Logisim to DEVS translation

Yentl Van Tendeloo[†] and Hans Vangheluwe[†,‡]
[†]University of Antwerp, Belgium
[‡]McGill University, Canada
Email: yentl.vantendeloo@student.ua.ac.be, hv@cs.mcgill.ca

***Abstract*—We propose a transformation from digital logic circuits modelled in the Logisim modelling language (and tool) to behaviourally equivalent models in the Discrete-EVent System specification (DEVS) formalism. This is achieved by mapping each Logisim component to a corresponding atomic DEVS model and by preserving the component coupling. The challenge in this work is the faithful preservation of all details of the Logisim semantics. The transformation is described and the translation of an example ALU is given.**

I. Introduction

Logisim[1] is a visual tool for designing and simulating digital logic circuits. It is primarily used for educational purposes. The Discrete-EVent System specification (DEVS)[2] formalism is highly expressive, modular and has a precise formal definition of both its syntax and its semantics. For these reasons, it is used as a semantic domain for many types of (domain-specific) discrete-event formalisms. As such, it is an excellent target for transformation, as it allows multi-formalism modelling through the mapping onto this single common formalism as demonstrated in [3] and [4].

Figure 1 gives an overview of all the implemented transformations. The boxes denote modelling languages and the arrows denote transformations. In particular, the vertical arrows denote simulation. The top layer in the figure depicts Verilog and Logisim, two domain-specific languages for the domain of logic circuits. Via an intermediate description (modelling language), Logisim models are translated to either Verilog models or to DEVS models. The bottom layer of the figure shows the behaviour traces generated by the three simulators (VCD trace from Verilog, XML trace from DEVS, Logisim trace from Logisim).

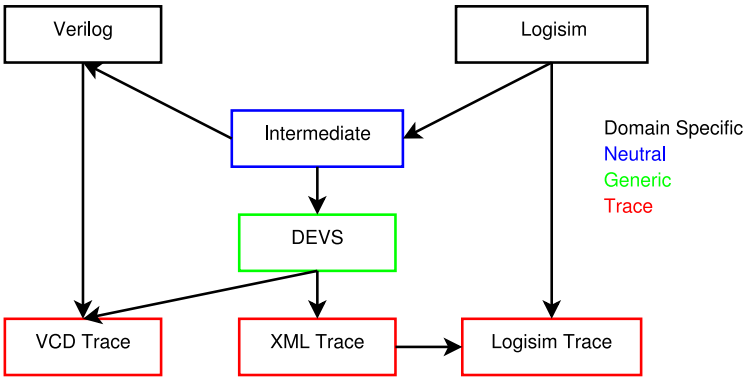


Figure 1. Overview of supported transformations

The remainder of this paper is organized as follows. Section II gives the rationale for the transformation to DEVS. Section III describes the transformation from Logisim to DEVS. Our testing strategy to verify that the generated DEVS models preserve the Logisim semantics is explained in section IV. Section V analyses simulation performance of the generated DEVS. Related work is explored in Section VI. Section VII concludes the paper.

II. Rationale

Performing the translation from Logisim to DEVS has several distinct advantages, which allows us to harness the power of both languages and all associated tools and techniques.

1) **Logisim as a domain-specific modelling environment**
Logisim with all its features (library of logic elements, simulation, basic logging, intuitive interface, static checking of correctness of models with visual feedback) implements a syntax-directed editing environment for a Domain-Specific Language (DSL). A Logisim to DEVS transformation is helpful as an intermediate structure is generated from each Logisim circuit, which can be easily reused for target formalisms other than DEVS (a Verilog Hardware Description Language [5] target is for example also supported by our translator, making Logisim a domain-specific visual front-end for Verilog).
2) **Couple other DEVS models**
As soon as a DEVS model of the circuit is generated, nothing prevents us from using this model as a component in other DEVS models. It is thus possible to create for example a probabilisitic event generator, a model of a circuit's environment. This is not possible in Logisim alone. The generated model can also be used in an educational setting, to verify a provided logisim circuit file with a specified interface. Furthermore, the output of the simulation can be passed to other DEVS models which could use this data for performance metrics gathering.
3) **Insight in semantics**
Logisim provides many standard components, from basic logical gates to a fully capable RAM component. Sadly enough, most of these components lack a *complete precise* description of their semantics in the *Logisim Library Reference*. While the semantics in the basic scenarios are mentioned, often there is no mention of what happens on a *floating* or *error* value. The only way to find out the semantics is through

experimentation. For example, the behaviour of a *Register* component on erroneous or floating input: will the error value be saved, will the memory be cleared, will the memory be kept the same as before? What will the output at that time be: the value of the register itself, or the input value?

4) **Static checks of Logisim circuits**
The save feature of Logisim does not prevent a user from saving an invalid circuit, thus it is possible that a circuit is created with non-matching bitsizes on both ends of a wire (or a collission of such wires). Performing a syntactic sanity check on Logisim models is therefore implemented. Other, less obvious checks, implemented in our translator, but not implemented in Logisim, are useful. For example, detecting a memory element without an input on its clock input port. These situations probably indicate a design error in the circuit, so a warning could be helpful to the modeller.

