Symbolic Flattening of DEVS Models

Bin Chen and Hans Vangheluwe
School of Computer Science
Mcgill University
Montréal, Canada
binchen, hv@cs.mcgill.ca

Keywords: DEVS, Direct Connection, Flattening

Abstract

Based on the Classic Discrete Event system Specification (DEVS) formalism, many efficient simulation languages have been proposed to meet the expressiveness and performance requirements of various applications. We propose the *Symbolic Flattening* of DEVS models to enhance simulation efficiency. The Modelica language is extended to provide a standard concrete syntax for DEVS. Based on the closure under coupling of DEVS, flattening is implemented in two steps:*Direct Connection* and *Flattening*. We use the domain of digital circuits to get quantitative insight into the improvement brought by our approach. We conclude that the approach is most effective for **High Interaction**, **Low Computation(HILC)** models. The technique is modular and may be combined with simulator-level (as opposed to our technique, which is symbolic, modeling level) techniques, or distributed simulation.

1. INTRODUCTION

The Discrete Event system Specification[1] formalism has been widely used in modeling and simulation for many years. The formalism is expressive and allows for the hierarchical and modular description of discrete event models.

Based on the Classic DEVS formalism, many extensions were proposed to meet different application requirements. This, as there does not exist a single formalism which is optimal, both from an expressiveness and from a performance point of view, for all application domains. Cell DEVS[2] adds spatial distribution features of Cellular Automata to DEVS and allows the modeling of physical systems by supporting spatially distributed, interacting cells. Dynamic Structure DEVS[3] is used when the model structure can change dynamically.

Parallel & Distributed DEVS[4] is well suited for **High Computation**, **Low Interaction(HCLI)** models, especially for large-scale systems.

The distribution of models speeds up the simulation by adding concurrent computing nodes. [4] presents a parallel simulation methodology which employs hierarchical scheduling for simulation of models within a processor and the **Time Warp** mechanism for global synchronization. [5] presents a DEVS modeling and simulation framework over a Peer to Peer network system DEVS / P2P. This extension of DEVS results in a customized new simulation protocol for distributed simulation which does not involve a coordinator.

[6] presents a sequential way to process **PDEVS** models. The abstract sequential simulator reduces the thread number, and the Flatten algorithm in simulator simplifies message transfer. The process way is integrated in a new simulator in [7]. Large Coupled model is split up into Virtual coupled models. These models can be simulated in both distributed and sequential manner with the support of the new simulator. [8] presents an improved simulation engine that combines Dynamic Structure DEVS with a real-time simulation engine. This leads to a powerful environment to support the development of embedded systems.

All the work mentioned above provides optimizations of the simulation level. [4] modifies the messaging mechanism to make **Time Warp** more efficient; [5] applies the P2P technique to emphasize the distribution; [2] focuses on the modeling of environments that are inherent compositional; [8] reduces the coordinator to improve the simulation performance by flattening the model hierarchy at run time. Algorithms are given to optimize the event scheduling, and message transfer in the coordinator. However, when the computation due to interactions between sub-models is much larger than the computation of individual models, the performance improvement is minimal. [6] modified the simulator for **PDEVS**, Select and data transfer in classical DEVS are not referred. [9] presents model composition to speedup simulation based on the closed under coupling property of DEVS.

We use these ideas as a starting point for our work to improve performance, at the modeling level, independent of the simulator. We propose to flatten DEVS models symbolically to optimize simulation performance for **High Interaction**, **Low Computation(HILC)** models. Our approach transforms coupled DEVS models textually represented in Modelica to a behaviorally equivalent atomic DEVS model. The interactions among models are integrated into the computation inside the atomic model. In contrast with simulation algorithms, all computation (transformation) is done on models before simulation time. This implies there is no run-time overhead.

The rest of this paper is organized as follows: Section 2 describes the Classic DEVS formalism in detail. Section 3 de-

scribes how to represent DEVS models in the neutral modeling language modelica. Section 4 describes the implementation of our approach. Section 5 presents a case study in which digital circuits are constructed to test the impact of our flattening on simulation performance. Section 6 concludes the paper.

2. DEVS FORMALISM

The DEVS formalism was introduced in the late seventies by Bernard Zeigler as a rigorous basis for the compositional modeling and simulation of discrete event systems [10]. It has been successfully applied to the design, performance analysis, and implementation of a plethora of complex systems.

A DEVS model is either an *AtomicBlock* or a *CoupledBlock*. An atomic model describes the behaviour of a timed, reactive system. A coupled model is the parallel composition of several DEVS sub-models which can be either atomic or coupled. Sub-models have *ports*, which are connected by channels. Ports are directional and are either *Inport* or *Outport*. Ports and channels allow a model to receive and send events from and to other models. A channel must go from an output port of some model to an input port of a different model, from an input port of a coupled model to an input port of one of its sub-models, or from an output port of a sub-model to an output port of its parent model.

An **atomic DEVS** model is a structure

$$\langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

where S is a set of sequential **states**.
X is a set of allowed **input events**.
Y is a set of allowed **output events**.
There are two types of transitions between states:
$\delta_{int} : S \to S$ is the **internal transition function** and
$\delta_{ext} : Q \times X \to S$ is the **external transition function**.
Associated with each state are $\tau : S \to \mathbb{R}_0^+$, the **time-advance** function and $\lambda : S \to Y$, the **output function**.
In this definition, $Q = \{(s,e) \,|\, s \in S, 0 \leq e \leq \tau(s)\}$ is called the **total state space**.
For each $(s,e) \in Q$, e is called the **elapsed time**, the time the system has spent in a sequential state s since the last transition. $\mathbb{R}^+$ denotes the positive reals (with zero included).

