phi_{extend}(x) = (extend(x) \wedge ((\sigma = H \wedge \tau = M) \vee (\sigma = M \wedge \tau = E) \vee (\sigma = E \wedge \tau = E))$

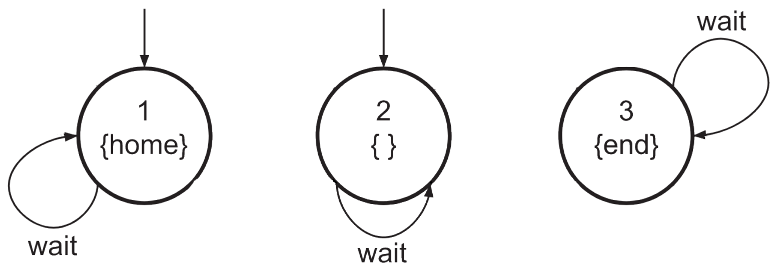


Figure 4: *wait* transitions activated

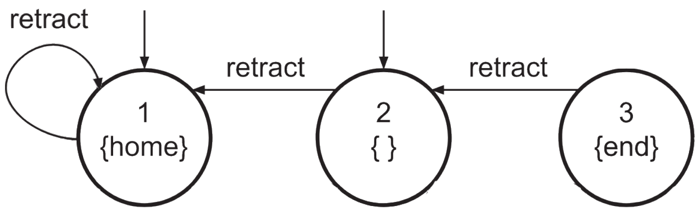


Figure 5: *retract* transitions activated

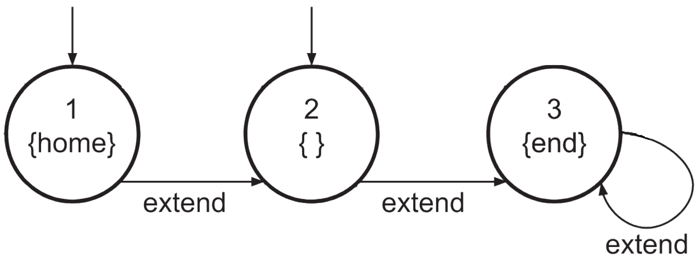


Figure 6: *extend* transitions activated

These PA can naturally be extended to a DPA - for instance with the B' (with ϕ' as a transition formula):

$\iota := I = \{1\}$

$\kappa := \top$

$\tau := \tau_A \vee \tau_D \vee \tau_+ \vee \tau_-$, where:

$\tau_A \approx \exists i \in I, \neg inactive(i)$ (activation)

$\tau_- \approx \exists i \in I, inactive(i) \wedge I' = I = I \backslash \{i\}$ (remove a component) And so on and so forth with the other formulae..

Note that it is written $\approx$ instead of $=$ because, as usual, I omit the γ_β associated to the written β formulae. The problems inherent to this approach are still quite a number;

- First of all, one still has no way to determine in which state is a component at any point of its execution. It is not very damaging if one only wants to test broadly the system, one may have to take into account all possible states at each step.
- Second of all, the reconfiguration and the interaction steps are not separated with this method, which leads to more unnecessary entanglement of otherwise separated behavior. Indeed, some integer or boolean variables may be used as intended (like *inactive*), whereas other are solely used as an 'input marker' (like *extend*, *retract* and *wait*) - they embody what is called an input in automata theory. One could only use transition formulae without the 'input marker' variables for reconfiguration steps and transition formulae with no other variables and no state creation or destruction (i.e. only using type 4) or 5) formula as explained in [Cim18]) for interaction steps, but it would be quite a far-fetched method.

This is why I devised another method, where interaction and reconfiguration steps are clearly separated, as advocated in Fig. 4

3.4 Separated transitions and Input-Parametrized Architecture

Definition: An Input-Parametrized Architecture is a tuple $A = (\mathfrak{I}, \mathscr{V}, \mathscr{P}, In, \psi, \phi)$, where:

- $(\mathfrak{I}, \mathscr{V})$ is a system of parameters
- $\mathscr{P}$ is a finite set of parametrized indexed sets of components, that all have the same parametrization $I_{\mathscr{P}}$
- In is a finite set (alphabet of possible inputs)
- ψ is defined as previously; no inputs appear in these formulae.
- $\phi = \{\phi_{PQ}(x)\}_{P,Q\in\mathscr{P}}$ is a set of formulae over $\mathfrak{I}, \mathscr{V}$ and x (free variable)

To model a system consisting of several components into an input-parametrized architecture, one uses the same paradigms as in the example of section b) above. Called the "**machine vision**", it consists in seeing each index as a number for a machine, which has several states, that are the "components" from before.

