Causal-Block Diagrams

Cláudio Gomes[‡†], Joachim Denil[‡†], and Hans Vangheluwe[‡†§]

[†]University of Antwerp, Belgium
[‡]Flanders Make, Belgium
[§] McGill University, Canada

Abstract. The description of a complex system in terms of constituent components and their interaction is one of the most natural and intuitive ways of decomposition. Causal Block Diagram (CBD) models combine subsystem blocks in a network of relationships between input signals and output signals. Popular modeling and simulation tools such as Matlab/Simulink® implement different variants from the family of Causal Block Diagram formalisms.
This chapter gives an overview of modeling and simulation of systems with software and physical components using Causal Block Diagrams. It describes the syntax and - both declarative and operational - semantics of CBDs incrementally. Starting from simple algebraic models (no notion of time), we introduce, first a discrete notion of time (leading to discrete-time CBDs) and subsequently, a continuous notion of time (leading to continuous-time CBDs). Each new variant builds on the previous ones. Because of the heavy dependency of CBDs on numerical techniques, we give an intuitive introduction to this important field, pointing out main solutions as well as pitfalls.
After reading this chapter, the reader will be able to judge when to use the CBD formalism and how to use it, as well as the main issues often encountered with the description of physical systems and the implementation of CBD simulators.

1 Introduction

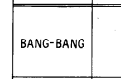
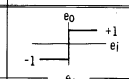
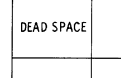
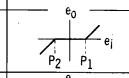
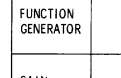
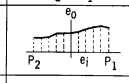
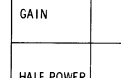
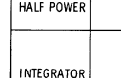
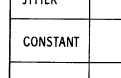
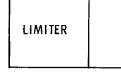
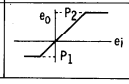
The design process of complex systems, aided by the technology advances in the last century, is rapidly shifting from small scale development of isolated systems, to large scale development of integrated systems [1, 6, 14].

The graphical representation of a system using blocks and arrows is one of the first methods used to represent systems. One of the benefits of this notation is that complex systems can be hierarchically decomposed into sub-systems, thus providing a way to deal with complexity. Causal Block Diagrams (CBD) is a formalization of this intuitive graphical notation.

Originally, CBDs were widely used to represent analog circuits [1, 13, 2, 8] (see the blocks commonly used in Table 1). Nowadays, this formalism is widely used in the development of systems that comprise physical and software parts, as in the system depicted in Fig. 1. In this kind of system, the software monitors the activity of physical processes by means of sensors, takes appropriate decisions,

and influences the physical processes through actuators. This architecture can be generalized to networked software and to any processes, not necessarily physical. For the purposes of introducing the formalism, we hold on to the traditional view of a physical process being controlled by a software component, also known as a feedback control system.

Table 1. Block representation for analog circuits. Reproduced from [3].

ELEMENT TYPE	LANGUAGE SYMBOL	DIAGRAMMATIC SYMBOL	DESCRIPTION
BANG-BANG	B		
DEAD SPACE	D		
FUNCTION GENERATOR	F		
GAIN	G		$e_0 = P_1 e_i$
HALF POWER	H		$e_0 = \sqrt{e_i}$ SQUARE ROOT
INTEGRATOR	I		$e_0 = P_1 + \int (e_1 + e_2 P_2 + e_3 P_3)\, dt$
JITTER	J		RANDOM NUMBER GENERATOR BETWEEN ±1
CONSTANT	K		$e_0 = P_1$
LIMITER	L		

n REPRESENTS THE BLOCK NUMBER

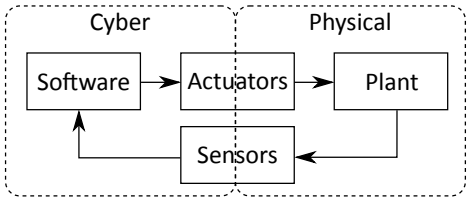


Fig. 1. Generic embedded system structure.

In the next section, a simple running example will be introduced, as well as some necessary background concepts. In the remaining sections, CBDs are

introduced gradually in three different flavors: algebraic, discrete time, and continuous time CBDs. These are distinguished by the class of blocks at the disposal of the modeler. The gradual presentation allows for a deeper understanding of all the concepts related to modeling and simulation of CBDs. The last few sections of the chapter deal with advanced concepts, related to the simulation of CBDs.

2 Background

A dynamical system is characterized by a state and a notion of evolution rules. The state is a set of point values in a state space. The evolution rules describe how the state evolves over an independent variable, usually time.

The domain of the time variable dictates the kind of dynamical system: if time ranges over a continuous set, usually $\mathbb{R}$, then the dynamical system is continuous time. Similarly, if time ranges over a countable set, usually $\mathbb{N}$, then the dynamical system is discrete time.

2.1 Cruise Control System

Consider the cruise control system depicted in Fig. 2. The car is a dynamical system whose state (e.g., the velocity) evolves continuously in time. The state of software controller, on the other hand, evolves discretely through time, as any software variable does, by consequence of assignment instructions.

The system in Fig. 2 is an example of a feedback control system: the physical system – the car – is actuated by control inputs generated by control software – the cruise controller. The sensors are the tachometers that translate wheel revolutions per minute into instantaneous velocity. The actuators are the motor throttle and brakes. The cruise controller software decides, based on current speed of the car, which amount of traction (throttle or brakes) should be applied to restore the car to a desired speed despite drag, weight, and other factors.

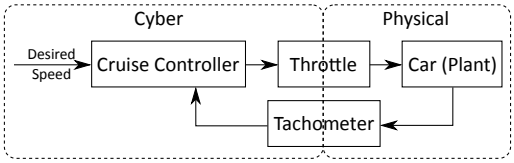


Fig. 2. Cruise control system.

Because of cost-efficiency, security, it is not desirable to wait for a car prototype to be built, in order to test the control software. This motivates the need for two dynamical systems:

1. A continuous dynamical system which acts as a mock-up of a real car.

2. A discrete time dynamical system which acts as a mock-up of the software unit.

With these models, early evaluation of many different control strategies can be performed, at a much lower cost. Furthermore, the chosen controller model can then be used to automatically generate the software code.

2.2 Models of Physical Systems

Physical systems are inherently continuous: their state evolves continuously through time. Ordinary Differential Equations (ODE) describe how physical quantities change continuously in time. ODEs are thus ideal models to describe physical systems' behavior. They take the form:

$$\begin{aligned} x'(t) &= F(x, u, t) \\ y(t) &= G(x, u, t) \\ x(0) &= x_0 \end{aligned} \tag{1}$$

where $x(t), x'(t), y(t), u(t)$ are vectors, $x(t) = [x_1(t), \ldots, x_n(t)]^T$ represents the state vector, $u(t)$ represents the input, and x_0 is the initial state. The state here denotes the minimal information required to determine, together with the input $u(t)$ and function $F(x, u, t)$, the complete future states of the system. Function G is the output function.

A possible model of the dynamic behavior of the car, to be used in the design of a cruise controller, is:

$$\begin{aligned} v' &= \frac{1}{m}(T - kv) \\ y &= v \end{aligned} \tag{2}$$

where v is the velocity, v', T is the traction force and k depends on the air density and car shape. This model assumes that the car moves in a straight line and neglects any effects that gravity might induce. Reasonable assumptions for early experimentation.

2.3 Discrete Time Models

While the solution of ODEs is continuous, the state of the software unit, in the cruise control system presented in Section 2.1, can only evolve discretely, by the nature of the digital computer on which it runs. To see why this is the case, suppose that the state of the control system takes values in $\mathbb{R}$ and evolves continuously over the time variable. This means that the state, over a finite interval of the time variable, may assume an infinite number of different values. In software running on a digital computer, variables get new values, computed from old ones, through assignment instructions, which take non-zero *wall-clock time* to execute. As a consequence, we would have to wait an infinite amount of wall-clock time for the computer to assign the infinite number of different values to state variables.

For mathematical analysis purposes, differential equations may be used for an early specification of the control software. However, when it comes to simulating those models in a digital computer, the only available option is to use discrete time models.

First order difference equations allow the specification of such models. They take the form

$$\begin{aligned} x^{[s+1]} &= F(x^{[s]}, u^{[s+1]}) \\ y^{[s]} &= G(x^{[s]}, u^{[s]}) \\ x^{[0]} &= x_0 \end{aligned} \tag{3}$$

where s denotes the step, $x^{[s]}$ is the state vector at step s, $u^{[s+1]}$ is the input vector, $y^{[s]}$ the output vector, and x_0 the initial value of the state vector. The new vector $x^{[s+1]}$ is computed from the old one $x^{[s]}$ and input $u^{[s+1]}$, according to the specification function F. The output function G allows values to be read from the dynamical system. The repeated application of functions F and G yields the discrete evolution of the state and output vectors. Difference equations can also be written as

$$\begin{aligned} x^{[s]} &= F(x^{[s-1]}, u^{[s]}) \\ y^{[s]} &= G(x^{[s]}, u^{[s]}) \\ x^{(0)} &= x_0 \end{aligned}$$

Obviously, the two representations are equivalent.