Informally, the operational semantics of an atomic model is as follows: the model starts in its initial state. It will remain in any given state for as long as the time-advance of that state specifies or until input is received on some port. If no input is received, after the time-advance of the state expires, the model first (before changing state) produces an output event as specified by the output function. Then, it instantaneously jumps to a new state specified by the internal transition function. However, if an input event is received before the time for the next internal transition, then it is the *external transition* which is applied. The external transition depends on the current state, the time elapsed since the last transition and the input event.

A **coupled DEVS** model named C is a structure

$$\langle X, Y, N, M, I, Z, select \rangle$$

where X and Y are as before.
N is a set of **component names** (or labels) such that $C \notin N$.
$M = \{M_n \mid n \in N, M_n$ is a DEVS model (atomic or coupled) with input set X_n and output set $Y_n\}$ is a set of DEVS **sub-models**.
$I = \{I_n \mid n \in N, I_n \subseteq N \cup \{C\}\}$ is a set of **influencer** sets for each component named n.
$Z = \{Z_{i,n} | \forall n \in N, i \in I_n, Z_{i,n} : Y_i \to X_n \vee Z_{C,n} : X \to X_n \vee Z_{i,C} : Y_i \to Y\}$ is a set of **transfer functions** from each component i to some component n.
$select : 2^N \to N$ is the **select** or tie-breaking function. 2^N denotes the powerset of N (the set of all sub-sets of N).

The connection topology of sub-models is expressed by the influencer set I_n of each component n. Note that for a given model n, this set includes not only the external models that provide inputs to n, but also its own internal sub-models that produce its output (if n is a coupled model). Transfer functions represent output-to-input translations between components. They can be thought of as channels that make the appropriate type translations. For example, a "departure" event output of one sub-model is translated to an "arrival" event on a connected sub-model's input. The *select* function takes care of conflicts as explained below.

The semantics for a coupled model is, informally, the parallel composition of all the sub-models. A priori, each sub-model in a coupled model is assumed to be an independent process, concurrent to the rest. There is no explicit method of synchronization between processes. Blocking does not occur except if it is explicitly modelled by the output function of a sender, and the external transition function of a receiver. There is however a *serialization* whenever there are multiple sub-models that have an internal transition scheduled to be performed at the same time. The modeler controls which of the conflicting sub-models undergoes its transition first by means of the *select* function.

We have developed our own DEVS simulator called `pythonDEVS` [11], grafted onto the object-oriented scripting language Python. We use `pythonDEVS` for the work described in this paper.

3. DEVS MODEL MODELICA SYNTAX

3.1. Why use Modelica?

Modelica is an object-oriented model description language[12]. It provides a structured, computer-supported way of doing mathematical and equation-based modeling.
The design goal of Modelica is to build a modeling language based on the Differential Algebraic Equation (DAE) formalism[13] with discrete-event features to handle discontinuities and sampled systems. It has been successfully used to specify models of systems in many physical domains. Though Modelica is originally designed for describing physical mod-

els, it has enough constructs to describe models in other formalism, including discrete-event ones.
This insight leads us to use Modelica as the description language for DEVS models [14]. The reasons for this choice are as follows. Firstly, Modelica has been developed and applied in modeling for several years, it is a relatively mature language. Secondly, Modelica comes with the Modelica Standard Library(MSL), a large model repository for different domains which has accumulated a large group of users. Thirdly, DEVS has been shown to be a formalism that is suitable for hybrid system modeling. Lastly, Modelica constructs such as (computationally) non-causal parameter-coupling equations may be used to increase the (re-)usability of DEVS models.

3.2. Elements of Representation

3.2.1. Basic Elements in Modelica

The basic structural element in Modelica is a class, as in other object-oriented languages. A class in **Modelica** uses equations to describe model behavior. Class composition and inheritance are used to mimic the hierarchy and composition of real-world entities.
There are at least seven restricted classes used to represent DEVS models. They are introduced below:
model: The only restriction of a model class is that it may not be used in connections. Its semantics are identical to the general class construct in Modelica, and it is most commonly used.
record: The record class is used to describe structured data. No equations are allowed in the definition or in any of its components.
type: A type is a class that is an alias or extension of an existing class. A type restricted class may only be an extension to a predefined type, `enumeration`, `record`, or `array` of type. The purpose of using type is to identify a data structure by a meaningful name.
connector: The restrictions of connector classes are identical to those of store classes[12], except that connector classes are designed to be used in connections.
block: The block restricted class is used to model causal (input/output) block diagrams. In Modelica, the two keywords, `input` and `output`, are used as component prefixes to postulate the data flow direction.
Function: The body of a Modelica function is an algorithm that specifies the execution behavior when the function is called. The parameters for a function are specified by the keyword `input`, and results are saved to variables marked by the keyword `output`.

3.2.2. Modelica representation of DEVS models

The atomic model consists of parameters, sequential states, model state, input and output ports, and behavior functions. In the parameter declarations part of a model, parameters needed for instantiating a model are specified (with their default values).
Sequential states are represented using the Modelica `enumeration` type. The model state is declared as a normal Modelica class instance variable. Input and output ports are declared using `input` and `output` keywords and are instances of a predefined class `DEVSPort`. All the DEVS functions are defined as Modelica functions. The `timeAdvance` function requires an output parameter `timespan`, through which the value of the time interval for a sequential state is returned.
Similar to atomic models, A coupled DEVS model has parameters, input and output ports, sub-models, and port connections. Ports and parameters are declared in the same way as for atomic models. Sub-models are declared as normal Modelica class instances.
Port connections are defined by Modelica's `connect` operator. In plain Modelica, `connect` is used to connect two Modelica connectors, and here we use it to connect two DEVSports.
Additionally, DEVS' **Z** (transfer) and **Select** functions are defined using Modelica functions in the coupled model. There may be several **Z** functions in a coupled model. It is worthwhile to note that the **Z** function is combined with the ports in the **connect** function, as will be shown in the example later. The `Select` tie-breaking function to resolve conflicts between simultaneous internal transitions in different sub-models is defined using the input and output parameters of Modelica and returns the selected sub-model.