Namely, we have got as many $I \in \mathfrak{I}$ as we have different types of machines. In our example, there is only one such I, that will be named $I_{\mathscr{P}}$ for the sake of clarity. In this report, it will always to be considered to be the case, but it could easily be extended to any arbitrary number of indexed sets, each I_j representing a different machine whose states would be the $P \in \mathscr{P}$ with I_j as index set, and whose variables would be the $V \in \mathscr{V}$ with I_j as index set.

As we only have one type of machines, $|I_{\mathscr{P}}|$ is the number of machines. $|P|$ is the number of states per machine. Each machine is in exactly one state at each step of the system, according to the method of instantiation described below.

The parametrized architecture B'' of the pneumatic cylinder (the one from the previous subsection that uses $\mathscr{V}'$ and ϕ'') can be changed into an input-parametrized architecture $Q = (\{I\}, \{\}, \mathscr{P}, \psi, \phi)$, where:

- $\mathscr{P} = \{H, M, E\}$ and $I_{\mathscr{P}} = I$
- $\phi_P(x) = \top$
- $\psi_{MM}(x) = \psi_{HH}(x) = \psi_{EE}(x) = wait(x)$

- $\psi_{EM}(x) = \psi_{MH}(x) = \psi_{HH}(x) = retract(x)$
- $\psi_{HM}(x) = \psi_{ME}(x) = \psi_{EE}(x) = extend(x)$

Definition: Given an input-parametrized architecture A and an assignment μ to the system of parameters $(\mathfrak{I}, \mathscr{V})$, the INSTANTIATED CONFIGURATION defined by the assignment μ is (C_μ, E_μ, S_μ), where C_μ, E_μ are defined as previously and:

$$\begin{aligned} S_\mu \colon I_{\mathscr{P}} &\to P \\ i &\mapsto p \end{aligned}$$

meaning that $\forall i \in I_{\mathscr{P}}$, $c_{S_\mu(i),i} \in C_{\mu p}$ is the current state of the component number i. We are ensured that this state exists because all $P \in \mathscr{P}$ have the same parametrization.

Concretely, this is a configuration instantiation as usual, but the current state of each machine is specified by the function S_μ.

One possible instantiation for the input-parametrized architecture Q defined above could be: $\mu(I_{\mathscr{P}}) = \{4 \dots 26\}$ and

$$\begin{aligned} S_\mu \colon I_{\mathscr{P}} &\to P \\ 17 &\mapsto M \\ i \neq 17 &\mapsto H \end{aligned}$$

which would for instance imply:

- $C_\mu = \{C_{\mu H}, C_{\mu M}, C_{\mu E}\}$
- $C_{\mu P} = \{C_{i,P}\}_{i \in \{4 \dots 26\}}$ for $P \in \mathscr{P}$

Definition: Given an input-parametrized architecture $A = (\mathfrak{I}, \mathscr{V}, \mathscr{P}, In, \psi, \phi)$ and an instantiated configuration $G_1 = (C_\mu, E_\mu, S_\mu)$, an instantiated configuration $G_2 = (C_\rho, E_\rho, S_\rho)$ is said to be DIRECTLY REACHABLE from G_1 iff:

- $C_\mu = C_\rho$ meaning that the same components/states are activated (no component reconfiguration)
- $E_\mu = E_\rho$ meaning that the same connections are activated (no connection reconfiguration)
- $\forall i \in I_{\mathscr{P}}$, $\phi_{S_\mu(i), S_\rho(i)}(i)$, meaning that there exists a transition in the machine i between the states $S_\mu(i)$ and $S_\rho(i)$

See the annex B for more details about this definition. It concretely does what can be seen in Fig. 4 5 6, which is allowing only some transitions at each step for each machine. The Dynamic Input-Parametrized architecture are defined as the DPA, but with inputs being used as variables:

Definition: A DYNAMIC INPUT-PARAMETRIZED ARCHITECTURE is a tuple (A, ι, κ, τ), where:

- $A = (\mathfrak{I}, \mathscr{V}, \mathscr{P}, In, \psi, \phi)$ is an input-parametrized architecture
- ι is a formula over $(\mathfrak{I}, \mathscr{V} \cup In, F(A))$ specifying the set of INITIAL ASSIGNMENTS. ι can be written as :$\forall_{I_1}\bar{i}_1$, $\forall_{I_2^C}\ \bar{i}_2$, α, where no $=$ and $\subseteq$ between index sets appear under a negation in α.
- κ is a formula over $(\mathfrak{I}, \mathscr{V} \cup In, F(A))$ specifying the INVARIANT. κ is a quantifier-free formula without operations $=$ and $\subseteq$ between index sets.
- τ is a disjunction of reconfiguration formulae over $(\mathfrak{I}, \mathscr{V} \cup In, F(A))$ specifying the RECONFIGURATION TRANSITIONS. τ is is now able to change the inputs given to one or several indices.

where $F(A)$ is the set of all formula of the form : $S(j) = P$ with j a variable (if avaliable) or an index (a natural integer) and $P \in \mathscr{P}$

The completed DYNAMIC INPUT-PARAMETRIZED ARCHITECTURE of the pneumatic cylinder can be written as (A, ι, κ, τ), where:

- Q has been defined above
- $\iota = I = \{1\} \wedge \neg S(1) = E$
- $\kappa = \top$
- $\tau = \tau_+ \vee \tau_-$
- $\tau_+ = \exists_{I^C}\ i,\ (I' = I \cup \{i\} \wedge \neg S(i) = E)$ (ensures that the component does not start by being extended)
- $\tau_+ = \exists_I\ i,\ (I' = I \backslash \{i\} \wedge \neg S(i) = E)$ (ensures that the component will not be extended when being restarted ; it is not useful formally speaking, but this is an expected behavior not to be deactivating while still active)

Overall what is important in this formalism is that one can express both reconfiguration dynamism (going upwards in Fig. 7 below) and transition dynamism (going rightwards in Fig. 7) in this modified version of the ADL described in [Cim18]. It is therefore now able to serve the same purpose as DR-BIP (expressing reconfigurable distributed systems), even though one is a programming language and the other one is still formal. Another advantage over DR-BIP is that the number of components is unbounded.

As a consequence, one possible future improvement could be to translate some restricted DR-BIP instances in this formalism (or reciprocally, as it seems that DR-BIP can express more things, find a canonical way to translate Dynamical Input-Parametrized Architectures into DR-BIP systems).

4 Dynamic Systems

4.1 What are dynamic systems?

CPS are systems where computer control, communication between components and movement in space are topical. In this section, we focus on the Dynamic aspects of CPS, which are generally revolving around staying safe and efficient while moving.

CPS can always be framed as hybrid systems, where discrete dynamics (control decisions) are coupled with continuous dynamics (differential equations of the movement).

The additional type of dynamism I focused on is the architecture dynamism and internal state dynamism (sometimes called software dynamism [Bra04]). Such dynamism means that component are being changed or reconfigurated on the fly.

One may represent the dynamism of system with potential architecture rewriting as a 2D lattice :

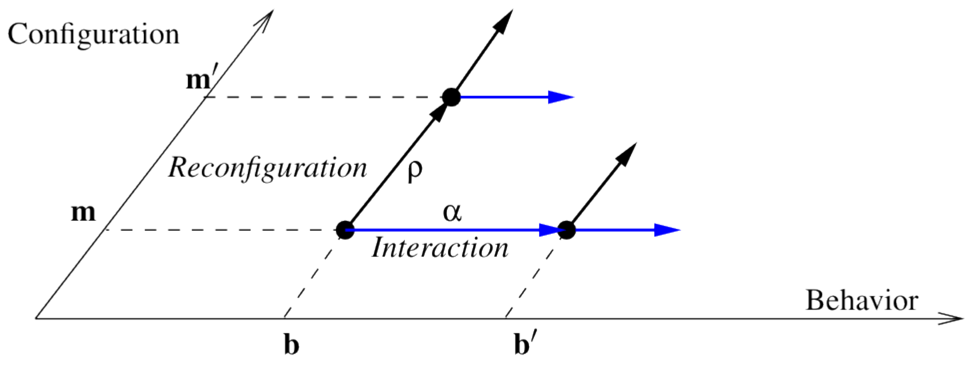


Figure 7: Interaction steps can also just be transitions, inside each component (synchronous) or inside one component (asynchronous)

The DR-BIP formal framework fully embraces this twofold evolution[DRBIP]. DR-BIP lies on the concept of $motif$, which is mainly a sort of group of components, following a certain pattern (for instance, a ring or a star pattern) and some reconfiguration rules.