The cruise control software can be described by the following difference equation:

$$\begin{aligned} e^{[s+1]} &= e^{[s]} + h\left(v_d^{[s+1]} - v^{[s+1]}\right) \\ T^{[s]} &= K_p\left(v_d^{[s]} - v^{[s]}\right) + K_i e^{[s]} \end{aligned} \tag{4}$$

where v_d is the velocity that the car should be kept at (input); v is the actual velocity (input) of the car; $\left(v_d^{[s+1]} - v^{[s+1]}\right)$ is the instantaneous error; $e^{[s]}$ denotes the accumulated error (state); $T^{[s]}$ is the traction force to be transmitted to the car (output). Finally, K_p and K_i are constant parameters of the controller.

The controller gets the car velocity input $v^{[s]}$ from the readings of the tachometer (recall Fig. 2). If the differential equation Eq. (2) is used to model the car, then $v(t)$ is a continuous quantity and so we can relate it to the input of the controller by $v^{[s]} = v(s \times \Delta t)$, where Δt denotes the constant interval of time between two successive tachometer readings.

Intuitively, the traction force $T^{[s]}$ is proportional to the instantaneous error $v_d^{[s]} - v^{[s]}$ and to the accumulation of errors $e^{[s]}$ from previous steps. From now on, assume that Eq. (2) is being used in place of the real car to test the controller proposed in Eq. (4). The following two paragraphs give an intuitive rationale for each component of Eq. (4).

$K_p\left(v_d^{[s]} - v^{[s]}\right)$ *component.* When the instantaneous error is large, the current velocity is far away from the desired one, so the traction force should be large in

order to ensure that the car quickly accelerates/brakes toward the desired speed. When the instantaneous error is small, the car is almost at the desired speed, so the traction force should be smaller to avoid causing discomfort to the driver. For now, neglect the $K_i e^{[s]}$ component in the traction force calculation. After a while, if the traction force given by $T^{[s]} = K_p \left(v_d^{[s]} - v^{[s]} \right)$ becomes symmetric (same magnitude, opposite direction) with the drag force in Eq. (2), the car acceleration will be null and its speed will be kept constant. However, that speed will not be exactly equal to the desired speed, because $-kv \neq 0 \implies \left(v_d^{[s]} - v^{[s]} \right) \neq 0$. This is where the $K_i e^{[s]}$ component, neglected until now, has its use.

$K_i e^{[s]}$ component. This component accumulates the instantaneous error over time, and contributes to the traction force accordingly. Suppose that the traction force is counteracted by the drag force and the car is kept at a constant speed, below the desired velocity, just like in the previous paragraph. Then, the accumulated error will keep growing, ensuring that the second component continues to increase the traction force until it overcomes the drag force. Notice that this might cause the car to overshoot the desired speed, which can be dangerous. The accumulated error will start decreasing once that happens, decreasing the traction force. The choice of parameters K_p and K_i is an important part of tuning the controller.

2.4 Summary

This section has introduced a running example, to be used throughout the current chapter. The example represents a typical feedback control system, the majority of which are developed with the aid of Causal Block Diagrams (CBDs). Making use of the example, two kinds of mathematical models of systems where introduced: differential equations for continuous systems (e.g., the car), and difference equations for discrete systems (e.g., the software cruise controller). Knowledge of differential and difference equations ensures that the formal meaning of the different flavors of Causal Block Diagrams will be understood in sections that follow.

3 Algebraic Causal Block Diagrams

Algebraic Causal Block Diagrams (CBDs) are CBDs in which the only atomic blocks permitted are algebraic ones: summation, negation, inversion, product, raise to power and roots. These can be used to represent systems where there is no notion of time and no notion of evolving state. In other words, the time is a constant *now*.

While it may seem restricted, this kind of systems arise in the study of the steady state behavior of dynamic systems. As an example, consider the car dynamics in Eq. (2). The steady state behavior of the car happens when it is

not accelerating. That is, for known constants T, k:

$$0 = \frac{1}{m}(T - kv)$$

The equation gives insight about the torque required to keep the car at the same speed: $T = kv$. The larger the drag force, the larger the torque, and the more energy is required.

3.1 Syntax

The main constituents of a CBD are blocks and connections between blocks. Blocks can be atomic or composite. Composite blocks stand for an external CBD, specified elsewhere. These blocks will be drawn with a dashed contour. In algebraic CBDs, atomic blocks can be summation, negation, inversion, product, raise to power, and roots. These will be denoted with the appropriate mathematical symbol.

Since a block can have more than one input and more than one output, the notion of ports is essential to distinguish between inputs and between outputs. Atomic blocks have up to two input ports - depending on the operation - and one output port. Composite blocks can have any number of input and output ports.

Fig. 3 shows an example of an Algebraic CBD, that calculates the drag force d affecting a car, as it moves with a velocity v, given as input. The composite block c refers to an algebraic CBD that calculates the drag coefficient, detailed in Fig. 4.

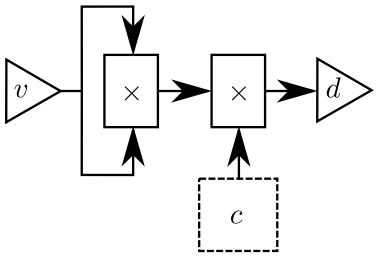


Fig. 3. Algebraic CBD of the drag force block.

The ports associated with a block will not be drawn explicitly but they are part of the CBD and have identifiers (ids). The directed connections will make clear which input ports and output ports are associated with a block. When there is a need, the input port id is shown at the border of the associated block. In the specification of a composite block, the input and output ports are represented as triangles. Whether the port is input or output is clear from the context.

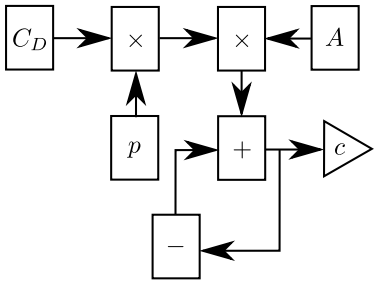


Fig. 4. Algebraic CBD of the drag coefficient block, used in the diagram of Fig. 3.

Blocks are also referred to by ids. The identifier of a port is comprised of the name of the port, and the identifier of the parent block. The identifier of a block is formed by its name, along with the identifier of its parent CBD. Note the recursive definition. While the identifier is almost never visible in the graphic representation, it is always defined.

Some of the uses of identifiers are: the unambiguous description of connections between ports and the unambiguous identification of individual blocks after the flattening process (Section 3.2).

More often, the names of the blocks will be depicted in the graphical representations, to enhance the readability. Names are not identifiers, they are a part of the identifier. For instance, the product blocks in Fig. 3 have two ports with distinct identifiers, even though these are not depicted in the graphical representation. In the same picture, the name v of the input and the name d of the output port are shown. Similarly, the name c of the composite block is shown. The reader may notice that in Fig. 4 the same name c denotes the output port. There is no ambiguity as the composite block and the output port have distinct identifiers, even though they have the same name.

Whenever a composite block is used, all its internal blocks adopt different identifiers, based on the id of the CBD where the composite block is used. For instance, the fact that the composite block c is used in the CBD of Fig. 3 means that, when processing that CBD, the identifiers of the inner blocks/ports of c (detailed in Fig. 4) include the identifier of c. This has two important consequences:

1. the identifier of any element depends ultimately on where it is being used, or where any of its parents are being used;
2. if the composite blocks are replaced by their specification in a *flattening* process, there will be no two identifiers alike, thus ensuring the well formedness of the CBD.

3.2 Semantics

The meaning of an algebraic CBD is an association of a value to each of the ports in the CBD. It can be conveyed in two general ways: by writing the mathematical

equations that correspond to the CBD (translational semantics), or by giving an algorithm which computes the value associated with each input/output port (operational semantics).

To simplify both these approaches, it is assumed that all composite blocks are replaced by their specification in a *flattening* process. This process is done recursively until all composite blocks have been replaced by their specification [10]. The following aspects are important to ensure the well-formedness of the flattened CBD:

1. After replacing a composite block, their input/output ports (e.g., the triangular ones in Fig. 4) will be connected from both sides. These are redundant ports and hence substituted by a single connection.
2. The identifier of the replaced composite block is still part of the identifiers of its inner blocks/ports. This ensures uniqueness among identifiers after the flattening process is complete.

Fig. 5 shows the result of replacing the composite block c with its specification (in Fig. 4). The identifiers, shown explicitly in the picture, contain the identifier of the composite block replaced. During the process, the port c of the composite block became redundant and thus was removed.