3.3. Example

We use a DEVS model of logical gates to demonstrate the atomic and coupled DEVS model representation in Modelica. The atomic and coupled DEVS model are listed below.

```
class NotGate
 extends AtomicDEVS
 parameter String Name='Not';
 input DevsPort NotGateInput;
 output DevsPort NotGateOutput;
 NotGateState state();
 function intTransition
   %current state not changed;
 end intTransition;
 function extTransition
  state.input := peek(NotGateInput);
 end extTransition;
 function outputFunc
  if state.input == 1 then
   poke(0, NotGateOutput);
  else
   poke(1, NotGateOutput);
  end if;
 end outputFunc;
 function timeAdvance
  output Real timespan
```

```
  timespan:=0.0000005;
 end timeAdvance;
end NotGate;

class Circuit
 extends CoupledDEVS
 parameter String Name='Circuit';
 input DevsPort Input;
 output DevsPort Output;
 NotGate Not1();
 NotGate Not2();
 equation
  connect(Input, Not1.NotGateInput,Z1);
  connect(No1.NotGateOutput, Not2.NotGateInput);
  connect(Not2.NotGateOutput, Output);
 end
 function Z1
  input Integer in
  output Integer out
  if in > 1 then
   out := 1
  end if;
 end Z1
 function Select
  Input BaseDevsList immList
  output BaseDEVS imm
  imm := immList[0];
 end Select
end Circuit;
```

4. FLATTENING DEVS MODELS

As mentioned in [1], it is possible to construct a *resultant* atomic DEVS model for each coupled DEVS. This *closure under coupling*[15] of atomic DEVS models makes *any* coupled DEVS behaviorally equivalent to the resultant atomic DEVS. Supported by this property of closure, the *Flattening* of DEVS is implemented in two steps: *Direct Connection* and *Flattening*.

4.1. Closure of DEVS under coupling

As mentioned before, any *hierarchically* coupled DEVS can thus be flattened to an atomic DEVS by induction. As a result, the requirement that each of the components of a coupled DEVS be an atomic DEVS can be relaxed to be atomic *or* coupled as the latter can always be replaced by an equivalent atomic DEVS.

The core of the closure procedure is the selection of the most *imminent* (i.e. soonest to occur) event from all the components' scheduled events [1]. In case of simultaneous events, the *select* function is used. In the sequel, the resultant construction is described.

From the coupled DEVS

$$\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select\rangle,$$

with all components M_i atomic DEVS models

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i\rangle, \forall i \in D$$

the atomic DEVS $\langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda\rangle$ is constructed.

The resultant set of sequential states is the product of the total state sets of all the components $S = \times_{i\in D} Q_i$, where

$$Q_i = \{(s_i, e_i) | s \in S_i, 0 \le e_i \le ta_i(s_i)\}, \forall i \in D.$$

The time advance function $ta\,ta : S \to \mathbb{R}^+_{0,+\infty}$ is constructed by selecting the *most imminent* event time, of all the components. This means, finding the smallest time *remaining* until internal transition, of all the components

$$ta(s) = min\{\sigma_i = ta_i(s_i) - e_i | i \in D\}.$$

A number of *imminent* components may be scheduled for a *simultaneous* internal transition. These components are collected in a set $IMM(s) = \{i \in D | \sigma_i = ta(s)\}$. From IMM, a set of elements of D, *one* component i^* is chosen by means of the *select tie-breaking* function of the coupled model

$$\begin{array}{rcccc} select & : & 2^D & \to & D \\ & & IMM(s) & \to & i^* \end{array}$$

Output of the selected component is generated *before* it makes its internal transition. Note also how, as in a Moore machine, input does not directly influence output. In DEVS models, *only* an internal transition produces output. An input can only influence/generate output via an internal transition similar to the presence of *memory* in the form of integrating elements in continuous models. Allowing an external transition to produce output could lead to infinite instantaneous loops. This is equivalent to algebraic loops in continuous systems. The output of the component is translated into coupled model output by means of the coupling information

$$\lambda(s) = Z_{i^*,self}(\lambda_{i^*}(s_{i^*})), \text{if } self \in I_{i^*}.$$

If the output of i^* is not connected to the output of the coupled model, the non-event ϕ can be generated as output of the coupled model. As ϕ literally stands for no event, the output can also be ignored without changing the meaning (but increasing performance of simulator implementations).

The internal transition function transforms the different parts of the total state as follows:

$$\delta_{int}(s) = (\ldots, (s'_j, e'_j), \ldots), \text{ where}$$

$$\begin{array}{rcll} (s'_j, e'_j) & = & (\delta_{int,j}(s_j), 0) & ,\text{for } j = i^*, \\ & = & (\delta_{ext,j}(s_j, e_j + ta(s), Z_{i^*,j}(\lambda_{i^*}(s_{i^*}))), 0) & ,\text{for } j \in I_{i^*}, \\ & = & (s_j, e_j + ta(s)) & ,\text{otherwise.} \end{array}$$

The selected imminent component i^* makes an internal transition to sequential state $\delta_{int,i^*}(s_{i^*})$. Its elapsed time is reset to 0. All the influencees of i^* change their state due to an external transition prompted by an input which is the output-to-input translated output of i^*, with an elapsed time adjusted

