# Applying the Model-Driven Architecture Approach to Dynamic Structure Applications

Min Zhu, Clément Foucher, Vincent Albert, Alexandre Nketsa

# APPLYING THE MODEL-DRIVEN ARCHITECTURE APPROACH TO DYNAMIC STRUCTURE APPLICATIONS

Min ZHU, Clément FOUCHER, Vincent ALBERT, Alexandre NKETSA
LAAS-CNRS, Université de Toulouse, CNRS, UPS, Toulouse, France
Email: {Min.Zhu, Clement.Foucher, Vincent.Albert, Alexandre.Nketsa}@laas.fr

**KEYWORDS**
Model Driven Architecture, Reconfigurable Architectures, Meta-Model, Simulator

**Abstract**

Model-Driven Architecture (MDA) is a system engineering approach which consists in separating the model description from the execution platform. It allows building a model without detailed knowledge of the target platform, as well as retargeting the execution platform without changing the model itself.
We present a meta-model called Partial Reconfigurable DEVS (PRDEVS) that is able to represent dynamic structure changes of a model. We base our approach on the DEVS formalism, which is modular and hierarchical. Our description paradigm differs from the previous DEVS-based dynamic meta-models in that it explicitly deals with adding and removing components. This approach is closer to the general reconfigurable embedded system design methodology. Both a software and a FPGA-based hardware platform are considered as dynamic execution platforms.

## INTRODUCTION

The Discrete Event System Specification (DEVS) formalism introduced by Zeigler et al. (2000) is a strong mathematical foundation for specifying hierarchical and modular models. The DEVS formalism allows to build discrete event systems and provides algorithms for simulation. DEVS models are made of atomic components, which define a behavior, and coupled components which can hold several other components and describe the way they are connected. As the initial DEVS formalism was not designed to handle structure changes, either in model composition or communication, various extensions were proposed to address these dynamic systems. However, our point of view on existing formalisms which allow to describe reconfigurable systems is that they are either too high level, making it difficult to apply to real systems, or too deeply linked to the execution platform.
The Model-Driven Architecture (MDA) approach by Object Management Group (2016), derived from Model-Driven Engineering (MDE), consists in separating the application model description from the execution platform. This brings various benefits, such as allowing the teams working on an application to be independent from the ones working on the platform, or enabling deploying an application built from a single model on various platforms. A complete MDA specification consists in a Platform-Independent Model (PIM), one or several Platform-Dependent Models (PDM), and sets of interfaces correspondence to allow building a Platform-Specific Model (PSM) by merging PIM with PDM, as depicted on Figure 1.

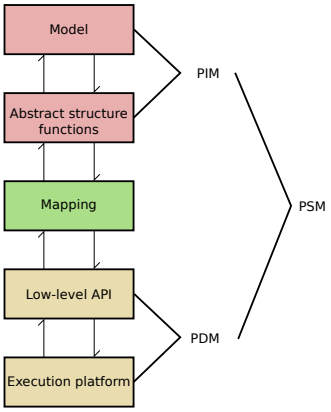



Figure 1: MDA Structure of PRDEVS

In this paper, we propose a DEVS-based PIM formalism able to describe applications whose structure can evolve dynamically. The aim of this formalism is to comply with hardware-reconfigurable architectures such as FPGAs, without however restricting the execution to these platforms. We thus take in consideration the generally observed approach used when building such systems, but remain sufficiently high level to target any platform supporting dynamic architecture change of the application, such as software-based implementation. We also introduce PDM candidates in order to show the compliance of the PIM with such architectures.
First, we present the existing work related to DEVS and its extensions. Then the background of DEVS formalism and reconfigurable hardware are presented. Afterwards, the PRDEVS model at PIM level is presented with its syntax and semantics. A software oriented implementation with its possible PSM is then presented. Finally, we conclude this article and present a view of what remains to be done.

## RELATED WORK

Zeigler et al. (2000) initially introduced the DEVS formalism in the late 70's as a way to build models with a discrete-event approach using a mathematically defined formalism. DEVS was later extended with Parallel DEVS (PDEVS), and we now reference the initial DEVS formalism as Classic DEVS (CDEVS). In this article, the acronym DEVS thus refers to the general DEVS ecosystem rather than to the original CDEVS formalism.

DEVS is a hierarchical set of components of two kinds: atomic components define a behavior while coupled components gather and link other components, either atomic or coupled. The DEVS formalism is inherently static, and dynamic structure behavior can only be emulated, e.g. using a selector to enable or disable models over time. Several extensions have been proposed aiming at dynamically adapting the models structure during the simulation.