5) **Static optimisation of the translator to the DEVS model**
Several optimisations can be made on a Logisim model, much the same as a compiler can perform several static optimisations while compiling. Besides the obvious replacement of blocks with only constant inputs, long connections can be resolved in a single step. Also, many (possibly conflicting) pull resistors can be converted to a single pull resistor. The generated DEVS models will only require computation as soon as their input changes, so the impact will be minimal in most situations.

6) **Use professional simulators instead of the internal Logisim simulator**
When the same semantics can be obtained using a different language, the translated models can be used to obtain information about the original models. Our PyDEVS simulator[6] for example, allows tracers to be used in different formats such as for example XML and Value Change Dump (VCD) which can be fed into a sophisticated VCD renderer. This way, it is possible to obtain a VCD trace file from a Logisim circuit. Other representations are also possible, depending on the simulator used.
The simulator may offer additional features such as parallel or distributed simulation.

III. Method

This section describes the core of the transformation. Logisim makes several implicit assumptions. These need to be made explicit in the corresponding DEVS models to ensure that identical simulation traces are produced. After these assumptions are made clear, the mapping of the basic components will be discussed, followed by the mapping of non-trivial components.

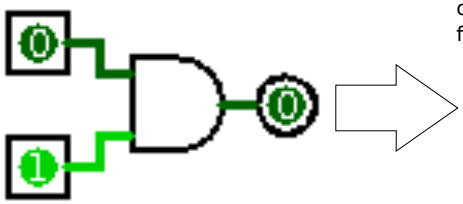
A. Assumptions

Several assumptions have to be made before translation can take place. The most important assumption relates to timing, as Logisim uses a very course model of time. All components compute their output at exactly the same time, and wires connecting components are assumed not to introduce delays. Note that this "synchronous" semantics does not match the Classic DEVS formalism's semantics well as internal transitions occurring at exactly the same point in time in different atomic DEVS models require tie-breaking and hence many (often artificial) invocations of the coupled DEVS `select` function. This problem is resolved by using Parallel DEVS[7].
There is a difference in the handling of time between Logisim and DEVS: Logisim uses discrete time, whereas DEVS uses continuous time. The time advance function of the DEVS models is used to return a fixed value, as to emulate the fixed timesteps. A clock model for example, which is simply an atomic DEVS model that has an internal transition to output the changed value. Afterwards, its time advance function will again return the value of the timestep.
This also means that our generated DEVS models will compute all outputs at exactly the same time, as was the case in Logisim, to guarantee identical simulation traces. DEVS does not have this limitation of discrete time, so it is possible to generate some jitter on the time advance function and thus allow non-simultaneous output generation.
In Logisim it is possible to assume that all input values of a component are valid, since the complete system model and its environment are built within the interactive syntax-directed modelling tool Logisim. This same assumption can not be made for our DEVS models, as it should be possible for the user to couple the generated (from Logisim) DEVS models to other DEVS models. Those other DEVS models are developed outside Logisim and may provide meaningless values (from the point of view of logic simulation) as input. To make sure that the input is valid, several checks must happen on *each* incoming event. These checks, which are virtually identical for each generated model, are not present in the shown code due to space restrictions. These checks include checking for compliant bitsize and whether or not the input value is a binary signal.
In DEVS, messages occur in a discrete fashion, so the model itself should remember the last input on each of its input ports. Hence, the *no event* input must be interpreted as *no change* and the last message received on a port should be taken into account for output computation. This implies state needs to be kept for every component, even though that component is not a logic circuit memory element. This allows several optimisations too, since a model could filter out the message if it matches the previously received event, which also means *no change*. Even this solution has problems, which are solved in section III-C1 on wires.

B. Simple mappings

Most Logisim components can simply be mapped to an atomic DEVS model which has the same semantics as the original component. There exist roughly two subgroups in this kind of components: the stateless and stateful components. Even though the name implies that there is no state, both will have a state due to the assumptions that were made. This naming refers to the ability of the component to depend on its state *in Logisim*.