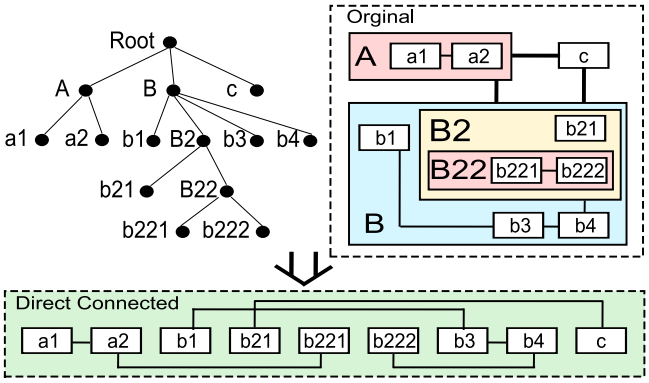

Figure 1. Transformation from Original Model to Direct Connected Model

```
direct_connect
input:Root model rm, coupled model cm in rm
output:rm with the cm direct connected

1  for each submodel m in cm:
2    if m is atomic:
3      add_model(rm,m)          //add the atomic of cm to the rm
4    if m is coupled:
5      direct_connect(cm,m)     //recursive invoke to direct connect all atomics
6  for each subconnection c in cm:
7    if c is between atomics in cm:
8      copy_connection(rm,c)    //copy the connection c to rm
9    if c is between atomics and rm:
10     adjust_ports(rm,c)       //adjust the ports in c and copy it to rm
```

Figure 2. *Direct Connection* Algorithm

for the time advance $ta(s)$. The influencees' elapsed time is reset to 0. Note how i^* is not allowed to be an influencee of i^* in DEVS. The state of all other components is not affected and their elapsed time is merely adjusted for the time advance $ta(s)$.

The external transition function transforms the different parts of the total state as follows:

$$\delta_{ext}(s,e,x) = (\ldots,(s_i',e_i'),\ldots), \text{ where}$$

$$\begin{aligned}(s_i',e_i') &= (\delta_{ext,i}(s_i,e_i+e,Z_{self,i}(x)),0) &&, \text{for } i \in I_{self},\\ &= (s_i,e_i+e) &&, \text{otherwise.}\end{aligned}$$

An incoming external event is routed, with an adjustment for elapsed time, to each of the components connected to the coupled model input (after the appropriate input-to-input translation). For all those components, the elapsed time is reset to 0. All other components are not affected and only the elapsed time is adjusted.

4.2. Direct Connection

Muzy and Nutaro [16] have presented a simulation algorithm for Dynamic Structure DEVS. The algorithm transfers messages directly from one atomic model to another without passing via a coordinator. In our paper, *Direct Connection* does not work at the simulator level, but rather we *symbolically* transform a hierarchical coupled model into a coupled model with depth one.

As shown in Figure 1, the original model's hierarchy tree describes the connection and containment relationships between the atomic and coupled sub-models. It is transformed to a two-level hierarchy model. *Direct Connection* is used to simplify the complexity and lower the redundancy in DEVS models. Without the overhead of passing messages through ports and coordinators, message transfering becomes direct and efficient.

4.2.1. Algorithm

The *Direct Connection* algorithm is shown in Figure 2. The algorithm starts from the root node and traverses all the nodes in the hierarchy tree using the *Direct Connection* function. Each node is checked whether it is a coupled model. The atomic models and connections between them are copied to the root node first.

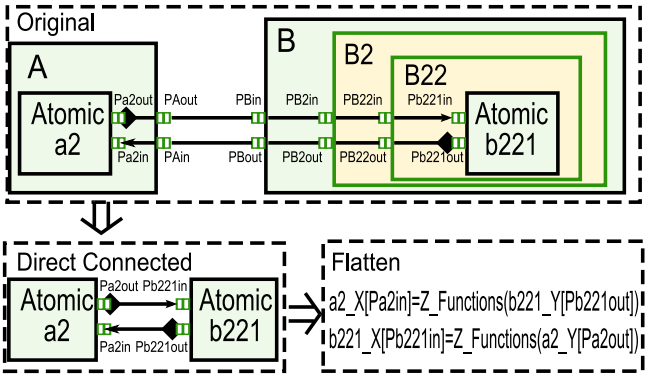

Figure 3. Ports Elimination in the Transformation

The connections that contain the ports from coupled models are replaced in the following three phases: Find the port which is connected to the coupled model's port; Construct a new connection to connect the target port with the atomic port; Delete the former connection and coupled ports, then move the new one to the root node. If the sub-model is still a coupled one, invoke the *Direct Connection* function again. The recursion is guaranteed to terminate thanks to the finite depth of the model tree.

Figures 3 and 4 describe the port elimination and connection evolution during the connecting process. Figure 3 also shows that messages transferred between ports are transformed to data passing in the ports dictionary in the *Flattened Atomic Model*. This will be discussed in detail later.

4.2.2. Z and Select Functions in Direct Connection

Though the *Direct Connection* function eliminates coupled models, the coupled model's **Z** Functions and the **Select** Function have to be taken into account to guarantee that the transformation preserves behavior. For **Z**, the first step is to rename the functions before *Direct Connection*. This is because the ports information of **Z** Functions will

be lost when the connection is eliminated. So we have to rename the functions to record the ports and model information. The new name format is: `Z_[Output Model and Port Name]_[Input Model Port Name]`. When the ports of atomic models are direct connected, there will likely be more than one **Z** Function. The problem is how to handle multiple **Z** Functions in one connection. The solution is shown in Figure 4. As mentioned before, the names of **Z** Functions have been modified to add the extra information. For the coupled ports that have been eliminated in *Direct Connection*, the names of **Z** Functions have to be changed accordingly. The new name is composed of the names from new output and input ports. When a name conflict occurs, a postfix is appended to make the names unique. The content of the postfix is the order of the function, ranked by the position from output port to input port. As shown in the figure, the function closest to the output port gets the highest order, and vice versa. This is somewhat arbitrary but guarantees uniqueness. As for the **Select**, the function is directly copied to the root model because there is only one **Select** Function in every coupled model. The name of the **Select** Function has to be changed in order to avoid name conflicts in the root model. The name of the coupled model is used to prefix the **Select** Function's name as in 'B_B2_Select()'.