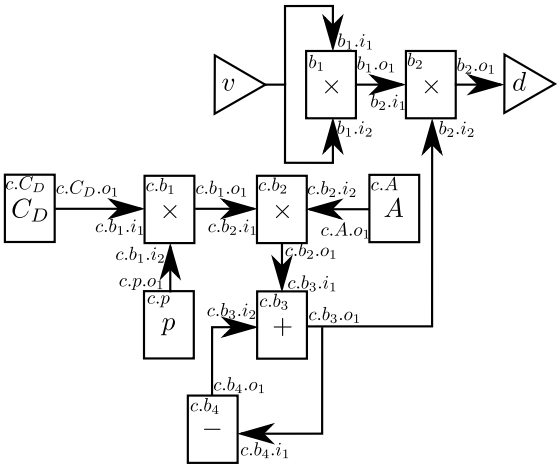


Fig. 5. Flattened version of algebraic CBD depicted in Fig. 3.

The meaning of a flattened CBD is the same as the original CBD. Every block/port has a unique identifier and every connection is between ports. The of CBDs can be thus explained assuming flattened CBDs only. This greatly simplifies the exposition.

Translational Semantics

Given a flattened algebraic CBD, the equations that it represents can be written down using the rules shown in Table 2.

Table 2. Translational semantics of a flattened algebraic CBD.

1. Assign a unique mathematical variable to the identifier of each port in the CBD.
2. Let (p, q) denote a connection from port id p to port id q, and let $\text{var}(p)$ and $\text{var}(q)$ denote the mathematical variables corresponding to p and q, according to the assignment made in Rule 1. Then, the equation associated with the connection (p, q) is $\text{var}(p) = \text{var}(q)$.
3. For each atomic block, let the sequence $p_1, p_2, \ldots$ denote the list of ids of its inputs ports, and let q denote id of the output port:
   (a) If the block is a constant with value c, then it has no input ports and the resulting equation is $\text{var}(q) = c$;
   (b) If it is a summation, then the resulting equation is $\text{var}(q) = \text{var}(p_1) + \text{var}(p_2)$;
   (c) If it is a product, then the resulting equation is $\text{var}(q) = \text{var}(p_1) \times \text{var}(p_2)$;
   (d) If it is a negation, then the resulting equation is $\text{var}(q) = -\text{var}(p_1)$;
   (e) If it is an inversion, the resulting equation is $\text{var}(q) = \frac{1}{\text{var}(p_1)}$;
   (f) If it is a raise-to-power, the resulting equation is $\text{var}(q) = \text{var}(p_1)^{\text{var}(p_2)}$;
   (g) If it is a root, the resulting equation is $\text{var}(q) = \text{var}(p_1)^{\frac{1}{\text{var}(p_2)}}$;

The system of equations that results from applying the following rules to each port, block and connection of an algebraic CBD can then be solved for the unknowns to get the values associated with each port in the CBD.

As an example, the flattened algebraic CBD depicted in Fig. 5 is translated into the following set of algebraic equations:

$$
\begin{aligned}
\text{var}(v) &= \text{var}(b_1.i_1) \\
\text{var}(v) &= \text{var}(b_1.i_2) \\
\text{var}(b_1.o_1) &= \text{var}(b_2.i_1) \\
\text{var}(b_2.o_1) &= \text{var}(d) \\
\text{var}(c.b_3.o_1) &= \text{var}(b_2.i_2) \\
\text{var}(c.C_d.o_1) &= \text{var}(c.b_1.i_1) \\
\text{var}(c.b_1.o_1) &= \text{var}(c.b_2.i_1) \\
\text{var}(c.p.o_1) &= \text{var}(c.b_1.i_2) \\
\text{var}(c.A.o_1) &= \text{var}(c.b_2.i_2) \\
\text{var}(c.b_2.o_1) &= \text{var}(c.b_3.i_1) \\
\text{var}(c.b_3.o_1) &= \text{var}(c.b_4.i_1) \\
\text{var}(c.b_4.o_1) &= \text{var}(c.b_3.i_2) \\
\text{var}(c.C_d.o_1) &= C_d \\
\text{var}(b_1.o_1) &= \text{var}(b_1.i_1) \times \text{var}(b_1.i_2) \\
\text{var}(b_2.o_1) &= \text{var}(b_2.i_1) \times \text{var}(b_2.i_2) \\
\text{var}(c.p.o_1) &= p \\
\text{var}(c.b_1.o_1) &= \text{var}(c.b_1.i_1) \times \text{var}(c.b_1.i_2) \\
\text{var}(c.A.o_1) &= A \\
\text{var}(c.b_2.o_1) &= \text{var}(c.b_2.i_1) \times \text{var}(c.b_2.i_2) \\
\text{var}(c.b_3.o_1) &= \text{var}(c.b_3.i_1) + \text{var}(c.b_3.i_2) \\
\text{var}(c.b_4.o_1) &= -\text{var}(c.b_4.i_1)
\end{aligned}
\tag{5}
$$

The system in Eq. (5) can be simplified to a quadratic drag force $\text{var}(b_2.o_1) = \text{var}(v)^2 \times \frac{1}{2} \times C_d \times p \times A$, where p is the air density, C_D the drag coefficient, and A the cross sectional area of the car. Obviously, the value of the input port v has to be known in order to solve for the value of the output port $b_2.o_1$.

Any system of algebraic equations that uses operations supported by the atomic blocks of algebraic CBDs can be represented as an algebraic CBD. For example, the CBD in Fig. 4 can be drawn directly from the equation

$$c = C_D \times p \times A - c \tag{6}$$

where c is the output, and C_D, A, p constants.

Operational Semantics

Instead of deferring the responsibility of computing the values associated with each port, to an equation solver, it is possible to do so directly. Two such algorithms are presented.

Algorithm 1 presents the dataflow version of the operational semantics. A list of atomic blocks to be computed is revisited iteratively until no blocks remain. A block can be computed only after all the blocks it depends on have been computed. The algorithm terminates after $\mathcal{O}((\#\text{atomic blocks in } D)^2)$ iterations.

The inefficiency of this algorithm lies in not taking advantage of the dependencies between blocks to come up with an optimal execution order for blocks. An improved algorithm will be presented later, after formalizing the dependency information between blocks.

Dependency Graph

The advantage of Algorithm 1 is its simplicity. It represents the execution model of the dataflow paradigm and, provided that no algebraic loops exist, it finds the values associated with every port of a flattened CBD.

An algebraic loop arises when a block depends indirectly on itself. It is thus natural to think of the CBD in terms of a dependency graph and identify the cycles thereof. In the CBD of Fig. 5, blocks $c.b_4$ and $c.b_3$ are part of one algebraic loop. Algebraic loops also happen in algebraic systems of equations. For example, in Eq. (6), the c variable depends on itself.

Both these loops where introduced artificially for the purposes of illustration. They can easily be removed by reformulating the mathematical expression that the CBD represents. However, in general, not all algebraic loops can be removed by this method and a way to detect them is required.

Given a flattened CBD, its corresponding dependency graph can be created applying the rules in Table 3. For example, Fig. 6 shows the dependency graph of the flattened CBD shown in Fig. 5.

Table 3. Rules for constructing the dependency graph.

1. For each block identified by b, create a unique node v. Let node(b) denote the corresponding node.
2. For each connection (p, q) from port id p to port id q, let b_p and b_q denote the block ids associated with ports p and q, respectively. If p or q have no associated blocks, then ignore this connection and proceed to the next one. Create a directed edge $(\text{node}(b_q), \text{node}(b_p))$ in the dependency graph, to mark that fact that b_q depends on b_p.

Solving Algebraic Loops

The dependency graph makes the detection of algebraic loops a simple matter of detecting the strong components in the graph. Formally, a strong component $S =$

Algorithm 1 Data-flow algorithm to evaluate an Algebraic CBD D.

function EvalAlgebraicCBD(D, $v_1, \ldots, v_n$)
Let val(p) be the computed value associated with port identified by p.
Let $i_1, \ldots, i_n$ be the ids of the input ports associated with the CBD D.
Let $o_1, \ldots, o_m$ be the ids of the output ports associated with the CBD D.
Then, val(i_1) := $v_1, \ldots,$ val(i_n) := v_n are the values associated with each input ports of D.
Let B denote the set of atomic blocks of D not yet computed.
Initially, B := all atomic blocks in D.
while $B \neq \{\}$ **do**
for $b_i \in$ B **do**
Let p denote the single output port of b_i.
Let $P = \{p_1, p_2, \ldots\}$ denote the inputs ports of b_i.
Let $Q = \{q_1, q_2, \ldots\}$ denote the output ports connected to each input port $p_j \in P$, respectively.
Let $\mathcal{B}$ = block(q_1) $\cup$ block(q_2) $\cup \ldots$ be the set of blocks that b_i depends on, where block(q_j) is the block associated with port q_j or the empty set, if no such block exists.
if $\mathcal{B} \cap B = \{\}$ **then**
Remark: val(q_1), val(q_2), ... have been computed before.
val(p) :=ComputeBlock(b_i, val(q_1), val(q_2), ...)
Let $\mathcal{P} = \{\rho_1, \rho_2, \ldots\}$ be the set of ports to which port p connects to.
val(ρ_j) := val(p), for $\rho_1 \in \mathcal{P}$
$B := B \setminus \{b_i\}$
end if
end for
end while
$P = \{\}$
return val(o_1), ..., val(o_m)
end function
function ComputeBlock(b, val(q_1), val(q_2), ...)
if b is a summation block **then**
return val(q_1) + val(q_2)
end if
if b is a Constant Block with value v **then**
return v
end if
...
end function

$\{n_1, n_2, \ldots\}$ of a graph G is a set of nodes where, between every $n_i, n_j \in S$, there are two different paths: $p_1 : n_i \overset{*}{\Rightarrow} n_j$ and $p_2 : n_j \overset{*}{\Rightarrow} n_i$. This implies that every node in a strong component is either the only node in that strong component, or depends on itself, through some other node, also in the same strong component. Fig. 6 illustrates the strong components of the dependency graph. As expected, the blocks $c.b_4$ and $c.b_3$ are part of the same strong component.