1) Stateless components: Stateless components generate output independent of their previous input, only taking into consideration their current input. Mapping these to DEVS thus simply consists of setting the right input values in the state



```
class ANDb1c01i2n(AtomicDEVS):
 ...
 def outputFnc(self):
  ...
  outValue = 'b'
  for i in range(len(state.inputWires0)):
   if state.inputWires0[i] == '1' and state.inputWires1[i] == '1':
    outValue += '1'
   elif state.inputWires0[i] == '0' or state.inputWires1[i] == '0':
    outValue += '0'
   elif state.inputWires0[i] == 'E' or state.inputWires0[i] == 'x' or
        state.inputWires1[i] == 'E' or state.inputWires1[i] == 'x':
    outValue += 'E'
   else:
    outValue += '0'
  self.poke(self.outputWire, outValue)
```

Figure 2. The transformation rule for a basic AND gate, common code not shown

and a (small) part where the output value is computed and put on the output port(s). Since the configuration of the state is very general, all this logic can be generalised to (nearly) every component. The only component-dependent code will be the computation of the output. We chose to put this code in the output function, though it could alternatively be put in the external transition function. Note that the output function is not allowed to change the state in DEVS, while the external transition function is allowed to do so. It is not necessary to change the state if the output is immediately put on the output ports, so there is no need to alter the state in the output function, thus complying with the DEVS formalism.
The primary advantage of putting it in the output function is to avoid cluttering the external transition function and have a clear separation between input handling and the actual output generation of this input. Furthermore, should a component be interrupted during its computation, it would be possible to omit the computation entirely, while the computed result would just be thrown away otherwise[1]. Using the external transition function has several advantages too, mainly in Classic DEVS, as the external transition function can be parallellised whereas the output function cannot.
As a simple example, we show the mapping of the AND port to an atomic DEVS model in Figure 2. Note that the AND port is not as trivial as it could be (e.g., just a simple bitwise AND as provided in most programming languages), due to the possibility of an error and floating value. Also, Logisim offers many different options for the AND port (bitsize, number of inputs, negate several ports, semantics of signal, ...), further complicating the actual AND logic.

2) Stateful components: Stateful components's behaviour is dependent on their history, which causes them to be more complex to translate. This complexity doesn't necessarily arise from the mapping to DEVS, but from the lacking documentation provided for the components. What happens for example to the state of a component when an error value is read? Also, parts of the logic must be transferred from the output function to the external transition function, since it is not allowed in the DEVS formalism to change the state in the output function.
As an example, we will use the *Register* component. For this component, the output function is relatively simple since it only needs to copy the value from the memory and output it. To comply with bitsizes, the input is prepended by zeros (or whatever bit is prepended in Logisim) if needed. Most of the logic is now transfered to the external transition function, shown in Figure 3, which will be responsible for setting up the correct state of the component (so the output function can output the correct result).

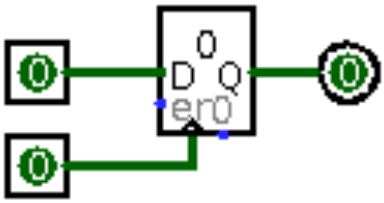
3) Subcircuits: Subcircuits naturally map onto a coupled DEVS model. As shown in Figure 4, they are translated to modular DEVS components allowing reuse. The inputs and outputs of the subcircuit are mapped to input and output ports in the Coupled DEVS model. Therefore, the original hierarchy is preserved in the transformation.

C. Complex mappings

Some constructs in Logisim cannot simply be mapped to a single DEVS model. Such constructs include colliding wires, pull resistors and the *splitter* element. To map these constructs to DEVS, we have to deviate from our previous 1-to-1 mapping. All these changes are implemented as preprocessing steps, since they have to happen before the actual model generation happens.
By dropping the 1-to-1 mapping, we introduce elements which do not have a direct counterpart in Logisim, thus making the trace files incompatible and obscuring the generated code. However, these artifacts are necessary to allow correct simulation in a relatively performant way.
Many of these problems are partially caused by a single port in Logisim being both an input and output port at the same time. This problem gets magnified due to a signal being able to 'go back' to where it came from and cause problems there. To solve these problems, all ports in a single connection are grouped on their direction (input or output). As soon as any rewriting of this connection needs to happen, this component needs to be generated in between this group, thus preserving the directionality of the signal. The values from the input signal will not be modified, though this is not a problem, since there is no input port connected to it to actually 'listen' to this value.

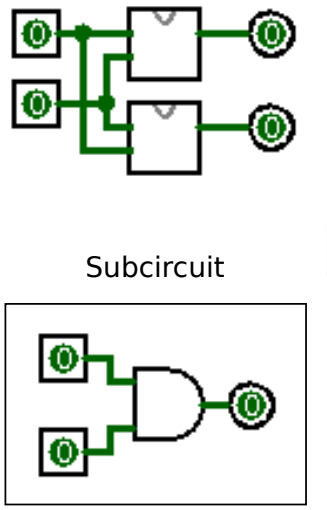
1) Wires: Wires cause a problem in DEVS. Since wires are mapped to connections between DEVS models, they also have to adhere to the discrete nature of message passing, while in reality, a wire is of course continuous. This presents two problems:

[1]As Logisim does not support the interruption of components, it is impossible to know the correct Logisim semantics for this situation.




```
class Registerb1i1trising(AtomicDEVS):
  ...
  def extTransition(self):
     ...
     state = Registerb1i1trisingState(active,
                inputWire[1:], clockInputWire[1:],
                extraInputWire[1:], resetInputWire[1:])
     state.triggered = triggered
     state.memory = self.state.memory
     state.timeFromTrigger = timeFromTrigger
     self.state.oldState = None
     state.oldState = self.state
     if state.resetInputWire == '1':
        state.memory = '0'
     elif state.extraInputWire != '0' and state.triggered:
        goodInput = True
        for i in state.inputWire:
           if i != '0' and i != '1':
              goodInput = False
        if goodInput:
           state.memory = state.inputWire
  def outputFnc(self):
    ...
    result = str(self.state.memory)
    while len(result) < 1:
      result = '0' + str(result)
    self.poke(self.outputWire, 'b' + str(result))
```

Figure 3. The transformation rule for a basic Register component, common code not shown




```
class main(CoupledDEVS):
   def __init__(self, name=None):
      ...
      from subcirc import subcirc
      self.importedCircuit0 = self.addSubModel(
            subcirc(name='importedCircuit0'))
      self.importedCircuit1 = self.addSubModel(
            subcirc(name='importedCircuit1'))
      self.input0 = self.addInPort(name='input0')
      self.input1 = self.addInPort(name='input1')
      self.output0 = self.addOutPort(name='output0')
      self.output1 = self.addOutPort(name='output1')
      self.connectPorts(self.importedCircuit0.output0, self.output0)
      self.connectPorts(self.importedCircuit1.output0, self.output1)
      self.connectPorts(self.input0, self.importedCircuit0.input0)
      self.connectPorts(self.input0, self.importedCircuit1.input1)
      self.connectPorts(self.input1, self.importedCircuit0.input1)
      self.connectPorts(self.input1, self.importedCircuit1.input0)

class subcirc(CoupledDEVS):
   def __init__(self, name=None):
      ...
      from ANDb1c01i2n import ANDb1c01i2n
      self.gate0 = self.addSubModel(ANDb1c01i2n(name='gate0'))
      self.input0 = self.addInPort(name='input0')
      self.input1 = self.addInPort(name='input1')
      self.output0 = self.addOutPort(name='output0')
      self.connectPorts(self.input0, self.gate0.inputWires0)
      self.connectPorts(self.input1, self.gate0.inputWires1)
      self.connectPorts(self.gate0.outputWire, self.output0)
```

Figure 4. The transformation rule for a Subcircuit

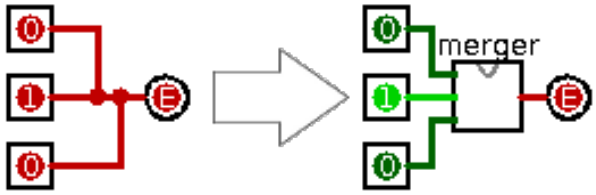


Figure 5. The necessary preprocessing for colliding wires

1) The value on the wire must be remembered in the atomic model itself, as it will not be repeated.
2) The values of colliding wires will overwrite each other, whereas for the DEVS model, there seems to be a sequential ordering to the incomming messages.

The second part is the primary concern here, since colliding wires should generate error and floating values if necessary. To circumvent this problem, an extra DEVS-only element is introduced, called a 'merger'. It takes in as many wires as necessary (determined at translate-time), remembers their values (to circumvent the first problem) and merges all values into a single output wire (circumventing the second problem). This is depicted in Figure 5 While wires are now considered as simple connections, it is also possible to map them to atomic DEVS models, for example to be able to model signal loss. We did not chose to do so, since Logisim does not support this functionality, but also to prevent huge performance degradation and to prevent additional cluttering of the tracing output (as every atomic model will be traced).

2) Pull resistors: A pull resistor is a component that has an effect on the complete wire, which again causes problems since a wire is merely a connection and does not contain any logic. This problem is solved by splitting all connected elements into a 'sending' and a 'receiving' group. These groups become seperated by a new atomic DEVS model which will perform the necessary pulling of the signal. Notice that the *output-to-input* mapping function of DEVS is not used, for multiple reasons:

1) PyDEVS, PyPDEVS and several other DEVS simulators do not support such a function.
2) It is possible that multiple pull resistors are present on a single wire, thus requiring these components to be taken together into a single component. Since this construction is now completely reduced to a simple component, performing a 1-to-1 mapping again seems more natural.
3) The generated component can be reused and complies better with all other translation problems that might occur (e.g., colliding wires with a pull resistor).
4) Introducing a new component allows some extra performance tuning, such as not recalculating and subsequently propagating an unchanged value. The pull register preprocessing is depicted in Figure 6.