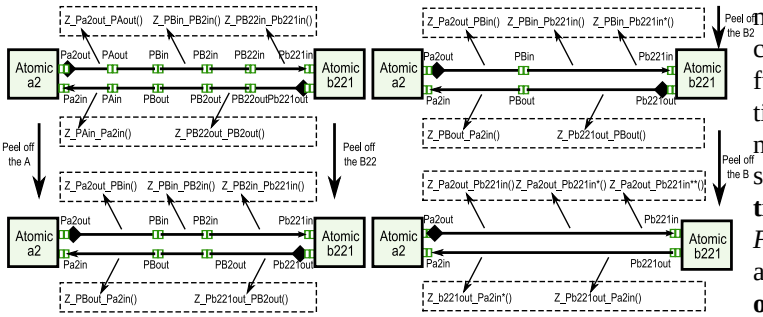

Figure 4. The **Z** Functions in Ports Elimination

4.3. Flattening

After the *Direct Connection*, *Flattening* is much easier since the hierarchy of original model has been simplified. The goal of *Flattening* is to turn a coupled model into an atomic one. It means that all the connections between atomic models are transformed to computation inside the *Flattened Atomic Model*. This implies that the simulation mechanisms traditionally encoded in a simulation engine's coordinator such as message transfer, event scheduling, and time advance have to be explicitly inserted in the *Flattened Atomic Model* to maintain the consistency in the model transformation. Additionally, the **Z** and **Select** Functions have to be re-implemented too.

4.3.1. Algorithm

The *Flattening* algorithm consists of three phases: Modify atomic DEVS models connected in the coupled model; Construct a new *Flattened Atomic Model* to reconstruct the message transfer, scheduling and time advance mechanism; Integrate the **Z** and **Select** Functions into the *Flattened Atomic Model*.

4.3.2. Modification of Atomic Models

The atomic models have to be modified to meet the requirements of *Flattening*. As all ports are eliminated, the messages can not be sent and received through ports. Dictionaries `X` and `Y` are added to store the messages. The keys used to index the dictionaries are the names of ports while the values are the messages. Likewise, the `extTransition` and `intTransition` and `output` functions are adapted to the dictionaries too. In the specific case of PythonDEVS, this entails updating the `peek` and `poke` functions which encode message reception in `extTransition` and message output in `output` respectively. The `peek` functions are replaced by `X[portName]` and the `poke` functions are replaced by `Y[portName]`. The atomic models have the responsibility to inform their context whether there are messages to be sent. The symbol `IsSendMsg` is used to encode this in our flattened model.

4.3.3. Flattened Atomic Model Construction

The *Flattened Atomic Model* is a special atomic model which encodes the coupled simulation mechanism. It optimizes inter-model message transfer and improves the efficiency of event list scheduling. As this model encodes the functionality of a coordinator, it requires some helper dictionaries and functions. The `X`, `Y` and `isSendMsg` are still needed at the highest level to communicate with the outside. **States**, **UpdateModels**, **OutputModels** and **Connections** are newly created dictionaries to be inserted into the *Flattened Atomic Model*. **CollectModels()**, **MapConnection** and **MapY2X** are the new functions while **extTransition**, **outputFunc()**, **intTransition()** and **timeAdvance()** are the modified functions. **States** and **Models**: **States** is the set of states in the *Flattened Atomic Model*. The set is encoded as a dictionary indexed by the unique identifiers of sub-models. These are the names of sub-models with '_State' appended. **Models** is the dictionary which stores all the atomic sub-models inside the *Flattened Atomic Model*. The keys of **Models** are **States**.

UpdateModels: collects all the sub-models' received messages. The keys are again the keywords of sub-models belonging to **State**. By means of **UpdateModels**, it is possibly to directly access only those models which have received a message.

OutputModels: collects all the sub-models sending messages after their internal transition. `isSendMsg` is used to check whether a sub-model sent a messages. If true, the sub-model is added to the dictionary.

Connections: replaces the connect function of the direct connected coupled model. It stores the connection information

between the sub-models. As every connection is composed of the object producing output and that receiving input, keys of **Connections** are the outputting features which are generated from the output sub-model and port name. Since one outputting object may be connected to several receivers, the values stored in the dictionary are collections of receiver features generated from the input sub-model and port name. Using **Connections**, all the receivers can be determined when some sub-models output messages.

CollectModels(): extracts the sub-models of a direct connected coupled model to construct the **Models** with **States**. The function is invoked during the initialization of *Flattened Atomic Model*.

MapConnection(): transforms the connect function in a direct connected coupled model into **Connections**. It is invoked during the initialization of *Flattened Atomic Model*. **extTransition()**: is the same as the atomic model's function. It should be emphasized however that the use of `Y` and `X` is reversed as messages are passed from `Y` to `X`. For this reason, the messages from outside the model are `poke`ed to the `Y` of the *Flattened Atomic Model*. Messages are subsequently passed to sub-models by **MapY2X**.

outputFunc(): The original `outputFunc` is modified to `poke` messages from dictionary `X` to the outside. `X` has to be cleared after poking.

intTransition(): is exactly the same as the original one. The sub-model is indexed by the current state and executes the internal transition and output of the current sub-model.

timeAdvance(): is modified to do the event list scheduling and time advancing in the *Flattened Atomic Model*.

When the *Flattened Atomic Model* is initialized, the `TimeList` is constructed by traversing all the initi ate times of sub-models. The keys of `TimeList` are the keywords of sub-models while the relevant value is t_L. During the construction of `TimeList`, the ordered time dictionary is built by *SetModelAtTime()*. The sub-models are arranged by the time stamp in the **ordered time dictionary**. Sub-models with identical t_L are collected in an array at every time stamp.