DSDEVS, defined in Barros (1997), is based on a 4-tuple network structure where atomic components can connect directly with other atomic components by a set of influencers $I$. The network executive $\chi$ is a specific component whose state represents the network structure. The $\gamma$ function allows to obtain the network structure from the current state of $\chi$. $\chi$ thus takes responsibility for all changes of model structure, meaning that components in the model can not take decision on structural adaptation. DSDE (Barros (1998)) is a parallel version of DSDEVS.

The principle of dynDEVS as described in Uhrmacher (2001) is that each atomic component has its own model transitions function $\rho_\alpha$ which controls its own structural transformation. At coupled component level, the equivalent function is the network model transition function $\rho_N$. However, dynDEVS assumes a static set of ports which is not adapted for most of the dynamic applications. Uhrmacher later developed $\rho-$DEVS, a dynDEVS variation supporting dynamic ports.

The formalisms mentioned above stay at theoretical level: they propose an abstract simulator and can be adapted to an actual execution environment.

There are recent works like RecDEVS which consider hardware models of computation (MoC) together with DEVS. RecDEVS (Madlener (2013)) proposes a model based on DEVS for final use on reconfigurable hardware like FPGA. In RecDEVS the system executive $C_\chi$ is in charge of the structure changes. However RecDEVS takes into account some hardware specificities from the beginning, like component communication relying on a bus structure with an address notion. This limits the model to a use on the target platform defined by RecDEVS. Thus, there is no separation between PIM and PDM. There are other limits on the meta-model itself, like the fact that a component deletion can only be triggered by the component itself. Eventually, there is no final implementation on FPGA, the workflow only goes to SystemC simulation.

## BACKGROUND

### Parallel DEVS (PDEVS)

PDEVS (Zeigler et al. (2000)) is the root formalism for DEVS extensions dealing with parallelism. It defines two kinds of components: atomic components, which are the base elements defining a behavior, and coupled components, which gather various other components and define their relationships. PDEVS allows different components to evolve simultaneously and provides resolution mechanisms to deal with conflicting simultaneous events.

*Atomic Models*

PDEVS defines an atomic component as an indivisible unit implementing a behavior. It can evolve in reaction to an external event (external transition), or when a timeout occurs (internal transition). The formal definition is as follows:

$$M =< X, Y, S, s_0, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, \tau > \text{ where}$$

$X$ $= \{(p, v) \mid p \in InPorts, v \in X_p\}$ is the set of input ports and values, where
  $InPorts$ is the set of input ports
  $X_p$ is the set of allowed input values for port $p$

$Y$ $= \{(p, v) \mid p \in OutPorts, v \in Y_p\}$ is the set of output ports and values, where
  $OutPorts$ is the set of output ports
  $Y_p$ is the set of possible output values for $p$

$S$ is the set of sequential states

$s_0$ is the initial state of the component

$\delta_{ext}$ : $Q \times X \rightarrow S$ is the external state transition function, where
  $Q = \{(s, e) \mid s \in S, 0 \leq e \leq \tau(s)\}$ is the set of total states, with $e$ the time elapsed since latest transition

$\delta_{int}$ : $S \rightarrow S$ is the internal state transition function

$\delta_{con}$ : $Q \times X \rightarrow S$ is the confluent transition function

$\lambda$ : $S \rightarrow Y$ is the output function

$\tau$ : $S \rightarrow \mathbb{R}^+_{0,\infty}$ is the time advance function

When an event $X$ occurs on an input port, the external transition function $\delta_{ext}$ is called which may result in a state change. The time advance function $\tau$ associates a time to each state which, when reached, triggers the output function $\lambda$ then the internal transition function $\delta_{int}$. Note that $\tau$ accepts both 0 and $\infty$ as values. When simultaneous external and internal events occur, the confluent function $\delta_{con}$ is called instead of $\delta_{int}$ or $\delta_{ext}$ to solve the conflict. $\delta_{con}$ can be as simple as calling $\delta_{int}$ or $\delta_{ext}$, which is a way of prioritizing between these two functions, or can be a totally different function.