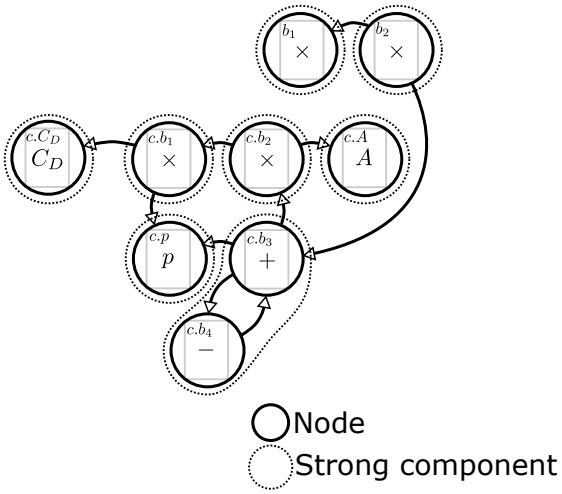


Fig. 6. Dependency graph and strong components.

Tarjan's algorithm [11] accepts a graph and outputs a sorted list of strong components. The sort order of the strong components is a topological order according to the dependencies between strong components. For the example in Fig. 6, one possible topological order is:

$$\{c.C_D\}, \{c.p\}, \{c.A\}, \{c.b1\}, \{c.b2\}, \{c.b_3, c.b_4\}, \{b_1\}, \{b_2\}$$

If no algebraic loops exist in the flattened graph, then the sorted list of strong components returned by the algorithm is just the topological sort of the nodes in dependency graph. In this list, a singleton strong component always appears after the nodes it depends on.

In the case where algebraic loops exist, all nodes belonging to the same algebraic loop will be in the same strong component. Regarding the sort order, non-singleton strong components appear after all components on which it depends on. A strong component depends on other if any one of its comprising nodes depends on at least one of the other's nodes.

These two facts about the sorted strong component list, given by Tarjan's algorithm [11], provide a basis for an improved algebraic CBD operational semantics, that not only can compute the values associated with all output ports much faster than Algorithm 1, but also detects algebraic loops. Algorithm 2 summarizes the steps for computing the values of all ports of a flattened CBD, under the improved algorithm.

The SolveLoop function computes the values of all unknown ports (whose value is unknown) associated with the blocks in the loop. A input port is unknown when an unknown output port is connected to it. An output port in unknown when the block it is associated with belongs to the strong component.

Algorithm 2 Evaluation an Algebraic CBD D with support for algebraic loops.

function EvalAlgebraicCBD(D, $v_1, \ldots, v_n$)
Let $\text{val}(p)$ be the computed value associated with port identified by p.
Let $i_1, \ldots, i_n$ be the ids of the input ports associated with the CBD D.
Let $o_1, \ldots, o_m$ be the ids of the output ports associated with the CBD D.
Then, $\text{val}(i_1) := v_1, \ldots, \text{val}(i_n) := v_n$ are the values associated with each input ports of D.
Let G denote the dependency graph induced by D.
Let $SC = (S_1, S_2, \ldots)$ denote the sorted list of strong components obtained with Tarjan's algorithm.
for $S_i \in SC$ **do**
if $S_i = \{n\}$ **then**
Let b denote the id of the unique block such that $\text{node}(b) = n$.
Let p denote the id of the output port associated with b.
Let $\{q_1, q_2, \ldots\}$ denote the ids of the input ports of b.
Remark: $\text{val}(q_1), \text{val}(q_2), \ldots$ have been computed.
$\text{val}(p) :=$ComputeBlock$(b, \text{val}(q_1), \text{val}(q_2), \ldots)$
Let $\mathcal{P} = \{\rho_1, \rho_2, \ldots\}$ be the ports that port p connects to.
$\text{val}(\rho_j) := \text{val}(p)$, for $p_j \in \mathcal{P}$
else if $S_i = \{n_1, n_2, \ldots\}$ **then**
Let $b_1, b_2, \ldots$ be the unique blocks such that $\text{node}(b_1) = n_1, \text{node}(b_2) = n_2, \ldots$
Let $p_1, p_2, \ldots$ denote the ids of the outputs ports of $b_1, b_2, \ldots$ respectively.
Let $Q_1, Q_2, \ldots$ denote the sets of ids of the inputs ports of $b_1, b_2, \ldots$ respectively, where $Q_i = \left\{q_1^{(i)}, q_2^{(i)}, \ldots\right\}$.
For each Q_i there might be input ports whose value is unknown, because these are connected to unknown output ports. Let $\bar{Q}_i = \left\{\bar{q}_1^{(i)}, \bar{q}_2^{(i)}, \ldots\right\} \subseteq Q_i$ denote the set of input ports whose value is known.
$(\text{val}(p_1), \text{val}(p_2), \ldots) :=$SolveLoop$(b, \text{val}(\bar{q}_1^{(1)}), \text{val}(\bar{q}_2^{(1)}), \ldots, \text{val}(\bar{q}_1^{(2)}), \ldots)$
for $p_i \in p_1, p_2, \ldots$ **do**
Let $\mathcal{P}_i = \left\{\rho_1^{(i)}, \rho_2^{(i)}, \ldots\right\}$ be the ports that port p_i connects to.
$\text{val}(\rho_j^{(i)}) := \text{val}(p_i)$, for $\rho_j^{(i)} \in \mathcal{P}_i$
end for
end if
end for
return $\text{val}(o_1), \ldots, \text{val}(o_m)$
end function

Equivalently, an input port is known when a known output port is connected to it. A known output port is associated with a block that does not belong to the strong component.

Essentially, solving an algebraic loop amounts to computing the solution of a matrix equation of the form $X = F(X, U)$:

$$\underbrace{\begin{bmatrix} \mathrm{val}(p_1) \\ \mathrm{val}(p_2) \\ \dots \end{bmatrix}}_{X} = \underbrace{\begin{bmatrix} F_1(\mathrm{val}(p_1), \mathrm{val}(p_2), \dots, \mathrm{val}(\bar{q}_1^{(1)}), \mathrm{val}(\bar{q}_2^{(1)}), \dots, \mathrm{val}(\bar{q}_1^{(2)}), \dots) \\ F_2(\mathrm{val}(p_1), \mathrm{val}(p_2), \dots, \mathrm{val}(\bar{q}_1^{(1)}), \mathrm{val}(\bar{q}_2^{(1)}), \dots, \mathrm{val}(\bar{q}_1^{(2)}), \dots) \\ \dots \end{bmatrix}}_{F(X,U)} \tag{7}$$

Where $X = [\mathrm{val}(p_1), \mathrm{val}(p_2), \dots]^T$ denotes the unknown values of the output ports of the strong component, and $U = \left[\mathrm{val}(\bar{q}_1^{(1)}), \mathrm{val}(\bar{q}_2^{(1)}), \dots, \mathrm{val}(\bar{q}_1^{(2)}), \dots\right]^T$ denote the known values of the input ports.

In Eq. (7), the unknown input ports are not considered because these depends directly, by algebraic equality, on the output ports connected to them. So finding the values of the unknown output ports is enough to be able to find the values of all unknown ports of the strong component.

The definition of F depends on the atomic blocks that belong to the strong component. If F is linear, then the above equation can be written in the form $AX = BU$ and solved with any technique suitable to solve linear systems of equations (Gaussian Elimination, Gauss-Seidel iteration, Jacobi-iteration, Gradient descent, etc...). Matrices A and B depend on the blocks in the strong component, and the product BU is known.