The **UpdateModels** is traversed to execute *extTransition()* of sub-models receiving messages. t_N of every sub-model is calculated and the **ordered time dictionary** is updated immediately so that the ordered sub-model array is maintained synchronously.

Thanks to the **ordered time dictionary**, event list scheduling can be done easily as the first array of the dictionary is `immList`. Then, the immediate sub-model is selected by *Flatten_Select* and the current state of *Flattened Atomic Model* is fixed. t_N of the current sub-model is calculated by *timeAdvance()* plus t_L and it is inserted into the **ordered time dictionary**. The time span is calculated by the current time and t_N of current sub-model. If t_N is greater than the current time, the time span is the time stamp minus the current time and the current time is set to t_N. Otherwise the time span is zero. The time span here is used to control the time advancing of *Flattened Atomic Model*. As each sub-model shares the same **timeAdvance()**, the time can only be advanced when all the sub-models have already advanced.

MapY2X(): replaces the message transfer mechanism in the coordinator of coupled models. It traverses **UpdateModels** and sends messages to connected sub-models. During the traversal, the outputting objects are retrieved through the sub-model and port information from **UpdateModels**. Then, **Connections** provides the receiving objects corresponding to the outputting objects. Based on the connected objects, the messages are passed from the `Y` of output sub-model to the `X` of the receiving sub-model. The **Z** Functions are executed.

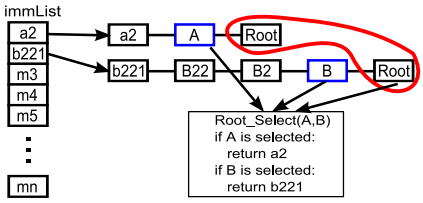

Figure 5. The Principle of *Flatten_Select Function*

4.3.4. Z Functions and Flatten_Select Function

As mentioned before, all the **Z** Functions have been transformed to the direct connected model. In order to reuse them in the flattened model, they are copied to *Flattened Atomic Model*. In the **MapY2X**, they are invoked in the **Z** Functions. The invoking order is decided by the postfix of the function's name.

Figure 5 describes the principle of *Flatten_Select Function*. If the sub-model meets a time advance conflict, the common select function is need. But not all the sub-models belong to the same coupled model. So, it is necessary to find the common parent coupled model of the sub-models. This means that the coupled model must contain all the sub-models. From the bottom of the hierarchy, the select function of the first common coupled model is the one used in the original model. The algorithm is depicted below:

First, the top two sub-models are popped to do the selection. According to the coupled information saved in the name of sub-models, the common coupled parent model is found. As shown in the figure, the common coupled parent of sub-model `a2` and `b221` is `Root`. So the `Root_Select` is chosen to do the selection. Thus, the input is `A` and `B`.

Second, if `A` is selected, the sub-model `a2` is selected, otherwise `b221` is selected. The next sub-model in `ImmList` is popped to do the selection with the former selected sub-model again. Finally, the immediate sub-model can be output while traversing the `ImmList`.

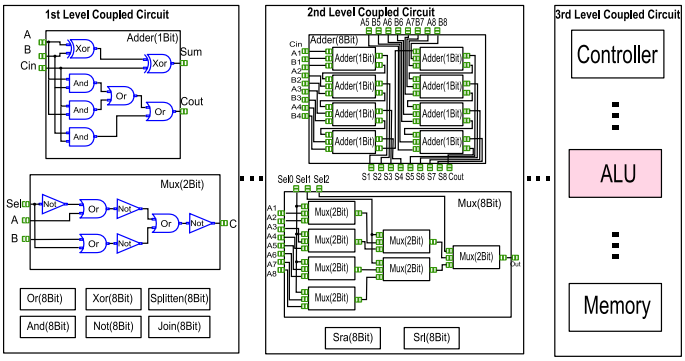

Figure 6. Case Study: Circuit from gate to ALU

5. CASE STUDY

5.1. Compiling a DEVS Model

Obviously, a DEVS model represented in Modelica can not be simulated directly as it is just a textual representation. Thus, a compiler is needed to transform the DEVS model into a form suitable for simulation. We use our μ**Modelica** compiler [14] which by default generates input for our **PythonDEVS** simulator. The compiler was extended with a flattening phase which is invoked before generating simulatable output.

5.2. Benchmark Model

As mentioned in the introduction, DEVS **Flattening** may improve simulation performance of **High Interaction and Low Computation(HILC)** models. We choose a **Logical Circuit** to be our benchmark model. It is well known that gates are the basic units of **Logical Circuit**s. Even **Very-large-scale integration (VLSI)** systems are composed of simple gates such as **And**, **Or**, **Not** and **Xor**. The time delay between input and output of basic gates is in the order of a few nanosenconds which is an indication that the internal computation is low. Hence, logical circuit models are a typical example of **HILC**.

We build two kinds of circuit: the real circuit and the test circuit. Real **VLSI** circuits are shown in Figure 6. Starting from basic gates, First Level Coupled Circuits can be constructed: the Adder(1Bit), Mux(2Bit), Or(8Bit) and so forth. The First Level Coupled Circuits can be combined to form Second Level Coupled Circuits: Adder(8Bit), Mux(8Bit),Sra(8Bit) and so forth. Lastly, Second Level Coupled Circuits can be combined to form an Arithmetic and Logic Unit (**ALU**). Likewise, **Control** and **Memory** components of a CPU's datapath (realizing a given Instruction Set Architecture) can be modeled.