*Coupled Models*
A coupled component is a way of linking other components. Externally, it behaves like an atomic component and thus can be used in another coupled to form a hierarchical model. Its definition is:

$$N =< X, Y, D, \{M_d\}, EIC, EOC, IC > \text{ where}$$

| | |
|---|---|
| $X, Y$ | as defined for atomics |
| $D$ | is the set of components names |
| $\{M_d\}$ | is the set of components in this coupled, with $d \in D$ |
| $EIC$ | is the external input coupling function |
| $EOC$ | is the external output coupling function |
| $IC$ | is the internal coupling function |

The three coupling functions directly link ports between them:

| | |
|---|---|
| $EIC$ | links $p_N \in InPorts_N$ to $p_d \in InPorts_d, d \in D$ |
| $EOC$ | links $p_d \in OutPorts_d, d \in D$ to $p_N \in OutPorts_N$ |
| $IC$ | links $p_a \in OutPorts_a$, $a \in D$ to $p_b \in InPorts_b$, $b \in D, a \neq b$ |

**Parallel Dynamic Structure DEVS (DSDE)**

DSDE (Barros (1998)) defines a specific component, $\chi$, whose state encodes the structure of the network, i.e. the current network structure can be obtained at any time from $\chi$ state using the structure function $\gamma$. A transition of $\chi$ thus can represent a change of the network structure. The DSDE component is defined as:

$$DSDE_N =< X_N, Y_N, \chi, M_\chi > \text{ where}$$

| | |
|---|---|
| $N$ | is the network name |
| $X_N, Y_N$ | $\equiv X, Y$ in PDEVS |
| $\chi$ | is the name of the dynamic network executive |
| $M_\chi$ | is the model of the executive $\chi$ |

The model of the executive $\chi$ is an extended definition of an atomic model defined as:

$$M_\chi =< X_\chi, S_\chi, s_{0,\chi}, Y_\chi, \gamma, \Sigma^*, \delta_\chi, \lambda_\chi, \tau_\chi > \text{ where}$$

| | |
|---|---|
| $\gamma$ | $: S_\chi \to \Sigma^*$ is the structure function |
| $\Sigma^*$ | is the set of network structures |

According to this definition, the set of components is defined, but their state is not. Barros thus defines the new components states after a $\chi$ transition to be equal to the same components state before transition (plus time advance) if the component existed, or to be the initial state if the component didn't exist.
In this definition, $\chi$ is the only component allowed to change the network structure. Moreover, the connections between the components of the network are also defined by $\chi$ state, i.e. a simple change of connector without affecting the atomic components themselves must be treated as a $\chi$ transition.

**RecDEVS**

$$N_{Rec} =< X_{ext}, Y_{ext}, D, C_\chi > \text{ where}$$

| | |
|---|---|
| $D:$ | Set of all available DEVS components |
| $C_\chi:$ | is the network executive which is a DEVS atomic |

RecDEVS defines an unique identifier $ID$ for each component. The creation of new RecDEVS components consists of a fixed sequence of messages as follows:

* if the component $C^{ID}_{orig}$ wants to create a new component of type $d \in D$, it sends a message ($C^{ID}_{orig}$,$C_\chi$,(new d)) to the network executive.
* $C_\chi$ receives the message and performs an external transition $\delta_{ext}$. This will create a new RecDEVS component $C^{id}_d$ and add it to the list of instantiated components
* A confirmation message ($C_\chi$, $C^{ID}_{orig}$, (confirm $C^{id}_d$ )) with the address of the new component is then sent to the originator.
* Starting from the reception of the confirmation message, the originator can address the newly created component.

**FPGA**

A Field-Programmable Gate Array (FPGA) is an integrated circuit designed to be configured to form a complex digital circuit. The majority of FPGA architectures carry out combinatorial logic using Lookup Tables (LUTs) (Koch (2012)), associated to flip-flops to form sequential circuits. Several LUTs and flip-flops form a base reconfigurable resource (e.g. configurable logic block (CLB) in Xilinx technology) which is the smallest reconfigurable unit in a FPGA. Configuration of the underlying structure in the FPGA can be stored using various technologies. The Static RAM-based FPGA is a common FPGA architecture. There are also flash-based FPGAs. A bitstream is the configuration data to be loaded on board to implement the desired logic.
Partial Reconfiguration (PR) (Feist (2012)) is the ability to dynamically modify the architecture hosted on the FPGA by loading a partial bitstream (i.e. a configuration of a specific area of the FPGA) while the remaining logic continues to operate without interruption.

**PRDEVS PIM: SYNTAX AND SEMANTICS**

We propose a PIM syntax based on PDEVS and inspired by RecDEVS. The platform-independent model is, as stated from the name, a model which is built to represent an application, without necessary knowledge of the simulation environment or of the target platform that will run the application or the application simulation. Thus, our PIM syntax must be able to represent dynamic structure models, but it shouldn't assume anything about how the structure changes will actually be

applied. This is the first difference with RecDEVS, as the RecDEVS meta-model makes no distinction between PIM and PDM. The target architecture is assumed from the model definition, notably with the use of address notions within the models.