If F is non-linear, successive substitution techniques (e.g., Jacobi or Gauss Seidel), or derivative based methods (e.g., Newton–Raphson or Wegstein method) can be used in an attempt to find X. Caution has to be taken when non-linear loops are solved, as they might not have a solution, or a unique solution. The iterative methods require initial guess values to be provided for X, and depending on those initial guesses, different solutions might be attained. Both the initial guesses, and the solutions attained have to be physically meaningful, as the equations often represent the characteristics of physical systems (e.g., drag forces, concentrations, etc...).

For the algebraic loop containing bocks $c.b_4$ and $c.b_3$ in Fig. 6, the resulting linear system of equations and its analytical solution is:

$$\begin{cases} \text{val}(c.b_3.o_1) = \text{val}(c.b_3.i_1) + \text{val}(c.b_3.i_2) \\ \text{val}(c.b_3.i_2) = \text{val}(c.b_4.o_1) \\ \text{val}(c.b_4.o_1) = -\text{val}(c.b_4.i_1) \\ \text{val}(c.b_4.i_1) = \text{val}(c.b_3.o_1) \end{cases} \leftrightarrow$$

$$\begin{cases} \text{val}(c.b_3.o_1) - \text{val}(c.b_4.o_1) = \text{val}(c.b_3.i_1) \\ \text{val}(c.b_3.o_1) + \text{val}(c.b_4.o_1) = 0 \end{cases} \leftrightarrow$$

$$\underbrace{\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} \text{val}(c.b_3.o_1) \\ \text{val}(c.b_4.o_1) \end{bmatrix}}_{X} = \underbrace{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{B} \underbrace{\begin{bmatrix} \text{val}(c.b_3.i_1) \end{bmatrix}}_{U} \leftrightarrow$$

$$\begin{bmatrix} \text{val}(c.b_3.o_1) \\ \text{val}(c.b_4.o_1) \end{bmatrix} = \begin{bmatrix} \frac{1}{2}\text{val}(c.b_3.i_1) \\ -\frac{1}{2}\text{val}(c.b_3.i_1) \end{bmatrix}$$

3.3 Summary

In this section, the syntax and semantics of Algebraic CBDs were described. A flattening process that pre-processes any CBD into a canonical form was presented. This process will be used in the following sections to simplify the definition of Discrete time and Continuous time CBDs.

For the semantics of algebraic CBDs, two well known approaches were given: translational and operational. For a given Algebraic CBD, both of these are approximately equivalent, i.e., they give the same approximate values for the same ports of the CBD. This equivalence can be summarized by the commuting diagram in Fig. 7. The solutions are only approximately equal because, in the presence of algebraic loops, these may have to be solved iteratively to get an approximate solution.

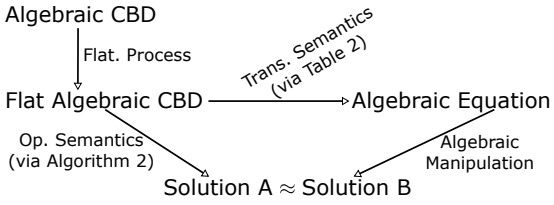


Fig. 7. Algebraic CBDs semantic equivalence (approximate).

Finally, it was mentioned that any algebraic system of equations can be written as an algebraic CBD.

In the next section, we expand the available atomic blocks to introduce the notion of evolving state via discrete jumps in time.

4 Discrete-time CBDs

In this section, the Discrete time CBDs are presented. Syntactically, the only difference to the Algebraic CBDs, is that the Discrete time CBDs allow the modeler to use not only algebraic blocks, but also a step delay block. Because the Delay block has a state, which gets updated whenever the block is computed, the other blocks in a Discrete time CBD no longer have static outputs (as in the algebraic CBDs case), but instead change whenever they are computed. Discrete time CBDs share many similarities with discrete time dynamical systems, presented in Section 2.3.

4.1 Syntax

The step delay block has two inputs i_1, i_c and one output o_1. It is represented with a $\mathcal{D}$ symbol, as highlighted in the discrete time CBD of Fig. 8. The input port i_c is called the initial condition and is distinguished by its subscript.

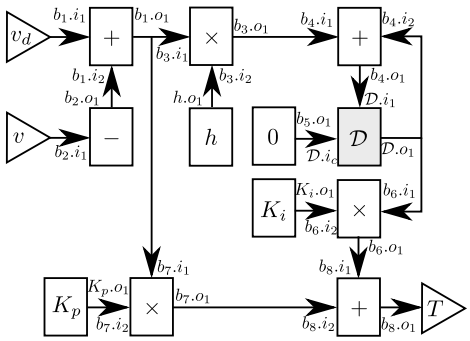


Fig. 8. Discrete-time CBD of the cruise controller with an highlighted Delay block $\mathcal{D}$. Equivalent to Eq. (4).

4.2 Semantics

The fact that the output of the delay block changes whenever it is computed means that the output of any other block that depends on the delay will also be dynamic. To formalize the multiple different values that any single port can assume, the notion of step is necessary. A step is a natural index that allows the distinction between the different outputs of each block. It is no different than the index used in difference equations, presented in Section 2.3.

Translational

The output of the delay block is defined in terms of the input provided at

the previous step. This is the essence of difference equations, where the current values are calculated from the previous ones. It is then natural that the meaning of a discrete time CBD is a set of difference equations.

Similarly to the algebraic CBD case, the flattening process ensures that only atomic blocks remain in the discrete time CBD. Given a flattened discrete time CBD, the difference equations that it represents can be written following the rules specified in Table 4.

Table 4. Translational semantics of a flattened discrete time CBD.

1. Assign a unique mathematical variable to the identifier of each port in the CBD.
2. Let (p, q) denote a connection from port identified by p to port identified by q, and let $\text{var}(p)$ and $\text{var}(q)$ denote the mathematical variables corresponding to p and q, following the assignment made in Rule 1. Then, the equation associated with the connection (p, q) is
   $\text{var}(p)^{[s+1]} = \text{var}(q)^{[s+1]}$.
3. Let $p_1, p_2, \ldots$ denote the list of ids of inputs ports of an atomic block, and let q denote id of its single output port:
   (a) If the block is a delay block, then the resulting equations are
   $\text{var}(q)^{[s+1]} = \text{var}(p_1)^{[s]}$ and
   $\text{var}(q)^{[0]} = \text{var}(p_c)^{[0]}$;
   (b) If the block is a constant block with value c, then the resulting equation is
   $\text{var}(q)^{[s+1]} = c$;
   (c) If the block is a summation block, then the resulting equation is
   $\text{var}(q)^{[s+1]} = \text{var}(p_1)^{[s+1]} + \text{var}(p_2)^{[s+1]}$;
   (d) If the block is a product block, then the resulting equation is
   $\text{var}(q)^{[s+1]} = \text{var}(p_1)^{[s+1]} \times \text{var}(p_2)^{[s+1]}$;
   (e) If the block is a negation block, then the resulting equation is
   $\text{var}(q)^{[s+1]} = -\text{var}(p_1)^{[s+1]}$;
   (f) If the block is an inversion block, then the resulting equation is
   $\text{var}(q)^{[s+1]} = \frac{1}{\text{var}(p_1)^{[s+1]}}$;
   (g) If the block is a raise-to-power block, then the resulting equation is
   $\text{var}(q)^{[s+1]} = \left(\text{var}(p_1)^{[s+1]}\right)^{\text{var}(p_2)^{[s+1]}}$;
   (h) If the block is a root block, then the resulting equation is
   $\text{var}(q)^{[s+1]} = \left(\text{var}(p_1)^{[s+1]}\right)^{\frac{1}{\text{var}(p_2)^{[s+1]}}}$;

The result is a set of difference equations, along with initial conditions (see Rule 3(a) of Table 4), that can be solved, either to obtain a closed-form solu-

tion, or simulated, by an independent solver. As an example, the discrete time CBD represented in Fig. 8 corresponds to, after simplification and renaming of variables, the software controller of Eq. (4).

Conversely, any difference equation written in the form of Eq. (3) can be represented as a Discrete time CBD. This is illustrated in Fig. 9.

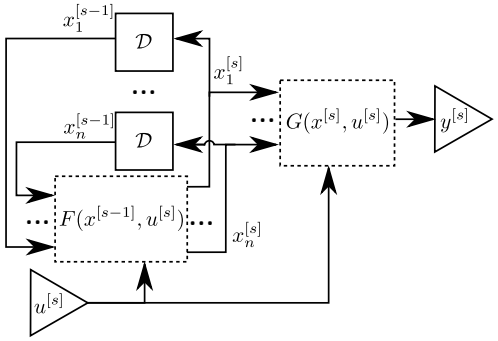


Fig. 9. Difference equation (written in the form of Eq. (3)) can be represented in a Discrete time CBD. The i-th component of the vector x is represented as x_i.

Operational

When compared to the algebraic CBDs operational semantics, in Algorithm 2, any algorithm that simulates discrete time CBDs has to compute not single values for variables, but discrete signals. A discrete signal is an ordered list of values, indexed by the step.