When defining a motif type, one defines also the way it will allow its components to interact with each other (IR, interaction rules) and how it will reconfigurate, by absorbing other components or removing a component from the motif (RR, reconfiguration rules).

4.2 Current validation

a) Hybrid Systems

One may use discrete, continuous or adversarial dynamics, differential game logic, quantified differential dynamical logic[DDL] with hybrid automata to verify systems, as in [Ho96]

One procedure very similar to the counter example guided abstraction refinement (CEGAR) has even been devised specifically for hybrid automata[Oak03]. However, this approach :

- is efficient in undoing automatic abstractions of an abstract hybrid system from spurious counterexample. A spurious counterexample us a concrete path that breaks in some failing state, which can be called 'dead end' or 'bad' state. This idea will notably be used by the *observer function blocks*[Roop09] that I haven't fully implemented eventually.
- is struggling to discover constraints on free parameters.

What is also common is techniques based on symbolic decomposition, such as in the model checker HyTECH. How to verify temporal properties of a finite state abstraction of a hybrid automaton is currently known and well documented. To do so, one explores exhaustively the state space[DDL]. However, there cannot be a finite state equivalent to the continuous state space of hybrid automata. Accordingly, model checkers use approximation[Oak03].

b) Dynamic Architectures

If the number of components is bounded, formal verification of dynamic architectures can be easily reduced to usual verification. All components always exist within the system, but some are inactive components. Sometimes, a component controlling the activation could be added[Cim18].

c) Dynamic Software

An excellent review of the existing techniques to model Dynamical Software (DS) has been done by [Bra04], and summarized in a presentation that can be found on the GitHub of the internship[M1GIT]. In this, he notably describes the existence of a language based on temporal logic to describe architectural changes, and how this facilitates formal verification. The goal of [Cim18] is also to facilitate verification.

5 Phantom Function Blocks

5.1 Implementation

There are two main software implementing IEC 61499; 4DIAC on Linux, nxtStudio on Windows.

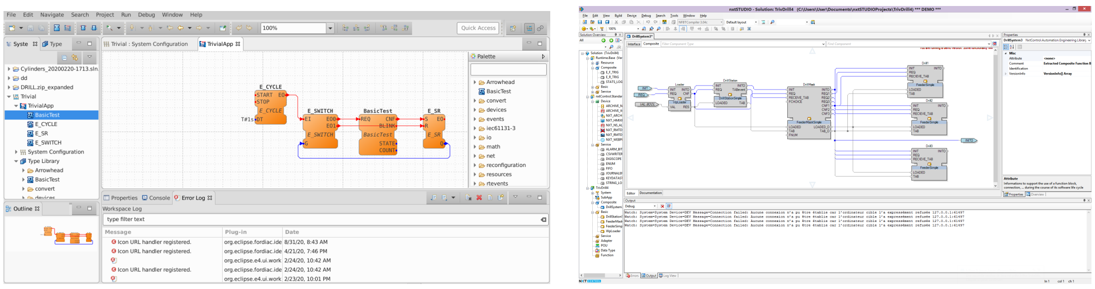
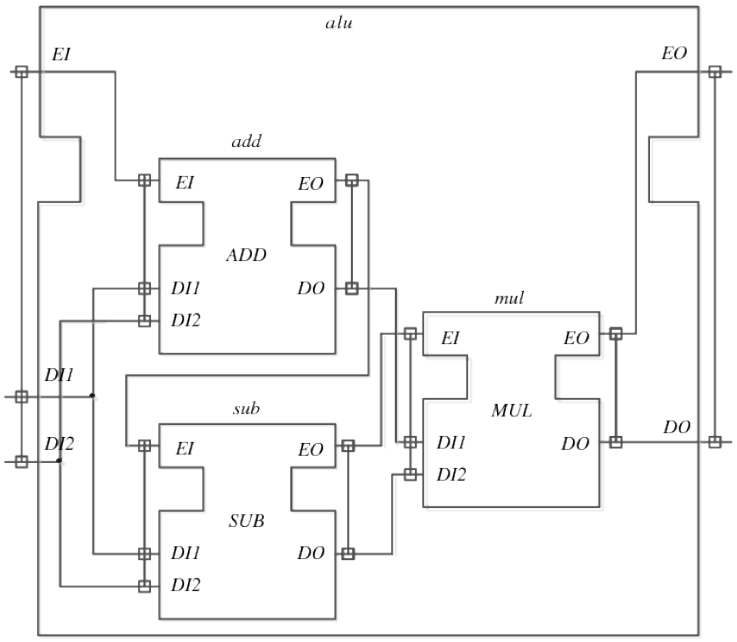