### PRDEVS PIM Abstract Syntax

A PRDEVS is a model which contains all required information about components, structure, and allows for structural changes. Though we use PDEVS and RecDEVS as a base reference, we slightly rewrite some definitions to clarify specific points. A major difference with RecDEVS is that we do not use a specific component like $C_\chi$ which stores the network structure in its state: we directly manipulate the sets, adding and removing elements. The first motivation for this approach is to be closer to the current engineering approach to describe dynamic systems. This is slightly equivalent to the software notion of new/delete instructions for object manipulation. Moreover, this way of doing offers the ability to reach structure states which may not have been predicted when first designing the system, allowing for auto-adapting systems to be more flexible.

Concerning the $D$ set, on one hand PDEVS defines $D$ to be *the set of components names*, i.e. a list of all the names of the components *inside* the coupled. On the other hand, RecDEVS definition of set $D$ is "a list of available component names", and this set is compared to a list of components *types*. They both define the set $\{M_d \mid d \in D\}$, which contains the components themselves. In our component sets-based description, we rather directly manipulate the components sets, and we think this description is redundant, as one can be obtained from the other. So we chose to merge these two sets, so that the $D$ set directly contains the components themselves. Instead of storing the names of the components, we rather use the notions of *identifiers* and *types*.

The *identifier* follows the notion introduced by RecDEVS where an identifier $ID \in \mathbb{N}$ is attributed to each component. For the *type* notion, we can make the connection with object-oriented programming, where there can be various instances of a *class*, we call *objects*. Here, the notion of *type* is equivalent to *class*, i.e. it defines a component structure and initial state, but there can be various components ($\equiv$ *objects*) sharing the same type with a different state. The identifier is then used to differentiate the components. We use here a definition close to RecDEVS but formalize the notation: we use $T$ as the list of defined types, i.e. a list of components types which can be instantiated.

A PRDEVS component then has an identifier, which is unique and dynamically defined, and a type, which can be shared.

*Main PRDEVS Component*

The dynamic structure ability relies on a library of available components, each being of a specific type, which can be added to the system. The library is defined as $L = \{C_t \mid t \in T\}$. Components in the library expose a null identifier, as it is defined on instantiation.
Components in use still have a type $t \in T$, but also an identifier $id$, and are noted $C_t^{id}$. The notation can be simplified to $C^{id}$. Unlike RecDEVS, we do not restrict $id \in \mathbb{N}$: although a PSM implementation will probably have to impose such a restriction, the PIM doesn't require so. We thus define an arbitrary set $ID$ which contains the allowed identifiers.

$$PRDEVS =< L, C^{Top} > \text{ with}$$

$L$ $= \{C_t \mid t \in T\}$, library of available components
$C^{Top}$ a coupled containing the application structure

To have a clear and simple definition, we do not include the sets $T$ and $ID$ in the definition. Indeed, $T$ can be retrieved from $L$ using the definition $T = \{t \mid C_t \in L\}$, while $ID$ can be any arbitrary set. We also define the set $ID_{PRDEVS}$ which contains all the identifiers of the components in $C^{Top}$, regardless of the hierarchy.

*Coupled Component*

A coupled component will be very alike PDEVS definition:

$$N^{nid} =< X, Y, D, EIC, EOC, IC > \text{ with}$$

$D$ $= \{C_t^{id} \mid t \in T, id \in ID\}$ the set of components contained in the coupled

All of the sets defined here can be linked to a specific component by displaying its identifier, e.g. $X^{nid}$ refers to the set of inputs $X$ of coupled component $C^{nid}$. A coupled component $C^{nid}$ does not have a specific type, as the component structure can change during execution when a component is added to or removed from $D^{nid}$.
For convenience, we define a few additional sets:

$ID^{nid}$ $= \{id \mid C^{id} \in D^{nid}\}$, the set of all identifiers of components in the coupled whose identifier is $nid$
$D_N$ the set of all coupled in the PRDEVS
$ID_N$ $= \{id \mid C^{id} \in D_N\}$, the set of coupled identifiers

*Atomic Component*

As our formalism deals with structure changes, the atomic components definition adds a structure change function to the PDEVS atomic formalism:

$$M_t^{mid} =< X, Y, S, s_0, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda_{SC}, \lambda, \tau > \text{ with}$$

$\lambda_{SC}$ $: S \to SC$ the structure function with
$SC = \{addComponent(), removeComponent(), addPort(), removePort(), addConnection(), removeConnection()\}$ the list of structure change functions.

The remark on sets identifiers applies to atomic components, e.g. $S^{mid}$ is the set of states $S$ of component $C^{mid}$.

