Discrete-Time Causal Block Diagrams Assignment

1 Discrete-Time Causal Block Diagrams

Discrete-Time Causal Block Diagrams (DT-CBDs) are used to model dynamic (*i.e.,* time-varying) systems at a discrete-time abstraction level. Their denotational semantics is given in terms of difference equations.

More information about CBDs (than in the MoSIS lectures) can be found in [1].

The purpose of this assignment is that you build a Discrete-Time CBD simulator by completing the code already provided as a starting point. You subsequently use your simulator to build and simulate a few simple models.

2 Discrete-Time CBD simulator

Implement a CBD simulator for Discrete time CBDs by filling in what is missing in the provided source files. Take the time to read the `README.*.md` files, and go through the folder structure and the code to understand the general architecture of the simulator. The Python script that you have to complete is the `CBD.py`.

The main classes in `CBD.py` are:

- `BaseBlock` class – An abstract class that represents a Block in a CBD. Specific blocks such as `ConstantBlock` extend this class.
- Specific block classes – A set of classes, each extending `BaseBlock` and representing a specific Block in a CBD.
- `Clock` – A class whose instance is used to keep track of the simulated time in a simulation. As this is a Discrete-Time simulator, the clock is incremented in steps of 1.
- `CBD` – A class representing an entire CBD diagram.
- `DepGraph` – Represents a dependency graph. This dependency graph is used to determine the order of evaluation of the blocks at each simulation step.
- `DepNode` – Represents a node of the dependency graph.

The `EvenNumberGen` example shows a hierarchical CBD that computes the sequence of even numbers. It can be used as a starting point for your solution. Read the `README.*.md` files for instructions on how to use both the drawio to CBD translator and the CBD simulator.

Look for the `TO IMPLEMENT` comments in the `CBD.py` and implement the following functions:

1. `compute` function of several blocks. These functions should compute the output value of the block at the iteration `curIteration`. Look at the code of the `BaseBlock` class to understand how the inputs and outputs are represented.

2. `getDependencies` function of the `DelayBlock` class. This function should override the `getDependencies` function of the `BaseBlock` class (which is given) because delay blocks have different dependencies depending on the current iteration.

3. `__isLinear` of the `CBD` class. This function should detect whether a strong component - representing an algebraic loop - is a linear algebraic loop or not.

4. Do not worry about implementing the `DerivatorBlock` and `IntegratorBlock` classes. For now.

3 CBD Simulator testing

Use the supplied unit tests (see `README.CBD.md`) to test your implementation. All tests except the ones for Derivative and Integrator block should pass.

4 Discrete-Time CBD Denotational Semantics

Implement a LaTeX generator for the equations that form the model's denotational semantics. Note that you may make use of `getPath()` which will give the "full" name of a block (and hence, its output signal). "Full" mean, the full path to the root of the model. This guarantees unique names. Show that your LaTeX generator works on the models in the `examples` directory as well as on the models described further in this document.

The generated LaTeX might look like documentclassarticle

```
\begin{document}
 \[
  \left\{
   \begin{array}{rcl}
    v(i) & = & w(i) - 2\\
    w(i) & = & w(i-1) + v(i-1) + 3\\
   \end{array}
  \right.
 \]
\end{document}
```

which gets rendered as

$$\begin{cases} v(i) &= w(i) - 2 \\ w(i) &= w(i-1) + v(i-1) + 3 \end{cases}$$

or in difference equation notation

```
\begin{document}
 \[
  \left\{
   \begin{array}{rcl}
    v_i & = & w_i - 2\\
```

```
    w_i & = & w_{i-1} + v_{i-1} + 3\\
   \end{array}
  \right.
 \]
\end{document}
```

which gets rendered as

$$\left\{ \begin{array}{rcl} v_i & = & w_i - 2 \\ w_i & = & w_{i-1} + v_{i-1} + 3 \end{array} \right.$$

5 Pseudo-Random Number Generator

Build a model for the following Linear Congruential Generator (LCG):

$$x_{i+1} = (ax_i + c)\%m$$

where % stands for modulo division.

simulate this model for $a = 4, c = 1, m = 9$. Determine this generator's period (the number of steps before it starts repeating itself), and this for different seeds $x_0 \in \{0, 1, \ldots, 8\}$.

6 Explicit vs. Implicit Equations

Explicit:

$$\left\{ \begin{array}{l} x_{i+1} = x_i + Dy_i \\ y_{i+1} = y_i - Dx_i \end{array} \right.$$

Implicit:

$$\left\{ \begin{array}{l} x_{i+1} = x_i + Dy_{i+1} \\ y_{i+1} = y_i - Dx_{i+1} \end{array} \right.$$

For both models, $x_0 = 0, y_0 = 1$.

Simulate both models with D first equal to 0.001 and then equal to 0.1. For both D values, simulate $2\pi/D$ steps. Plot all results.

For all simulations, compare the values of x_i with those of $sin(iD)$. Note that you need to use a `GenericBlock` to model $sin()$. Have a look in `examples/SinGen`.

What can you conclude about the influence of (1) D and (2) explicit vs. implicit?

7 Document

Write a small report containing the tasks that you have completed and how you have completed them. Include all the plots that you have made from the simulations. The CBD models that you have built in the previous tasks should be displayed graphically in the report.

References

[1] Cláudio Gomes, Joachim Denil, and Hans Vangheluwe. *Causal-Block Diagrams: A Family of Languages for Causal Modelling of Cyber-Physical Systems*, chapter 4, pages 97–125. Springer International Publishing, 2020.