Figure 8: 4diac and nxtStudio interfaces

The idea behind my work with phantom FB was to provide a way to encompass into a seemingly unchanged structure several available configurations for a same function block. In other words, it means that I implemented a facade CFB, called "**mask**" behind which several FB (with the same inputs and outputs), each representing a possible configuration, can be activated, but never more than one at a time.



Fortunately, the FB structures, with the clear separation between the outside, with the signal and data interfaces, and the inside, with the execution control chart (ECC), and the possibility to devise composite function blocks (CFB), that encompass several function blocks as described by the figure on the right →

My implementation of the phantom FB for a FB named A consisted in a composite function block in which there were :

- N blocks of different reconfigurated versions of A, all obviously with the same signal interface, but with different ECCs.
- One mask block, with the same front interface as A, plus a "choice" input (data and signal, to get the renewed data) and N times A's input interface as its output interface, so that the N^{th} version of the drill receives an input (through the corresponding output from the mask) if and only if the mask gets as an input that the N^{th} configuration should be used.

Here is the display of the phantom drill in nxtStudio:

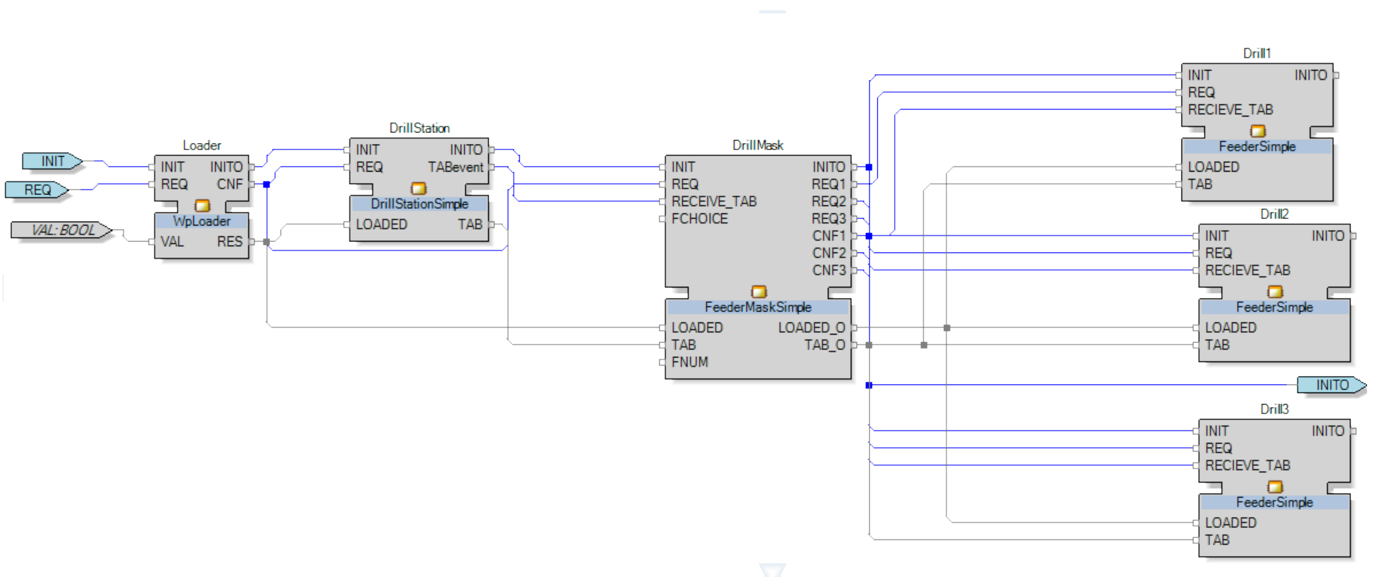


Figure 9: Phantom drill interface

The loader and the drill station FB are just part of the project but not of the would-be composite function block. There are three possible configurations for the Drill, noted Drill1, Drill2 and Drill3. All $INIT$ inputs are linked to the $INITO$ output of the mask, because all possible configurations must be initiated when the device is launched, so that they are eventually initialized when needed.

The other inputs in of the Drill$N$ FB are linked to the corresponding inN outputs of the mask. Note that the data inputs are all linked to the same outputs; it is because function blocks never read the content of their data inputs unless "told so" by the corresponding event input.

Note that there are non-linked inputs for the mask (namely the event $FCHOICE$ and the data $FNUM$); it is a trick so that I can more easily trigger by hand the needed values (these two correspond to the choice of the running configuration).

In this specific example, the configuration activated will always be the one specified by hand. For instance, if $FNUM = 2$ and $FCHOICE$ has been activated since $FNUM$ has been set to 2, then the running configuration is Drill2. As a consequence, this models ad-hoc dynamism [Bra04].

Nevertheless, it could be easily possible to model programmed dynamism[Bra04], just by adding a number of states tackling this in the ECC of the Mask.

5.2 Function Block Verification

Here is the full method to verify phantom function blocks:

1. create the different variations of the function blocks (FB) in nxtStudio 3.0.1.0