As for coupled, we define the following sets:

$D_M$ the set of all atomics in the PRDEVS
$ID_M$ $= \{id \mid C^{id} \in D_M\}$, the set of atomics identifiers

*PRDEVS Implementation*

The $L$ set can be represented as a list of available models, which are defined by the modeler or provided by a predefined library. The list must match a type name and a component. This way, when adding a component to a model, only its type must be provided, and the list is used to retrieve model information.

*Components Common Characteristics*

The coupled and atomic models share two properties: the Ports set and the identifier. Any number of ports can be present on a component.

The main difference with RecDEVS is that they use the identifier to manage communication between components on a message-passing paradigm. As we separate the implementation from the high-level model, we do not presuppose of a communication scheme in the PIM.

The ports of a component are implicitly defined by the couples $(p, v)$ of $X$ and $Y$ sets. However, a specific definition can be derived to formally identify the ports as mathematical objects. This representation of ports as independent objects, i.e. not only as names belonging to a set, and whose allowed values are defined by the $X$ and $Y$ sets, is easier to manipulate.

The ports of the components must be uniquely identified, but only among a component. Thus, the name of the port on the component is sufficient to identify it uniquely, as long as the component itself has a unique identifier. Moreover, a port has a direction (input/output), and a set of available values. We then introduce a definition of what is a port:

$$P^{id}_{Name} =< Name, Dir, Type > \text{ with}$$

$id$ the identifier of the component
$Name$
$\in InPorts^{id} \cup OutPorts^{id}$
$Direction$
$\in P_{dir} = \{in, out\}$
$Type$ $\in P_{type}$

The set of allowed types $P_{type}$ can be defined as a set of allowed definition sets, and will most likely contain $\mathbb{N}$, $\mathbb{R}$, $\mathbb{B} = \{True, False\}$, etc.

Concerning the connection between two ports, it must define a source and a sink. The data type does not have to be recorded, but type match between source and sink should be checked when the connection is created.

*Coupled Models*

The coupled models are actually sets of sets: a set of components, and a set of connections between ports.

By using the tree representation for holding the structure, the set of components $D$ is represented by the children of the node. Using the previous definition of ports, the $X$ and $Y$ sets can be better represented as sets of ports. Thus, $X$ and $Y$ will be merged into a list called "Ports" containing ports as defined previously. The remaining three sets $EOC$, $EIC$ and $IC$ are represented by a list of two elements: a source and a sink.

*Atomic Models*

The atomic models are leafs of the model tree, so they can be defined as in the abstract syntax.

**Structure Change Functions Exposed by the Simulator**

Unlike RecDEVS, we do not restrict which component can call a structural change function. Indeed, RecDEVS states that a component can only delete itself, not another component. The main justification provided is that it avoids accidentally deleting a component which is still in use. By only allowing self-deletion, the component can announce its own deletion to linked components before committing deletion. But this approach doesn't seem to be a good answer to this issue. Indeed, deleting a component which is still in use in an application may be a conception error. Restricting the remove call to the component itself does not solve the case where the component itself is badly defined, and forget announcing its deletion to some of related component. Imposing such a constraint do not avoid errors, so the restriction is irrelevant. We believe the application correctness is up to the modeler, and to avoid such errors, applications should be checked for correctness, e.g. using formal methods.

By allowing any component to call structure change functions, we let the modeler decide how to handle its structure changes: all components can be autonomous and directly trigger the functions, or there can be one or more components in the model which are in charge of the structure, only them being able to call these functions.

Most functions defined here could possibly fail in some circumstances, but we do not want to handle exceptions or errors in this early definition, so we assume their use is made with correct parameters. Error-checking will be part of future work.

Structure change functions have a different priority level than other messages in the simulator: the simulator has a list of pending SC tasks and the list is executed only at the end of an imminence cycle. As a first approach, we chose to execute all SC functions in zero time relatively to the simulator, i.e. the simulator is paused while the structural changes are carried on. In future work

however, we are planning to allow structural changes to be applied while the simulation is running in order to allow taking full advantage of hardware partial reconfiguration technology. This will require structural checks on the model such as making sure that a newly added component will not be required for simulation until it is fully operational. This can be carried on by separating the SC function call from its return, obtaining of the new identifier on a separate external transition of the component which called the add function.

• addComponent : $T \times ID_N \rightarrow ID$
Adds a new component into an existing coupled component. The new component type and the hosting coupled ID are provided as parameters. The identifier of the new component is returned. The abstract function $getAvailableId : \emptyset \rightarrow ID$ determines an available identifier in $ID$. Abstract function $getNewComponent : T \rightarrow D$ returns a new component from the library matching the given type, while $getExistingComponent : ID_{PRDEVS} \rightarrow D$ returns an existing component from its identifier, as displayed on Algorithm 1.