In order to test the performance improvement obtained by *Flattening*, we build a test circuit which is composed of Not gates. The number of gates and connections in the test circuit can be changed for the purpose of our experiments. The circuit is created according to the rules below: the coupled models are not allowed to contain more than 10 sub-models. When the number of the sub-models in the coupled model is going to exceed 10, another coupled model is created to contain the new sub-model. By increasing the number of gates, the hierarchy of the test model becomes a structure like a pyramid. This is what is commonly found in real circuit models.

The simulations are tested on an Intel Core 2 Extreme X6800 processor at 2.66GHz with 8Gb of memory. All the results are obtained from multiple repeated experiments. First, the outliers are removed and subsequently, the average is computed.

5.3. Analysis

In Figure 7, the left part shows the improvement of direct connected models compared to original models, while the right part shows the improvement of flattened models with respect to the original models. The direct connected model does not improve performance much as shown in the figure. The curve only goes up a bit and then down quickly. On the contrary, the curve of flatten/original goes up all the time. This means that the more complex a model is, the higher the improvement achieved.

Let us first investigate the performance of the direct connected model as compared to that of the original model. Port elimination improves the performance slightly when the model is not complex. This, because the messages can be transferred directly between atomic models, doing away with port overhead. As the model becomes more and more complex, all the atomic models are managed by a single coordinator. The reasons for bad performance are summarized below:

- When the model becomes complex, the sorting process gradually occupied most of the time. The size of the event list grows with the complexity of the model (assuming that all components are active most of the time). The larger the the event list, the lower the performance of the sorting process.

- According to the coordinator protocol, sub-models are traversed to check whether there are some messages to be transferred. The traversing is time consuming if the number of sub-models is high. However, it is very well possible that most of the sub-models traversed did not receive any messages.

The two causes cut off the improvement brought by port elimination. And the performance improvement decrease to 0 when the model complexity reaches that of an **ALU**.

The results in the figure show that the flattened model leads to significantly improved simulation performance compared with the original model. The reasons are summarized below:

- Messages are transferred inside the *Flattened Atomic Model*. Time used to send and receive messages between the ports and atomic models is saved.

Table 1. Simulation Performance for the Real Circuit: Original, Direct Connected, Flattened

Circuit	GateAmount	Original(s)	Connected(s)	Flattened(s)
NotGate	1	0.000678	0.000679	0.000633
Adder(2Bit)	17	0.021018	0.020978	0.009163
Adder(2Bit) + Mux(2Bit) * 2	25	0.070707	0.092659	0.032808
Adder(4Bit)	33	0.048223	0.087518	0.017283
Adder(4Bit)+Mux(2Bit)*3	54	0.364086	0.837083	0.136674
Adder(8Bit)	54	0.465491	1.401712	0.161146
Adder(8Bit)+Mux(8Bit)	183	3.215183	20.686988	0.913302
Adder(8Bit)*2+Mux(8Bit)*3	439	12.846081	109.308622	3.435668
ALU	641	29.153758	371.523992	6.380175

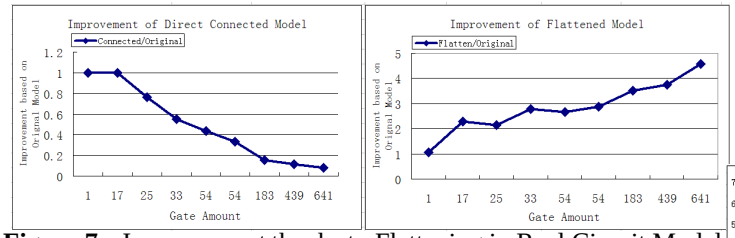

Figure 7. Improvement thanks to Flattening in Real Circuit Models

- The event list data structure is optimized to help improve the efficiency of the sorting process. All the sub-models are ordered by time stamps. The complexity of the event list is related to the number of events, not to the increasing number of sub-models. In the case of a digital circuit, the time stamps of gates are always the same. Thus, the size of the event list is so small that the speed of the sorting process is improved drastically.

- The dictionary **UpdateModels** is used to collect the sub-models that receive the messages. Therefore, only the sub-model receiving messages is traversed to do `extTransition()`.

- The dictionary **OutputModels** is used to collect the sub-models that send the messages. In **MapY2X**, only the sub-model sent messages is traversed to pass the message to relevant sub-models.

These four causes help improve the simulation performance of a flattened model. Theoretically, the improvement will be proportional with the increasing complexity of the model. This is especially so when the limitations of the simulator hardware are reached. This is because the complexity of the original model is higher than that of the flattened one. Thus, simulation of the original model will reach the limitation (usually memory) first.

The runtime becomes extreme long at that point, whereas the runtime for the flattened model still increases linearly. So, the improvement is close to infinity at this point. We use the speed of gates (v_g) as another performance metric in the case of the test circuit. v_g is the reciprocal of time used per gate. It descends when the improvement ascends by the increasing of gates. The combination of v_g and the improvement of performance in *Flatten DEVS* can help find the optimized point where the improvement and efficiency are all considered. The performance of the test circuit are shown in Figure 8. The v_g curves of original and flattened model are shown in the top-left. The top-right is the improvement curve which is still rising, though there are negligible oscillations. The bottom curve shows that the optimal point is 5600 gates with improvement factor `5.4` over the original model. The circuit scale at the optimal point is relatively smaller than the circuit simulation mentioned in [17]. There are two main reasons: First, the experiments here are implemented in `Python` code. Though it is an easy to use object oriented language, the efficiency is lower, especially compared with **C++**. Second, we used the Classic DEVS formalism without any optimizations in the simulation framework. We emphasized the performance increase by *Symbolic Flattening* in the modeling domain instead of working on the simulation itself.