3) Splitter: Splitters probably present the hardest part of the complete translation process. Mapping them onto an atomic DEVS model seems to be the straightforward solution. This is not possible due to the Logisim semantics. In Logisim, a splitter is a-causal, meaning that every port of the splitter can function as an input and output port at the same time, with

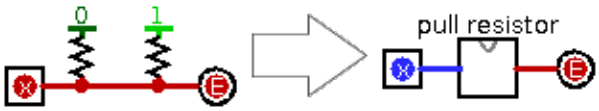


Figure 6. The necessary preprocessing for pull resistors

even possibly alternating roles at different times in the same model. Should an atomic DEVS model be written for this, taking into account all these possibilities, the model would become relatively big and there would be situations where the simulation becomes erroneous. When multiple splitters are connected to each other for example, they will transition in a sequential manner, possibly resulting in overwriting of previous values, while actually an error should be generated. This problem becomes even worse knowing that a splitter should have a zero *time advance*, due to it being an artifact from our transformation process. The only viable solution was to adhere to the Logisim interpretation of the splitter and consider all multi-bit signals to be a bunch of seperate wires in *every* part of the simulation. A naive implementation would cause several performance problems, since the number of messages passed would increase immensly and some processing would be necessary in each component to split and merge the signals back together.
For these reasons, a hybrid approach was taken, only splitting and subsequently merging the wire if a splitter is present on the connection. The connections between the splitters are resolved at translate-time and the meaning of a splitter gets reduced to a pair of a bitmerger and a bitshredder, with the appropriate connections in between. With Logisim semantics it is also possible to remap the input/output ports with the change of a configuration setting, so these situations are also resolved.
It should be clear that this transformation is partially an optimisation in case lots of splitters are used, since all this routing no longer has to happen at simulation time. All that is required at simulation time is splitting a string in seperate characters and subsequently merging them back together later on. The preprocessing for a splitter is depicted in Figure 7.

4) Illegal couplings: In the DEVS formalism, several couplings are deemed illegal due to them being either useless (connect an input port of a coupled model to the output port of the same coupled model) or dangerous (algebraic loops). However, all these situations are valid in Logisim.
To provide a complete mapping between both semantic domains, an extra element needs to be inserted to trick the simulator in thinking that the model complies to these specifications. This model will be the *Null* model, which merely copies its input value to its output port with a zero delay. This is by no means a clean solution as these situations most likely indicate a problem in the model. It is only included for completeness.

5) Putting it together: The translation becomes more of a problem if multiple difficult cases occur together. In such cases, the order in which these problems are resolved is important to prevent problems in the assumptions made when fixing these problems. This correct order is:

1) **Remove splitters**
A splitter must be removed as soon as possible. This is because it must be interpreted as a wire and not as a

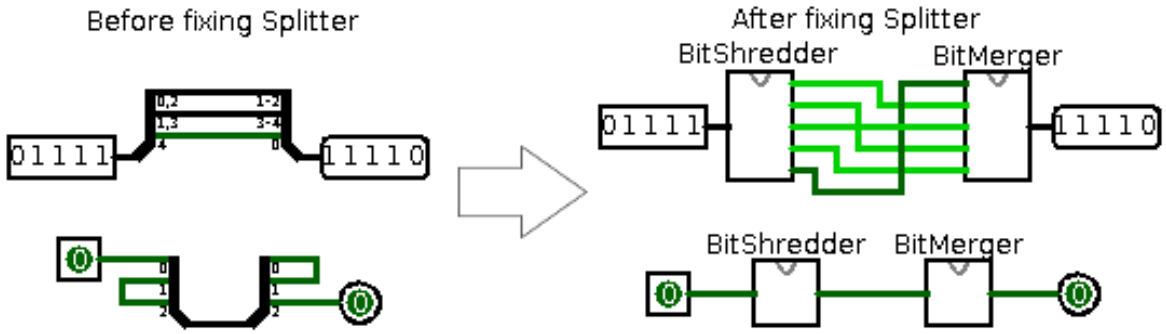


Figure 7. The necessary preprocessing for a splitter in different situations

model, so it must also propagate all special situations, such as pull resistors and colliding wires. When a splitter is solved, its complete connection is traversed, which allows us to decide the direction in which the wire has to be pulled in case of a floating value. In the case that a pull resistor is placed on the multi-bit wire and on the split wire, they must be checked for compatibility and the direction must be set to an error bit, if necessary.

2) **Merge colliding wires**
There is an overlap between a merger and a pull resistor, which causes them to be conflicting when they are not applied in the correct order. The Logisim semantics states that a pull resistor will set the value on a wire to the specified value, if the value on the wire is floating. A merger on the other hand, will merge wires and if one of the c olliding wires is floating, it will be set to the value of the other connected wires. This means that a pull resistor should not set the value of a wire that could still collide with another. Only if there is still a floating value after a merge (in case all wires are floating), should the value be set by a pull resistor.
3) **Solve pull resistors**
This should happen near the end as it is only seen as a 'last resort' for filling in a value on a wire.
4) **Inserting the Null model where needed**
The Null model is only inserted because it is necessary to allow simulation. Therefore, it has no priority and even has no meaning. The reason why it is last is simply due to performance. It could be possible that the translator identified a possible situation where a Null element is required, though it is no longer needed after one of the previous transformations. One such situation would be when wires that would be illegal in DEVS collide, thus creating a Merger model, which will then fulfill the role of the Null model.