**input :** $t \in T$; $id_{host} \in ID_N$
**output:** $id \in ID$
**Data:** $newId \in ID$; $newC \in D_M$; $hostC \in D_N$

newId ← `getAvailableId()`
newC ← `getNewComponent(`$t$`)`
newC.id ← newId
hostC ← `getExistingComponent(`$id_{host}$`)`
hostC.D ← hostC.D ∪ {newC}
return newId

**Algorithm 1:** addComponent procedure

• removeComponent : $ID_{PRDEVS} \rightarrow \emptyset$ Removes the existing component, whose identifier is passed as a parameter, from the PRDEVS. The abstract function $getParentComponent : D \rightarrow D_N$ returns the parent component of a model, as displayed on Algorithm 2.

**input:** $id_{removed} \in ID_{PRDEVS}$
**Data:** $rem_C \in D, hostC \in D_N$

remC ← `getExistingComponent(`$id_{removed}$`)`
hostC ← `getParentComponent(`$remC$`)`
hostC.D ← hostC.D \ {remC}

**Algorithm 2:** removeComponent procedure

• addPort : $ID_N \times Name \times P_{type} \times P_{dir} \rightarrow \emptyset$
Adds a port with name $Name$ to a coupled component whose identifier is provided, with the associated *type* and *direction*, as displayed on Algorithm 3.

**input :** $id \in ID_N$; $p_n \in Name$; $p_t \in P_{type}$; $p_d \in P_{dir}$
**Data:** $new_{port} \in Port$; $hostC \in D_N$

$new_{port} \leftarrow (p_n, p_t, p_d)$
hostC ← `getExistingComponent(`$id$`)`
hostC.port ← hostC.port ∪ $\{new_{port}\}$

**Algorithm 3:** addPort procedure

• removePort : $ID_N \times Name \rightarrow \emptyset$
Removes the port name $Name$ from the component whose ID is provided. The abstract function $getPort : D_N \times Name \rightarrow Port$ gets an existing port from a coupled component, as displayed on Algorithm 4.

**input :** $id \in ID_N$;$p_n \in Name$
**Data:** $hostC \in D_N$; $removed_{port} \in Port$

hostC ← `getExistingComponent(`$id$`)`
$removed_{port}$ ← `getPort(`$hostC$,$p_n$`)`
hostC.port ← hostC.port \ $\{removed_{port}\}$

**Algorithm 4:** removePort procedure

• addConnection : $ID_N \times Name \times ID_N \times Name \rightarrow \emptyset$
Adds a connection between two ports. The ports are referred to using the combination of the component identifier and the port name. The $Z_{i,d}$ definition must be respected, i.e. the two components must be in the same coupled, or one of the two component must be a coupled and the other one a component inside the coupled, and one must be an input and the other an output. Moreover, the definition interval *type* must match between the two ports, as displayed on Algorithm 5.

**input :** $id_1 \in ID_{PRDEVS}$; $p_n1 \in Name$;
$id_2 \in ID_{PRDEVS}$; $p_n2 \in Name$
**Data:** $hostC1 \in D_N$; $hostC2 \in D_N$;
$connection \in PortConnection$

$hostC1$ ← `getParentComponent(`$id_1$`)`
$hostC2$ ← `getParentComponent(`$id_2$`)`
$connection \leftarrow \{(id_1, p_n1), (id_2, p_n2)\}$
**if** $hostC1.id$=$hostC2.id$ **then**
  $hostC1.IC = hostC1.IC \cup \{connection\}$
**else if** $hostC1.id$=$id_2$ **then**
  $hostC1.EOC = hostC1.EOC \cup \{connection\}$
**else if** $hostC2.id$=$id_1$ **then**
  $hostC2.EIC = hostC2.EIC \cup \{connection\}$

**Algorithm 5:** addConnection procedure

• removeConnection : $ID_N \times Name \times ID_N \times Name \rightarrow \emptyset$
Removes a connection between two ports using the same notation as the previous function, as displayed on Algorithm 6.