2. encompass all of them in a composite FB (CFB)

3. ensure that it is compatible with a SoftPLC (software Programmable Logic Controller) from nxtStudio 3.0.0.0

4. Convert the CFB to SMV, using the newest version of FB2SMV

5. Replace (manually or with any program) "integer" in the SMV code by "0..99" (or any constant, 19 works usually fine)

6. Replace all "x + 1" by "(x + 1) mod 100" (with the correct modulo for the number chosen in 5)

7. move it in the NuXMV folder and launch 'NuXMV "filename"'

8. type "go"

9. type "check_ltlspec -p [YOUR PROPERTY]"
   One may also type "check_ctlspec [ANOTHER PROPERTY]"
   Please note that properties have the following form:
   AG (LOADER.REQ = TRUE → EF (RECEIVE_TAB = TRUE))

What has been complex is that several versions of each software has been tested, and only works for a specific version of the other ones. I verified some properties that were easily verifiable by hand with the setup described above, for the expected results. However, it is worth noting that it was only very easy manual tests with very simple function blocks, so it may not have been the best verification.

This implementation of a phantom FB can still be looked as successful. It is quite a pivotal result, because this implementation, where I had to do all the connections by hand (I couldn't use any adapter, which is a ubiquitous feature allowing the merging of several inputs or outputs, as they are not supported by the FB2SMV converted), paves the way for the creation of a macro. Creating a macro, or a template, or syntactic-sugar, for reconfigurations as this one, is one of the possible goals of the Part 5 of the IEC 61499.

6 Results and conclusion

6.1 Results

My most meaningful results are:

- The extension of Cimatti ADL into a full system description language, by adding inputs and current states to each components.
- The concrete coding of a true distributed system with Iceblocks.
- A lot of bibliography work, with reviews of lots of papers and articles. For this reason, it is firmly advised to look at the first part of the Appendix A, as there is a quick summary of what I got from all the most important papers.
- I peer-reviewed two papers unrelated to the field we were researching, of very different quality. It was great to see the procedure and the "other side", if I may say so.
- The encoding and testing of a so-called "phantom function block" in nxtStudio, that could lead to the Part 5 of IEC 61499

I also have lot of unfinished work:

- My contribution to an article reviewing several dynamical architectures and dynamical cyberphysical systems is incomplete. I don't know, and did not learn, how to sum up the specificities of each language.
- I never even began writing one about verification of such systems, even though I thoroughly understood and double-checked most results from [Vya17], as it can be seen in the [M1GIT].

- The observer FB project, about implementing a genre of FB called "Observer FB"[Roop15] that would be useful for effectively and easily verifying a large class of properties on FB, was never completed.