The operational meaning of the discrete time CBDs is thus the computation of the discrete time signal associated with each port. That can be done by fixing the step at 0, then computing all values in the CBD, as if it were an algebraic CBD. Then, step is incremented to 1, and the evaluation of all values is repeated, and so on. The fact that, within the same step, the discrete time CBD is evaluated as if it were an algebraic CBD, allows us to reuse the EvalAlgebraicCBD function, defined in Algorithm 2, with some minor changes:

- A parameter s is added to the ComputeBlock function, denoting the current step.
- All values are indexed by the current step. For example, the instruction
  $\text{val}(p) :=$ComputeBlock$(b, \text{val}(q_1), \text{val}(q_2), \ldots)$
  becomes
  $\text{val}(p)^{(s)} :=$ComputeBlock$(b, \text{val}(q_1)^{(s)}, \text{val}(q_2)^{(s)}, \ldots, s)$;

Algorithm 3 summarizes the operational semantics of discrete time CBDs. The definition of the ComputeBlock function is included, to specify the computation of the delay block. The computations of the remaining atomic blocks are trivial.

Algorithm 3 Operational Semantics of an Discrete time CBD D.

function EvalDiscreteTimeCBD(D, $v_1, \ldots, v_n$, N)
  Let val(p) be the computed value associated with port identified by p.
  Let $i_1, \ldots, i_n$ be the ids of the input ports associated with the CBD D.
  Let $o_1, \ldots, o_m$ be the ids of the output ports associated with the CBD D.
  Then, val(i_1) := $v_1, \ldots,$ val(i_n) = v_n are the values associated with each input ports of D.
  $s := 0$
  while $n \leq N$ **do**
    EvalAlgebraicCBD(D, $v_1^{(s)}, \ldots, v_n^{(s)}, s$)
    $s := s + 1$
  end while
  return val(o_1), ..., val(o_m)
end function
function ComputeBlock(b, val(q_1), ..., s)
  if b is a delay block **then**
    if $s = 0$ **then**
      return val(q_c)$^{(0)}$, where q_c is the id of the initial condition port.
    else
      return val(q_c)$^{(s-1)}$.
    end if
  end if
  if b is a summation block **then**
    return val(q_1)$^{(s)}$ + val(q_2)$^{(s)}$
  end if
  ...
end function

Algebraic Loops

If there are algebraic loops in the discrete time CBD, they are handled in the same as way in the algebraic CBDs (or they wouldn't be called algebraic loops...).

A note has to be made, however, about the dependencies of the delay block. At the first step ($s = 0$), the output of the delay block is equal to the input associated with its initial condition port (val(q_c)$^{(0)}$ in Algorithm 3). At any other step $s > 0$, the output is computed from the *previous* step $s - 1$.

This means that, except for $s = 0$, the delay block has no algebraic dependencies. And at $s = 0$ it depends on whatever block is connected to its initial condition port.

As a result, Rule 2 of Table 3 has to be adapted specifically for the Delay block and the current step being computed.

4.3 Summary

Following the same structure as Section 3, this section presented the syntax and semantics, both translational and operational, of discrete time CBDs.

A discrete time CBD can be translated into a system of difference equations by applying the rules of Table 4. Analogously, a difference equation written in the form of Eq. (3) can be translated into a discrete time CBD as shown in Fig. 9.

Difference equations, or discrete time CBDs, as exemplified in Section 2.3, can be used for the description of software components whose state evolves discretely in time.

The operational semantics of discrete time CBDs reuse the operational semantics of algebraic CBDs, with minor modifications.

As in the algebraic CBDs case, any solution obtained via the translational approach is approximately the same as the one obtained via Algorithm 3. This is summarized in the commuting diagram of Fig. 10.

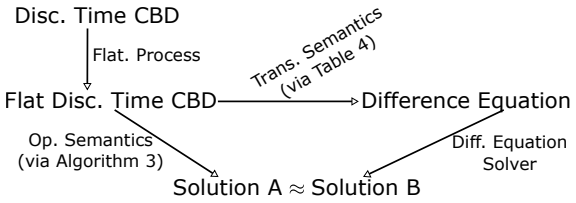


Fig. 10. Discrete time CBDs approximate semantic equivalence.

5 Continuous-time CBDs

The meaning of algebraic CBDs is a set of algebraic equations (see Fig. 7) and the meaning of Discrete time CBDs is a set of difference equations (see Fig. 10).

As shown in Section 2.2, differential equations are ideal to represent physical systems, whose state evolves continuously in time. By the end of this section, it will be clear that continuous time CBDs too, are suited to model these systems.

5.1 Syntax

Syntactically, continuous time CBDs include the standard algebraic blocks, a derivative, and an integral block. The Delay block is not included.

The derivative and integral blocks have two inputs i_1, i_c, and one output o_1. The input subscripted by c denotes the initial condition. Both blocks will be denoted by the appropriate mathematical symbol: $\frac{d}{dt}$ and $\int$.

Fig. 11 shows a continuous time CBD example, with the Drag block specified in Fig. 12.

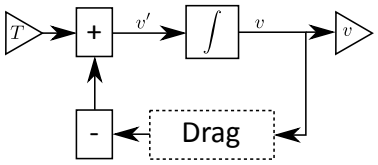


Fig. 11. Continuous time CBD of the car in the cruise control system of Fig. 2.

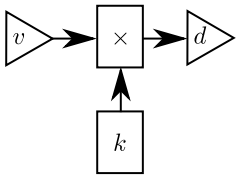


Fig. 12. Continuous time CBD of the drag.

5.2 Semantics

Translational – To Differential Equations

The meaning of a flattened continuous time CBD is a system of Ordinary Differential Equations (ODEs). Table 5 shows the rules that build such system. The meaning of Fig. 11 is represented, after simplification and renaming the variables, in Eq. (2).

Furthermore, any ODE written in the form of Eq. (1) can be translated to a Continuous time CBDs as illustrated in Fig. 13.

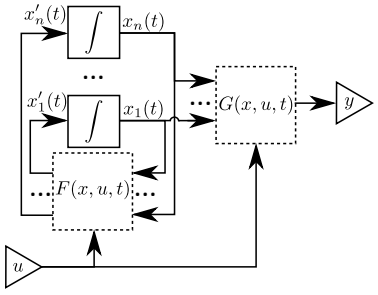


Fig. 13. First order ODE, written in the form of Eq. (1), can represented as a continuous time CBD. The i-th component of the vector x is represented as x_i.

Table 5. Translational semantics of a flattened continuous time CBD.

1. Assign a unique mathematical variable to the identifier of each port in the CBD.
2. Let (p, q) denote a connection from port identified by p to port identified by q, and let $\text{var}(p)$ and $\text{var}(q)$ denote the mathematical variables corresponding to p and q, following the assignment made in Rule 1. Then, the equation associated with the connection (p, q) is $\text{var}(p)(t) = \text{var}(q)(t)$.
3. Let $p_1, p_2, \ldots$ denote the list of ids of inputs ports of an atomic block, and let q denote id of its single output port:
   (a) If the block is an Integral block, then the resulting equation is $\text{var}(q)(t) = \int_0^t \text{var}(p_1)(\tau)d\tau + \text{var}(p_c)(0)$;
   (b) If the block is a Derivative block, then the resulting equations are $\text{var}(q)(t) = \text{var}(p_1)'(t)$ and $\text{var}(q)(0) = \text{var}(p_c)(0)$;
   (c) If the block is a summation block, then the resulting equation is $\text{var}(q)(t) = \text{var}(p_1)(t) + \text{var}(p_2)(t)$;
   (d) ...

Basics of ODE Discretization

In many ODEs arising in science and engineering, and this includes, by the translational semantics, continuous time CBDs, a closed-form solution cannot be found. One of the possible ways to get insight into the solution is via simulation. Nowadays, most simulations are performed in a digital computer, where the state of software can only evolve discretely through time. In this context, solutions to ODE's obtained via simulation can only be approximate.

Contrarily to differential equations, the solution to difference equations (recall Section 2.3) can be obtained exactly in a digital computer, provided that roundoff errors due to the floating point representation are neglected.

In the following paragraphs, we show how to translate ODEs into difference equations, whose solution, obtained via simulation, approximate the solution of the ODEs. The reader is referred to [6, 4] for a more detailed exposition on numerical approximation methods. The method explained is then used in Section 5.2 as the basis to describe how a continuous time CBD is translated into discrete time CBDs, which approximate the solution to the original.