**input :** $id_1 \in ID_{PRDEVS}$;$id_2 \in ID_{PRDEVS}$;$p_n1 \in Name$;$p_n2 \in Name$
**Data:** $hostC1 \in D_N$; $hostC1 \in D_N$;
$connection \in PortConnection$

$hostC1$ ← `getParentComponent(`$id_1$`)`
$hostC2$ ← `getParentComponent(`$id_2$`)`
$connection \leftarrow \{(id_1, p_n1), (id_2, p_n2)\}$
**if** $hostC1.id$=$hostC2.id$ **then**
  $hostC1.IC = hostC1.IC \setminus \{connection\}$
**else if** $hostC1.id$=$id_2$ **then**
  $hostC1.EOC = hostC1.EOC \setminus \{connection\}$
**else if** $hostC2.id$=$id_1$ **then**
  $hostC2.EIC = hostC2.EIC \setminus \{connection\}$

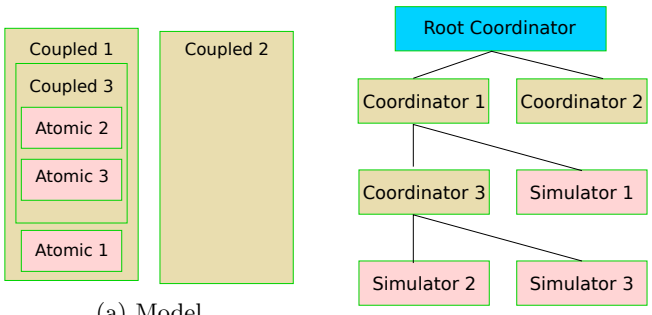
**Algorithm 6:** removeConnection procedure

## AN EXAMPLE OF PRDEVS PDM AND PSM

While our final aim is to implement PRDEVS on reconfigurable hardware, we choose to first develop a software PDM using well-known tools. This is intended as a proof of concept to check that the simulator structure is reliable.

### Software PDM Definition

We propose a PDM for software simulation that implements PRDEVS abstract syntax. This implementation is based on Zeigler's abstract simulator described in Zeigler et al. (2000). This simulator deals with a static hierarchy of components and we add the dynamic SC functions to treat SC-messages from $\lambda_{SC}$.
The simulator described by Zeigler uses a hierarchical tree of models, which has a coordinator object for each coupled model and a simulator object for each atomic model. To each of the simulator objects, a model object describing the structure of the represented atomic is associated. There is a single root coordinator which lead the hierarchical tree. The root coordinator contains a list of imminent models and their next event time. This list is updated at the beginning of each event step.
Under root coordinator are coordinators which can be the parents of coordinators or simulators which are the leaves of this hierarchical tree. A correspondence from model to simulation can be seen in Figure 2. We name this a one-to-one correspondence process. This is part of the initialization phase.



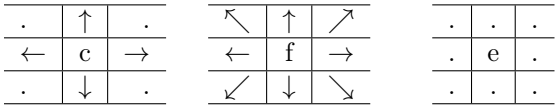

(a) Model (b) Controller

Figure 2: One-to-One Correspondence

As a first approach, a flat representation is chosen for our implementation of PRDEVS. That is to say, the levels of hierarchy are ignored as if all the atomics were directly instantiated in $C^{TOP}$. A list of available components $L$ is held in the root coordinator. A simulator or a coordinator which correspond to a type can be created using the SC-functions.
There are four types of messages sent between simulators and/or coordinators during simulation: X-message, Y-message, *-message and SC-message which correspond respectively to $\delta_{ext}$, $\lambda$, $\delta_{int}$ and $\lambda_{SC}$ of atomic components. The first three messages are defined as Zeigler's: imminent models are chosen based on their minimal next event step and the imminent models triggers *-message. If the conditions to trigger $\lambda$ on the current state are met, a Y-message is then integrated into the Y-message bag. The bag is sent to the target model and is received as a X-message at the end of each event cycle. The model which received X-message will move to upcoming state and wait for next cycle. SC-message, in some aspect similar to Y-message, will build a SC-message bag and the dynamic SC functions are executed at the end of each event cycle. The SC-message is then treated in zero-time compared to the simulation time.

### Use Case Definition

We apply PRDEVS syntax by creating a PSM simulation implementing a game: Within a $size \times size$ grid co-exist three types of players: chicken, fox and egg. Each cell can hold only one player and players move under certain rules, as shown in Figure 3: chickens can move randomly around in four direction while foxes can move randomly around in all eight directions; eggs can not move. There is a rules model recording the position of all players and judging if each move is authorized.
Each round, all players are imminent and move simultaneously. If a chicken reaches another chicken, an egg will be laid randomly around and it stays at the same position. If a fox reaches a chicken, the chicken is eaten and its cell occupied by fox. The game ends when there are no chicken any more or if foxes are blocked by eggs.
Chickens and foxes components communicate using their ports. $valid$ is an input port and $askAvailability$ is an output port.
$P_{valid} =< valid, in, \{isChicken, isFree, else\} >$
$P_{askAvailability} =< askAvailability, out, (positionX \in size, positionY \in size) >$
Players calculate their destination themselves and verify with the rules component before moving.
The state machine for the Chicken model is as shown in Figure 4. The initial state of a chicken is $S1$. When the chicken component is imminent, it receives a *-message to execute the internal transition and randomly defines the desired destination. It moves to state $S2$. Then an output is sent to verify the availability of this position. It moves to state $S3$ and wait for an input. The Y-message arrives to the rules model which responds ac-