IV. Semantics and Testing

One of the reasons to perform this transformation to DEVS was to obtain insight in the semantics of Logisim. This differs from the approach in [8], where a completely new DSL was designed as a library of DEVS components and the semantics were created anew. We comply with the existing semantics of Logisim. This causes some distinct difficulties, as we need to extract the semantics from Logisim and replicate this in our generated DEVS models.
With only the limited amount of information available in the Logisim Library Reference, we are unable to create semantically equivalent models. Therefore, explicit simulation and observation of behaviour traces is necessary to guarantee equivalence. To automate this task, we created an exhaustive test suite for every basic component. The most important variations in configuration of this component are placed next to each other and every possible combination of inputs for this component is provided. The output is logged with the (very basic) logging functionality provided by Logisim. Subsequently, the logged circuit is translated to DEVS and simulated again. The DEVS simulation creates an XML behaviour trace. This trace file is converted to the same format as that generated by Logisim. Both files are compared and the test passes if both files are identical.

V. Performance

As described above, mapping Logisim to DEVS has several conceptual advantages. The question remains however how behaviourally equivalent DEVS and Logisim models compare in terms of simulation speed. To get some insight into simulation performance, we use our translator to transform a relatively complex Logisim circuit to its DEVS equivalent and simulate it using our PyDEVS simulator. Tests were performed on an Intel i5-2500 3.3GHz with 4GB RAM, with Logisim v2.7.1 and PyDEVS v1.1.

A. Model

In order to prevent a bias due to a single component being implemented far more efficiently in either Logisim or in the translated model, we chose to implement a basic 16-bit ALU shown in Figure 8. It makes uniform use of all the most important core components from Logisim (logic gates, arithmetic, shifting, ...). To increase the size of the model, we put several ALUs in parallel as shown in Figure 9. All these subcircuits will receive the same input (and will thus generate the same output). To prevent further bias from the instructions fed into the ALUs, we filled the ROM components containing the input data (except the *instruction*) with random data before starting the simulation. This data was only inserted once and is thus identical for each simulation run.