Forward Euler Method

Consider a first order ODE without input:

$$\begin{aligned} x'(t) &= F(x, t) \\ x(0) &= x_0 \end{aligned} \tag{8}$$

where $x(t) = [x_1(t), \ldots, x_n(t)]^T$ is the state vector, $F(x, t) = [F_1(x, t), \ldots, F_n(x, t)]^T$ the state derivative function, and x_0 the initial value of x. Let $x_i(t)$ denote the

i-th state trajectory and $x_i'(t) = F_i(x,t)$ the i-th state derivative. Assuming that $x_i(t)$ and any of its derivatives are smooth, it can be approximated around any point t^* by the Taylor series:

$$x_i\left(t^* + \Delta t\right) = x_i\left(t^*\right) + x_i'\left(x\left(t^*\right), t^*\right)\Delta t + x_i^{(2)}\left(t^*\right)\frac{\Delta t^2}{2!} + \ldots \tag{9}$$

Using Taylor's theorem, it is possible to write the Taylor series expansion in the finite form of a polynomial and a residual in Lagrange form [4]:

$$x_i(t^* + \Delta t) = x_i(t^*) + x_i'(x(t^*), t^*)\Delta t + \ldots + x_i^{(n)}(t^*)\frac{\Delta t^n}{n!} + x_i^{(n+1)}(\xi(t^*))\frac{\Delta t^{n+1}}{(n+1)!}$$

where $\xi(t^*)$ is an unknown number between t^* and $t^* + \Delta t$. The residual term $x_i^{(n+1)}(\xi(t^*))\frac{\Delta t^{n+1}}{(n+1)!}$ denotes the truncation error. It cannot be computed directly but, since any of the derivatives of x_i are smooth in all points between t^* and $t^* + \Delta t$, there exists a maximum constant $K < \infty$, such that, for any t^* and any n,

$$\frac{x_i^{(n+1)}(\xi(t^*))}{(n+1)!} \leq K$$

An upper bound can then be written for the remainder term in the Big O notation:

$$\lim_{\Delta t \to 0} \frac{x_i^{(n+1)}(\xi(t^*))}{(n+1)!}\Delta t^{n+1} \leq K\Delta t^{n+1} = \mathcal{O}(\Delta t^{n+1})$$

Notice that the Big O notation $\mathcal{O}(\Delta t^{n+1})$ highlights the dominant term as $\Delta t \to 0$

Taylor theorem allows us to write the Taylor series taking into account the ODE of Eq. (8) and replacing the residual term by its order:

$$x_i\left(t^* + \Delta t\right) = x_i\left(t^*\right) + F_i\left(x\left(t^*\right), t^*\right)\Delta t + \mathcal{O}(\Delta t^2) \tag{10}$$

For small $\Delta t < 1$ we can neglect the $\mathcal{O}\left(\Delta t^2\right)$ term and approximate $x_i\left(t^* + \Delta t\right)$ by:

$$x_i\left(t^* + \Delta t\right) \approx x_i\left(t^*\right) + F_i\left(x\left(t^*\right), t^*\right)\Delta t \tag{11}$$

Going back to the vector case, this suggests that we can approximate the solution vector $x\left(t\right)$ by the following algorithm:

$$\begin{aligned} x(\Delta t) &:\approx x(0) + F(x(0), 0)\Delta t \\ x(\Delta t + \Delta t) &:\approx x(\Delta t) + F(x(\Delta t), \Delta t)\Delta t \\ x(2\Delta t + \Delta t) &:\approx x(2\Delta t) + F(x(2\Delta t), 2\Delta t)\Delta t \\ &\ldots \end{aligned} \tag{12}$$

Let $x^{[s]} = x(s\Delta t)$, we get the Forward Euler method to numerically approximate Eq. (8):

$$x^{[s+1]} = x^{[s]} + F(x^{[s]}, s\Delta t)\Delta t \tag{13}$$

Geometrically, the Forward Euler can be derived by fitting a line going from a known point $x^{[s]}$, along a known slope $F(x^{[s]}, s\Delta t)$. The slope is given because $F(x^{[s]}, s\Delta t)$ is the derivative of $x^{[s]}$. An alternative interpretation is that $x^{[s+1]}$ is the area, approximated by rectangles, under the curve F up to $(s+1)\Delta t$. Fig. 14 illustrates these views.

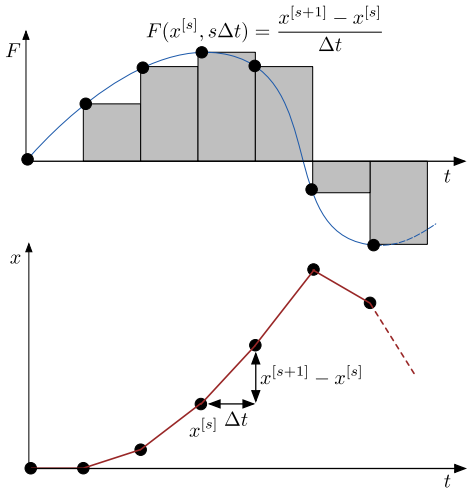


Fig. 14. Geometric interpretation of the Forward Euler method.

Newton's Difference Quotient

The Taylor series, introduced in Eq. (9) also works backward from any point, including the point $x_i\,(t^* + \Delta t)$:

$$x((t^* + \Delta t) - \Delta t) = x(t^* + \Delta t) - x'(x(t^* + \Delta t), t^*)\Delta t + \mathcal{O}(\Delta t^2) \tag{14}$$

Replacing the derivative by F, from Eq. (8), neglecting the residual term, and simplifying gives the Newton's Difference Quotient:

$$\frac{x(t^* + \Delta t) - x(t^*)}{\Delta t} \approx F(x(t^* + \Delta t)) \tag{15}$$

Rewritten as a difference equation:

$$\frac{x^{[s+1]} - x^{[s]}}{\Delta t} = F(x^{[s+1]})$$

Contrary to the Forward Euler, it is not possible to get an iterative algorithm immediately out of this method: the vector term $x^{[s+1]}$ depends on itself. This is an algebraic loop (recall Section 3.2). It requires that the matrix equation be solved for $x^{[s+1]}$. The presence of these loops distinguishes implicit (with loops) from explicit (without loops) methods. The important point is that, as shown in Section 3.2, these loops can be solved at each simulation step.

Translational – To Discrete-time CBDs

As explained in the previous section, differential equations are discretized to difference equations by means of numerical approximation techniques, which can then be easily simulated.

Since any continuous time CBD can be translated into an ODE (by Table 5), and since the meaning of a discrete time CBD is a system of difference equations, it is natural to wonder whether a discrete time CBD can be transformed directly into a discrete time CBD, which realizes the approximation. The only blocks that need to be approximated are the derivative and the integral. All the other blocks are algebraic. The integral block is left as an exercise.

Derivative Block Approximation

The derivative block outputs the derivative of its input u, at time t:

$$y(t) = u'(t)$$

except at time $t = 0$, where the output $y(0)$ is given by the input initial condition u_c, i.e., $y(0) = u_c(0)$.

Applying the Newton's Difference Quotient, from Eq. (15), yields:

$$\frac{u(t + \Delta t) - u(t)}{\Delta t} \approx y(t + \Delta t)$$

Solving for the output $y(t + \Delta t)$ and writing as a difference equation gives:

$$y^{[s]} :\approx \frac{u^{[s]} - u^{[s-1]}}{\Delta t} \tag{16}$$

Since the input is not differentiable at time $t = 0$, the initial condition of the derivative block is provided with an initial value $y(0) = u_c(0)$.

It is easy to build a discrete time CBD equivalent to Eq. (16) using a delay and algebraic blocks. The delay block will ensure the delayed signal of the input ($u^{[s-1]}$) can be obtained. However, at the initial step, $s = 0$, the delay block has to have an initial condition defined because the value $u^{[-1]}$, in Eq. (16), is unknown. Let u_{-1} denote this unknown value. u_{-1} cannot be equal to $y^{[0]}$. That does not satisfy the initial condition of the derivative, expressed as:

$$y^{[0]} \approx \frac{u^{[0]} - u_{-1}}{\Delta t} = u_c^{[0]}$$

To find out the initial condition of the delay, one can rearrange the above equation to get $u_{-1} = u^{[0]} - \Delta t \cdot u_c^{[0]}$, which defines the initial condition of the delay. Fig. 15 shows the transformation rule.

5.3 Summary

In this section, continuous time CBDs where introduced. These are a suitable formalism to model continuous time dynamical systems such as the car, in the cruise control example (recall Fig. 2).

The meaning of continuous time CBDs was given in two ways:

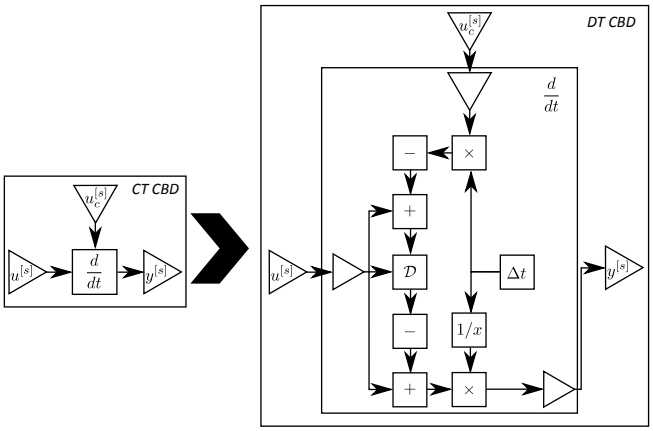


Fig. 15. Sample continuous time CBD with a Derivative block (on the left) and the corresponding discrete time CBD (on the right).