(a) Chicken (b) Fox (c) Egg

Figure 3: Moving Rules for Players

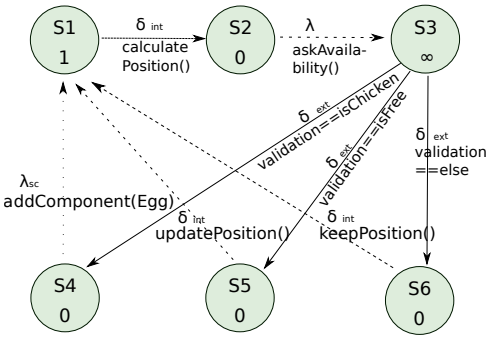



Figure 4: Internal View of the State Machine of Chicken

cording to the availability. Depending on this response, the chicken model moves to state $S4$, $S5$ or $S6$ and then the SC-function or the internal transition is called.
When the SC-function addComponent is triggered, the simulator stores the call. After simulation cycle is over, the root coordinator copies the Egg library object into the $C^{Top}$ component. After what, the simulation cycle resumes.
The game starts with an initial numbers of players, an example is presented on figure 5a. After the simulation runs, we found the result as shown in figure 5b.

| | | | |
|---|---|---|---|
| c | c | . | . |
| . | . | f | . |
| . | . | . | . |
| . | e | . | . |

(a) Game Starts

| | | | |
|---|---|---|---|
| . | . | . | . |
| f | e | . | . |
| . | . | . | . |
| . | e | . | . |

(b) Game Ends

Figure 5: Example of game turns

## Hardware PDM Overview

A hardware PDM is being formalized, relying on several reconfigurable areas and two buses: one control bus and one data bus. There is a correspondence table for each area and its address. Each reconfigurable area will have a type and certain limits since their hardware resources can be different.
Reconfigurable areas can store a component model. An API (Application Programming Interface) for the SC-functions can be implemented at software level in specific area containing a processor. The API is able to match the configuration of the FPGA, such as which type of reconfigurable areas are suitable for which type of component model.

## CONCLUSION AND FUTURE WORK

In this article, we presented a system engineering approach using formal modeling which aims at supporting dynamic architectures. This approach leads to a separation between the model and the final application platform. We presented a PIM model which can adapt its structure dynamically, both component- and connection-related. With the formal SC-functions presented in this article, a PIM level of PRDEVS is defined. A possible PDM and PSM is introduced with application on software by a game. With this practical example of PRDEVS syntax, the feasibility of the dynamical formal model is verified. However in practice, the SC-functions are only applied at the end of each event cycle. One future work will be to integrate the SC-functions during the simulation without pausing the other components, in order to allow extending beyond simulation purposes and take advantage of PR.
Finally, the main objective will be the FPGA-based implementation of a use case defined at PRDEVS PIM level. This will require real-time handling, and partial reconfiguration scheduling to allow reconfigurations to be completed before using the component, where the structure change was carried on in zero time in simulation.

## References

Barros F.J., 1997. *Modeling Formalisms for Dynamic Structure Systems. ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7, no. 4, 501–515. ISSN 1049-3301. doi:10.1145/268403.268423.

Barros F.J., 1998. *Abstract simulators for the DSDE formalism.* In *1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274)*. IEEE, vol. 1, 407–412. doi:10.1109/WSC.1998.745015.

Feist T., 2012. *Vivado design suite. White Paper*, 5.

Koch D., 2012. *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*, vol. 153. Springer Science & Business Media.

Madlener F., 2013. *A Model of Computation for Reconfigurable Systems.* Ph.D. thesis, Technische Universität, Darmstadt.

Object Management Group, 2016. *MDA - The Architecture of Choice for a Changing World.* URL `http://www.omg.org/mda/`. [Online; accessed 19-January-2017].

Uhrmacher A.M., 2001. *Dynamic Structures in Modeling and Simulation: A Reflective Approach. ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 11, no. 2, 206–232. ISSN 1049-3301. doi:10.1145/384169.384173.

Zeigler B.P.; Praehofer H.; and Kim T.G., 2000. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems.* Academic press, Orlando, FL, USA, 2nd ed. ISBN 0127784551.