As possible future outcome of these works, there could be not only finishing any of those projects, but also be the usage of the closed-loop verification technique[Vya17], paired with either a language capable of modeling all types of dynamism, that could be obtained from my extension of Cimatti's ADL by adding hybrid systems, or a language that could only model some types of dynamism, such as Cimatti's ADL or DR-BIP.

6.2 Conclusion

I learned a lot during this stage. I had to learn the IEC 61499 norm, and the softwares 4DIAC and nxtStudio that implement this norm, and then code a number of systems to get used to how operate the softwares (and help an office mate with his own project), then implement the phantom FB. I coded in Python both the Iceblocks, their configurator and the Observer FB generator project. Indeed, what I was trying to do was already complex enough without to have to pay too much attention to the coding itself, and even though it may not be the fastest programming language, it comes with a number of libraries and takes care by itself of lots of minor stuff. During the coding of the Iceblock configurator, I also had to understand the networking interface of Windows.

On a personal note, I would underline that during my five-month internship, I realized that, in research, even though time is key to let ideas mature, deadlines are important as well to stimulate reflection. Overall, a constant dialogue with other researchers is truly the best safeguard.

I really want to thank my supervisor Valeriy Vyatkin for his support, his enthusiasm and our inspiring talks, during which he made lots of proposals and listened generously to my modest suggestions. I am also very grateful towards Sandeep Patil for how warmly he welcomed me in Luleå, towards Dmitrii Drozdov for what he taught me about programming, skiing and life in general, towards Marco Romanato for his friendship, towards Yulia Berezovskaia for her studious presence, and towards Sara Aronson for her hospitality.

7 Appendices

7.1 Appendix A : Bibliography

a) Most important references

[M1GIT] https://github.com/Lludion/M1-Internship-DPA
(You can see and test the files in this GitHub folder.)

[Cim18] A. Cimatti, I. Stojic and S. Tonetta, 2018. Formal Specification and Verification of Dynamic Parametrized Architectures. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 625–644. Springer, Cham (2018). [LINK TO THE ARTICLE]
See Part 3; it defines a DPA as there, and then reduces the problem of information flow in DPA to a problem and studies various problems.

[Atm19] D. Drozdov, U. Atmojo, S. Patil, et al. A Product-Driven software design pattern for distributed industrial automation system, 17th IEEE conference on industrial informatics (INDIN 2019), 2019, Helsinki-Espoo
There are precisions about what can be done with function blocks, and types of patterns that can be used with software design. A design pattern with five layers is introduced: production

sequence services recieves ordres and defines the choosable products, awareness services interact with non 61499 services (python, a socket ..) Order services implement the HMI (human machine interaction) like buttons, planning services organise the scheduling and other planning, execution services consist in FB interacting with actuators and sensors.

[Vya17] Buzhinsky, I. and Vyatkin, V., 2017. Automatic inference of finite-state plant models from traces and temporal properties. IEEE Transactions on Industrial Informatics, 13(4), pp.1521-1530
Great article, explaining how to build plant models (here, Moore automata) from negative and positive traces

[Pat17] D. Drozdov, S. Patil, V. Vyatkin, "Towards Formal Verification for Cyber-physically Agnostic Software: a Case Study", International Annual Conference of IEEE Industrial Electronics Society IECON'17, Beijing, 2017
Case study of an elevator, explains lots of things about the two types of time in the IEC 61499 architecture (E_CYCLE and E_DELAY)

[Buz17] I. Buzhinsky, V. Vyatkin, "Explicit-State and Symbolic Model Checking of Nuclear I&C Systems: A Comparison", 43rd International Annual Conference of IEEE Industrial Electronics Society IECON'17, Beijing, 2017
A comparison between SPIN (explicit-state) and NuSMV (symbolic state); the reason why I chose NuSMV in the end; Note that I&C stands for Instrumentation and Control

[Bra04] Bradbury, J.S. (2004), Organizing Definitions and Formalisms of Dynamic Software Architectures. Queen's University, Technical Report 2004-477
*I wrote a presentation expliciting notions described in this paper **here**; notably, it defines various types of dynamism such as programmed dynamism and ad-hoc dynamism.*