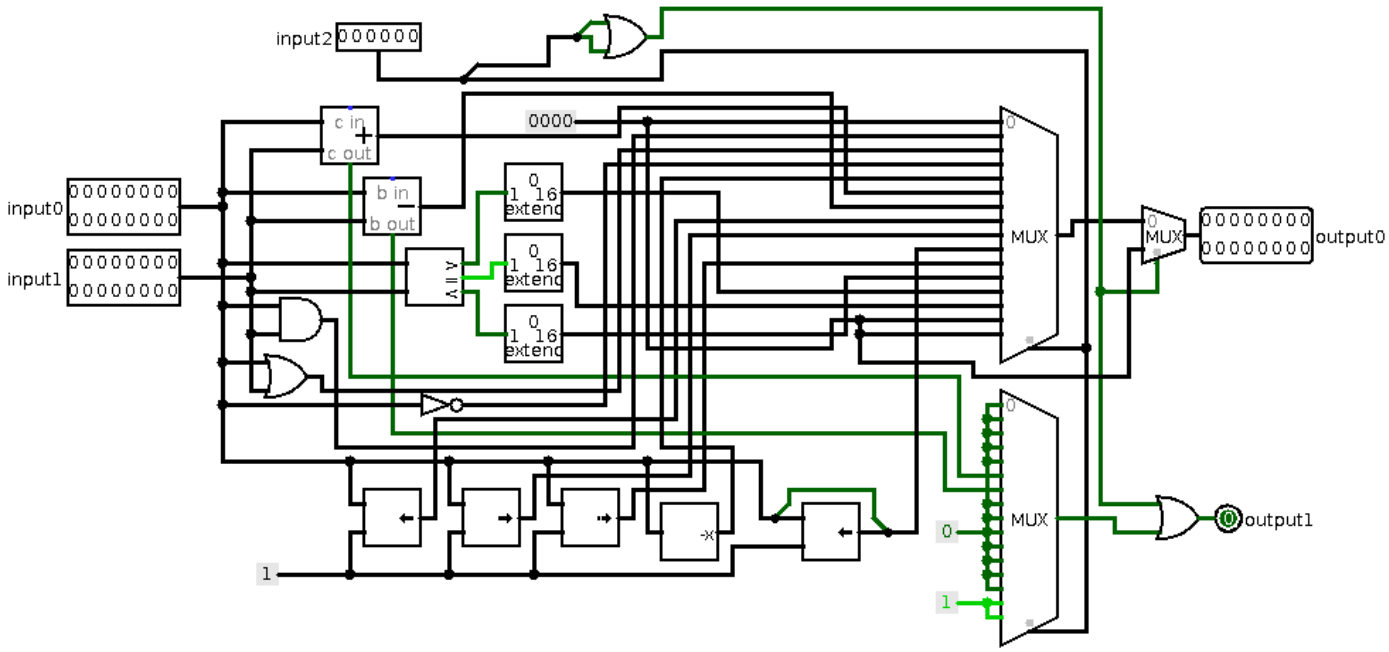


Figure 8. A basic ALU subcircuit using Logisim's core components

B. Results

Figure 10 depicts the running time of a Logisim simulation and that of the corresponding DEVS simulation for increasing model sizes (i.e., number of ALUs). Note that we use our Parallel DEVS simulator (PyPDEVS) to avoid the need for the *select* function when simultaneous internal transitions occur. We use the PyPy just-in-time interpreter as it yields (slightly) better performance than the standard Python interpreter. Logisim does not support automatic simulation and timing of such simulations, so these were done manually. Therefore, the Logisim timings are only accurate up to one second. A first observation is that both simulations have $O(n)$ complexity. Our translation from Logisim to DEVS does hence not introduce artificial complexity. Simulation of the generated DEVS version is however on average a factor 3.9 slower than that of the Logisim original (as noticed in the difference in slopes of the two performance curves). There are several reasons for this performance difference. First of all, Logisim is written in Java, whereas our DEVS simulator is written in Python. The most important cause for the performance difference is however the generality of DEVS compared to the domain-specificity of the Logisim logical circuit simulation. While the Logisim simulator can make several assumptions about the simulation (e.g., timing of components, collission resolution, ...), the DEVS simulator has no such knowledge and is thus unable to provide the same level of performance. Note that we did not choose to use multiple cores for the PyPDEVS simulation, since the computation of the transition functions does not take much time. This would actually slow down the simulation due to the setup and destruction time of the process, certainly because the actual transition functions are relatively small[2]. Though slower, the use of DEVS allows one to use different simulators than the one provided in Logisim, possibly even a parallellized and distributed simulator (should it be applicable to the generated model).

VI. Related work

Another approach to the mapping between digital logical circuits and DEVS (in particular, DEVS-Suite) is presented in [8]. Our approach differs in that we *symbolically transform* arbitrary Logisim models instead of providing a *library* of DEVS models for the common components in logic circuits. The DEVS models for common logic components are very similar in both approaches. In our transformation however, we faithfully reproduce Logisim semantics, including all its idiosyncracies.
The ability to model and simulate the possibly non-deterministic *environment* (modelled in DEVS) in which the system under study operates was also the reason behind the transformation presented in [9]. In that work, UML2.0 Class Diagrams + Statecharts were transformed to DEVS.

VII. Conclusions and Future Work

In this paper, a transformation from Logisim models to behaviourally equivalent DEVS models was presented. The transformation rules were described and implemented. The translator implementation as well as the DEVS simulator used and the models used for performance analysis are available at [10]. The behavioural equivalence between Logisim models and their generated DEVS counterparts was exhaustively tested. Also, some performance comparisons were made between native Logisim simulation and that of the generated

[2]In Python, threads are unable to use true parallellism due to the global interpreter lock. If true multicore performance is desired, it is necessary to use multiple processes which will all run in a seperate Python interpreter. Needless to say, this has a high overhead.