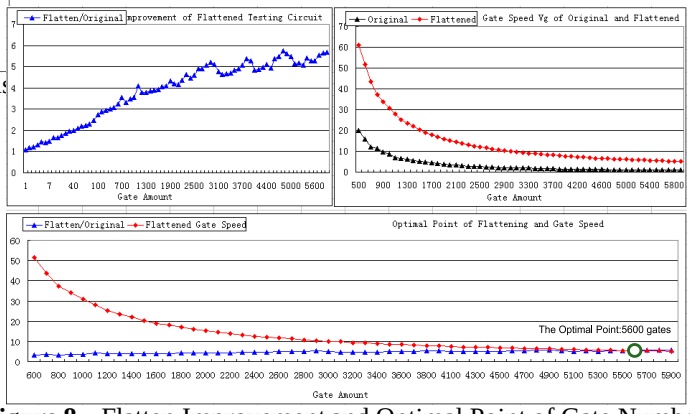

Figure 8. Flatten Improvement and Optimal Point of Gate Number

6. CONCLUSION

In this paper we first introduced the DEVS formalism and subsequently presented its textual representation in the standard modelling language Modelica. The closure of DEVS under coupling was discussed. We present our contribution: optimization at the modeling level (as opposed to simulation-level optimizations), first by *Direct Connection* and then by

Flattening. Compared to [9], transformation algorithms are introduced and the automatic *Symbolic Flattening* of DEVS model is described in detail. The transformations are implemented inside our own compiler μ**Modelica**.

Digital circuits are modeled to illustrate the performance of our flattening procedure. We notice that flattening greatly improves simulation performance. Actually, *Symbolic Flattening* is perfectly complementary to the formalisms and runtime optimization mentioned in section 1.. From our performance studies, we conclude that *Symbolic Flattening* is most effective in the **High Interaction**, **Low Computation** domain. Moreover, flattening scattered sub-models can improve the utilization of memory.

It is worthwhile to point out that the optimization brought by flattening is compositional. This means that it can be applied to models generated from other formalisms (such as Statecharts or Differential Equations). The approach is independent of the simulation framework used.

Thanks to the compositionality, we envision flattening other formalisms in future. Examples are Parallel & Distributed DEVS and Dynamic Structure DEVS. In the case of Parallel & Distributed DEVS, the models can be completely analyzed before the simulation. The high interaction models are flattened before partitioning to minimize the size of the simulation. Thus, the final simulation will benefit from both modeling and simulation-time optimization in two aspects.

The remaining bottleneck in our approach is still the event list management in a *Flattened Atomic Model*. This is however a known problem for which solutions have been devised in the simulation literature. We plan to apply and evaluate our techniques on very large scale problems.

REFERENCES

[1] P. B. Zeigler, P. Herbert, and K. T. Gon., *Theory of Modeling and Simulation, Second Edition*. Academic Press, 2000.

[2] G. A. Wainer, "Applying cell-devs methodology for modeling the environment," *SIMULATION*, vol. 82, no. 10, pp. 635–660, October 2006.

[3] F. J. Barros, "Modeling formalisms for dynamic structure systems," *ACM Trans. Model. Comput. Simul.*, vol. 7, no. 4, pp. 501–515, 1997.

[4] T. G. K. Ki Hyung Kim, Yeong Rak Seong and K. H. Park, "Distributed simulation of hierarchy devs models: Hierarchical scheduling locally and time warp globally," *Trans. Soc. Comput. Simul. Int.*, vol. 13, no. 4, pp. 135–154, 1997.

[5] S. Cheon, C. Seo, and B. P. Z. Sunwoo Park, "Design and implementation of distributed devs simulation in a peer to peer network system," in *2004 Military, Government, and Aerospace Simulation*, 2004.

[6] J. Himmelspach and A. M. Uhrmacher, "Sequetial processing of pdevs models," in *3rd EUROPEAN MODELING AND SIMULATION. SYMPOSIUM*, Barcelona, Spain, 2006, pp. 239–244.

[7] J. Himmelspach, R. Ewald, S. Leye, and A. M. Uhrmacher, "Parallel and distributed simulation of parallel devs models," in *SpringSim '07: Proceedings of the 2007 spring simulation multiconference*. San Diego, CA, USA: Society for Computer Simulation International, 2007, pp. 249–256.

[8] H. Shang and G. A. Wainer, "Dynamic structure devs: Improving the real-time embedded systems simulation and design." in *Annual Simulation Symposium*. IEEE Computer Society, 2008, pp. 271–278.

[9] W. B. Lee and T. G. Kim, "Simulation speedup for devs models by composition-based compilation," in *Summer Computer Simulation Conference, SCSC2003*, 2003, pp. 395–400.

[10] P. B. Zeigler, *Multifacetted modelling and discrete event simulation*. San Diego, CA, USA: Academic Press Professional, Inc., 1984.

[11] J.-S. Bolduc and H. Vangheluwe, "The modelling and simulation package pythondevs for classical hierarchical devs," Tech. Rep., 2001.

[12] P. Fritzson and P. Bunus, "Modelica, a general object-oriented language for continuous and discrete-event system modeling and simulation," in *In Proceedings of the 35th Annual Simulation Symposium*, 2002, pp. 14–18.

[13] H. Elmqvist and S. E. Mattsson, "An introduction to the physical modeling language modelica," in *Proc. 9th European Simulation Sympossium ESS97, SCS Int*, 1997, pp. 110–114.

[14] H. Song, "Infrastructure for devs modelling and experimentation," *Master Thesis*, 2006.

[15] H. Vangheluwe, "Lecture notes for cousre of modeling and simulation," Montreal, Canada, Tech. Rep., 2002.

[16] A. Muzy and J. Nutaro, "Algorithms for efficient implementations of the devs & dsdevs abstract simulators," in *1st Open International Conference on Modeling & Simulation, OICMS2005*, 2005, pp. 273–279.

[17] Q. Xu and C. Tropper, "Towards large scale optimistic vlsi simulation," *Simulation Modelling Practice and Theory*, vol. 14, no. 6, pp. 695–711, 2006.