[DRBIP] El Ballouli, R., Bensalem, S., Bozga, M., Sifakis, J.: DR-BIP - programming dynamic reconfigurable systems. Tech. Rep. TR-2018-3, Verimag Research Report
introduction and definition of DR-BIP; DR-BIP approach [62] can encompass perfectly both ad-hoc and adaptive dynamism, while still being able to describe subtle evolution and synchronization of the motifs (that are architectural patterns and groups), components and states within the components. Another characteristic of DR-BIP is that this model is directly written as an usual programming language.

b) Other references

[End94] M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In Proceedings of 12^{t}h Brazilian Symposium on Computer Networks, pages 175–187, 1994

[Roop15] Model-Driven Design Using IEC 61499 : A Synchronous Approach for Embedded and Automation Systems. Li Hsien Yoong, Partha S. Roop, Zeeshan E. Bhatti, Matthew Ming Yen Kuo, 2015. Springer International Publishing Switzerland eISBN: 978-3-319-10521-5 DOI : 10.1007/978-3-319-10521-5 ISBN: 978-3-319-10520-8

[DDL] A.Platzer, Differential Dynamic Logic for Hybrid Systems, 2008. DOI 10.1007/s10817-008-9103-8

[Ho96] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho, Automatic Symbolic Verification of Embedded Systems, 1996.

[Uly12] Ulyantsev and F. Tsarev, "Extended finite-state machine induction using SAT-solver," in Proc. 14th IFAC Symp. Inf. Control Problems Manuf.,2012, pp. 512–517

[Oak03] Oaknine, B.H. Kragh, Theobald, Clarke, 2003 CEGAR for hybrid systems https://doi.org/10.11
pages 107 - 120 were the ones used. Observer FB are described.

7.2 Appendix B: Clarification on Input-Parametrized Architectures

If a component is c_{pi}, we note $\pi(c_{pi}) = p$ and $xi(c_{pi}) = i$.

Remark : One does not propose such a definition of S_μ:

$$\begin{aligned} S_\mu \colon C_\mu &\to \bigcup_{C_{\mu p} \in C_\mu} C_{\mu p} \\ C_{\mu p} &\mapsto c_{pi} \end{aligned}$$

where $\forall C_{\mu p} \in C_\mu$, $S_\mu(C_{\mu p}) \in C_{\mu p}$.

This proposition would foster the following definition of DIRECT REACHABILITY :

<u>Definition:</u> Given an input-parametrized architecture $A = (\mathfrak{I}, \mathscr{V}, \mathscr{P}, In, \psi, \phi)$ and an instantiated configuration $G_1 = (C_\mu, E_\mu, S)$, an instantiated configuration $G_2 = (C_\rho, E_\rho, S_\rho)$ is said to be DIRECTLY REACHABLE from G_1 iff:

- $C_\mu = C_\rho$ meaning that the same components/states are activated (no component reconfiguration)
- $E_\mu = E_\rho$ meaning that the same connections are activated (no connection reconfiguration)
- $\xi(S_\mu(C)) = \xi(S_\rho(C))$ (meaning that the two activated states are in the same component)
- $\phi_{\pi(S_\mu(C)),\pi(S_\rho(C))}(\xi(S_\mu(C)))$

However, such a proposition would be nonsensical, as I finally chosen the 'component vision', meaning that the $C_{\mu P} \in C_\mu$, that are the translations by μ of the $I \in \mathfrak{I}$, are always as many as types of states per machine existing in the modelization. Reciprocally, $|I_{\mathscr{P}}|$ is the number of machines. The size of this C is only driven by $\mu(I_{\mathscr{P}})$, and the number of states by component is $|\mathscr{P}|$. As a result, choosing one state per $C \in C_\mu$ means choosing only one state per $P \in \mathscr{P}$, and not one state per machine (i.e. per index). With the example of the serial pneumatic cylinder, it would have meant having three active state, regardless of the number of components.

7.3 Appendix C : Important GitHub Files

The most important files are :

Dynamic_Software_Architectures.pdf

Chapter 1. of the written documents

7.4 Appendix D : Meta-information

As explained **here**, here are the meta information:

General bibliography & drafts of articles 50% , transforming an ADL into a language for dynamic CPS description : 16%, 4DIAC & nxtStudio implementations 16%, Iceblock coding 16% , Observer project 2%