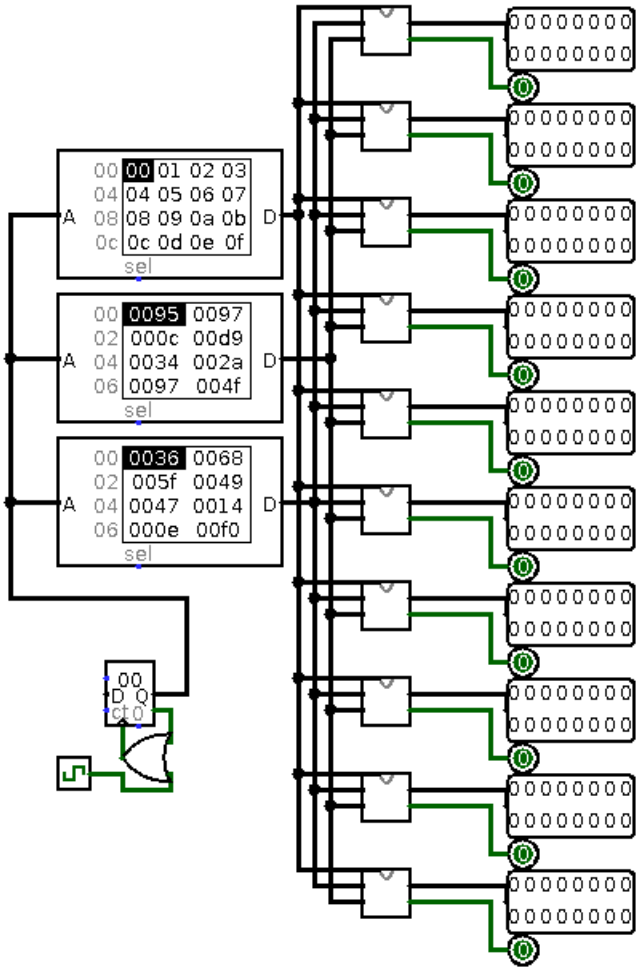


Figure 9. The main model, containing sub-circuits

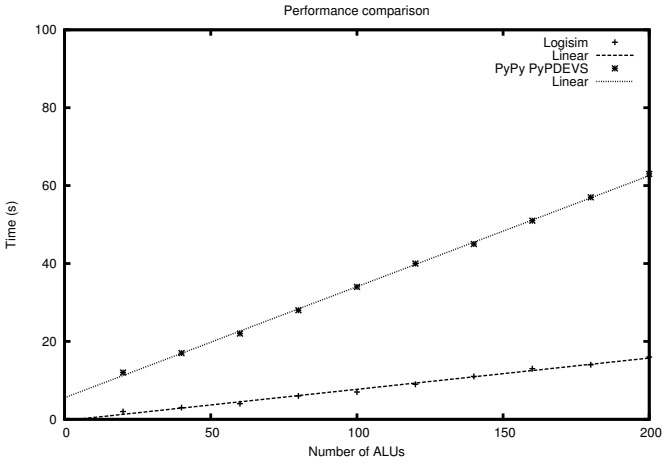


Figure 10. Performance of Logisim and generated DEVS

DEVS models. The simulation of the generated DEVS models using our PyPDEVS simulator (with the PyPy interpreter) was on average 3.9 times slower than native Logisim simulation. This result is not surprising as Logisim is specifically optimized for the domain of logic circuit simulation whereas DEVS and its simulators are generic.

Crafting the Logisim to DEVS transformation did give us insight into the intrictate details of the semantics of the Logisim components. Another advantage of transformation to DEVS is that a Logisim circuit can thus be embedded in a larger DEVS model. This allows for example coupling of the circuit to a model of the environment in which it will be used. The transformation also allows us to use Logisim as a domain-specific visual modelling environment yet use any DEVS-compliant simulator.

In the future, we plan to bridge the (performance) gap between a dedicated simulator such as Logisim and generic simulation in DEVS. A first approach is to transform a Logisim model to a single atomic DEVS model, including the domain-specific (discrete-time) simulation techniques. This is similar to the approach we took in [9] when translating Statechart models to DEVS. A second, and more general approach is to use activity-enhanced simulation [11]. In this approach, the DEVS model is enhanced with a model of computational load and the generic DEVS simulator is augmented with capabilities to use these "hints" to increase performance. We believe that with appropriate hints, the DEVS simulator will be able to use the domain-specific information and achieve performance matching that of a dedicated domain-specific simulator.

ACKNOWLEDGMENT

Naomi Christis' work on a Logisim to Verilog translator, the starting point for our contribution, is gratefully acknowledged.

REFERENCES

[1] C. Burch, "Logisim v2.7.1," http://ozark.hendrix.edu/~burch/logisim/, March 2011.

[2] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation*, 2nd ed. Academic Press, 2000.

[3] J.-S. Bolduc and H. Vangheluwe, "Expressing ODE models as DEVS: Quantization approaches," in *Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems)*, F. Barros and N. Giambiasi, Eds. Society for Modeling and Simulation International (SCS), April 2002, pp. 163 – 169, Lisboa, Portugal.

[4] H. Vangheluwe and G. Vansteenkiste, "The cellular automata formalism and its relationship to DEVS," in 14^{th} *European Simulation Multi-conference (ESM)*, R. Van Landeghem, Ed. Society for Modeling and Simulation International (SCS), May 2000, pp. 800–810, Ghent, Belgium.

[5] "IEEE Standard for Verilog Hardware Description Language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[6] J.-S. Bolduc and H. Vangheluwe, "The modelling and simulation package PythonDEVS for classical hierarchical DEVS," McGill Univ., Tech. Rep., 2001.

[7] A. C. H. Chow and B. P. Zeigler, "Parallel DEVS: a parallel, hierarchical, modular, modeling formalism," in *Proceedings of the 26th conference on Winter simulation*, ser. WSC '94. San Diego, CA, USA: SCS, 1994, pp. 716–722. [Online]. Available: http://dl.acm.org/citation.cfm?id=193201.194336

[8] Y. Chen and H. S. Sarjoughian, "A component-based simulator for MIPS32 processors," *Simulation*, vol. 86, no. 5-6, pp. 271–290, 2010.

[9] R. Shaikh and H. Vangheluwe, "Transforming UML2.0 Class Diagrams and Statecharts to atomic DEVS," in *SpringSim/TMS-DEVS*, 2011, pp. 205–212.

[10] Y. Van Tendeloo, "Logisim to DEVS translator," http://msdl.cs.mcgill.ca/people/yentl, 2012.

[11] A. Muzy, L. Touraille, H. Vangheluwe, O. Michel, D. R. Hill, and M. K. Traoré, "Activity regions in discrete-event systems," in *SpringSim/TMS-DEVS*. SCS, April 2010, pp. 176 – 182.