1. Translation to differential equations. These meaning of these, in turn, can either be obtained analytically, or approximated via difference equations. In addition, ODEs can be written as continuous time CBDs (see Fig. 13)
2. Translation to discrete time CBDs, which in turn can be either operationally simulated, or translated to difference equations. In this translation, integral/derivative blocks get replaced by composite blocks realizing the approximation (recall Fig. 15).

Fig. 16 summarizes the translational semantics approximate equivalence.

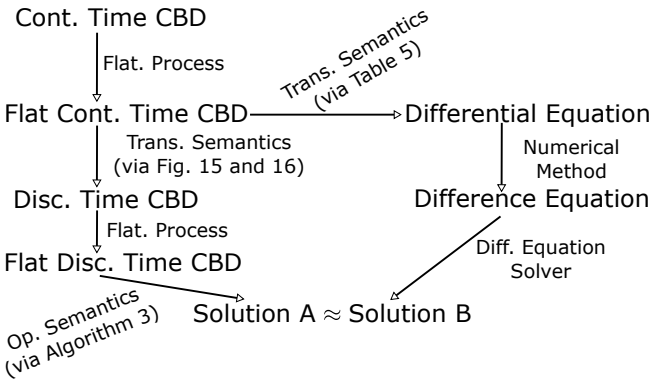


Fig. 16. Approximate equivalence between the two translational semantics approaches presented.

6 Advanced Concepts

Until now, we have introduced the minimal set of concepts that allow the reader to use and understand the semantics of CBDs. We skipped over a few details which, for that purpose, are not important, but in order to become a proficient user of CBDs, one has to understand some advanced simulation concepts.

For example, we took for granted how approximate the solution computed by the Forward Euler method is. Furthermore, we have not introduced the operational semantics of continuous time CBDs. Such algorithm can be easily devised for the approximations of the integral and derivative blocks already given – Forward Euler and Newton's Difference Quotient – with special attention to the fact that the approximation of the derivative may introduce algebraic loops in the CBD. However, if those approximation methods are used, the algorithm will be hardly useful: the translation to discrete time CBDs is equivalent and already deals with the algebraic loops problem for free (recall Fig. 16 and Section 4.2).

After being introduced to more advanced numerical methods (Section 6.2 and Section 6.3), the reader can devise smarter algorithms for the simulation of continuous time CBDs directly, while minimizing the error in the approximation (Section 6.1). We focus on numerical integration methods, that is, approximations of the integrator block, because these are the most commonly used when modeling physical systems (see Fig. 13).

Finally, we introduce an extension that is widely used in CBDs: logic blocks. These allow higher level reasoning to be used in CBDs, conveying more expressive power to the modeler, but introducing other interesting challenges when it comes to simulation.

6.1 Approximation Error

Recall the Forward Euler method in Eq. (11) and Eq. (13). To derive it, the term $\mathcal{O}(\Delta t^2)$ of the Taylor series was neglected. Recall Equations 9, 10 and 11.

The resulting method is an iterative approximation, as shown in Eq. (12). Let $x(t)$ denote the solution to Eq. (8) approximated with Forward Euler, and let $\hat{x}$ denote the real solution. The first term $x(0) = \hat{x}(0)$ is known from the initial condition of Eq. (8).

The second term – $x(\Delta t) = x(0) + F(x(0))\Delta t$ – deviates from the true solution $\hat{x}(\Delta t)$ by an order $\mathcal{O}(\Delta t^2)$, which is the residual term ignored in the Taylor series (recall Eq. (10) and Eq. (11)). Formally, that is,

$$\|\hat{x}(\Delta t) - x(\Delta t)\| = \left\|\hat{x}(0) + F(\hat{x}(0))\Delta t + \mathcal{O}(\Delta t^2) - x(0) - F(x(0))\right\| = \mathcal{O}(\Delta t^2)$$

The third term $x(2\Delta t)$ will deviate further from the true solution not only because of the residual – of order $\mathcal{O}(\Delta t^2)$– but also because $F(x(\Delta t), \Delta t)$ is evaluated with the approximated term $x(\Delta t)$ and most likely $F(x(\Delta t)) \neq F(\hat{x}(\Delta t))$

The iteration continues and it is easy to see that the error accumulates over the iterations. In order to analyze the accumulation of errors, it is best to distinguish two kinds of errors: the local truncation error, due to the ignored residual

term, and the derivative error, due to evaluating the derivative F at approximated points x. Both these errors contribute to the accumulation of error over time, that is, the global error.

The local truncation error denotes the deviation made by a single step of the numerical method, starting from accurate information, i.e., with no previously accumulated error, just like the first step of the iteration method in Eq. (12).

Let

$$\hat{x}((s+1)\cdot \Delta t) = \hat{x}(s\cdot \Delta t) + F\left(\hat{x}(s\cdot \Delta t)\right)\Delta t + \mathcal{O}\left(\Delta t^2\right) \tag{17}$$

denote the real solution expanded with the infinite Taylor series, and let

$$x((s+1)\cdot \Delta t) \approx \hat{x}(s\cdot \Delta t) + F\left(\hat{x}(s\cdot \Delta t)\right)\Delta t$$

denote the solution computed across one step of the Forward Euler method, starting from accurate information. The local truncation error is thus given as

$$\|\hat{x}((s+1)\cdot \Delta t) - x((s+1)\cdot \Delta t)\| = \mathcal{O}(\Delta t^2) \tag{18}$$

Studying the global error is more difficult as it depends on the derivative error, which, for a generic analysis, can be any function F. If any error in the parameter of F gets amplified, then the global error will grow faster. If it gets contracted, then the global error will grow in a slower fashion. To formalize, suppose that we know something about F:

$$\|F(\hat{x}(t)) - F(x(t))\| \leq K_f \|\hat{x}(t) - x(t)\|, \text{ for all } t \in \mathbb{R} \tag{19}$$

where $0 \leq K_f < \infty$ is a constant.

Then the error at the second step of the Forward Euler can be derived as follows:

$$\begin{aligned}
&\|\hat{x}(2\Delta t) - x(2\Delta t)\| \\
&= \left\|\hat{x}(\Delta t) + F(\hat{x}(\Delta t))\Delta t + \mathcal{O}(\Delta t^2) - (x(\Delta t) + F(x(\Delta t))\Delta t)\right\| \\
&= \left\|\hat{x}(\Delta t) - x(\Delta t) + (F(\hat{x}(\Delta t)) - F(x(\Delta t)))\,\Delta t + \mathcal{O}(\Delta t^2)\right\| \\
&\leq \|\hat{x}(\Delta t) - x(\Delta t)\| + \|F(\hat{x}(\Delta t)) - F(x(\Delta t))\|\,\Delta t + \mathcal{O}(\Delta t^2) \\
&\leq \|\hat{x}(\Delta t) - x(\Delta t)\| + K_f \|\hat{x}(\Delta t) - x(\Delta t)\|\,\Delta t + \mathcal{O}(\Delta t^2) \\
&= (2 + K_f \Delta t)\,\mathcal{O}(\Delta t^2) = \mathcal{O}(2\Delta t^2)
\end{aligned} \tag{20}$$

Notice that, as $\Delta t \to 0$, the big O definition implies that

$$(2 + K_f \Delta t)\,\mathcal{O}(\Delta t^2) = \mathcal{O}(2\Delta t^2 + K_f \Delta t^3) = \mathcal{O}(2\Delta t^2) \tag{21}$$

Similarly, for the third step:

$$\begin{aligned}
&\|\hat{x}(3\Delta t) - x(3\Delta t)\| \\
&= \left\|\hat{x}(2\Delta t) + F(\hat{x}(2\Delta t))\Delta t + \mathcal{O}(\Delta t^2) - (x(2\Delta t) + F(x(2\Delta t))\Delta t)\right\| \\
&= \left\|\hat{x}(2\Delta t) - x(2\Delta t) + (F(\hat{x}(2\Delta t)) - F(x(2\Delta t)))\,\Delta t + \mathcal{O}(\Delta t^2)\right\| \\
&\leq \|\hat{x}(2\Delta t) - x(2\Delta t)\| + \|F(\hat{x}(2\Delta t)) - F(x(2\Delta t))\|\,\Delta t + \mathcal{O}(\Delta t^2) \\
&\leq (1 + K_f \Delta t)\,\|\hat{x}(2\Delta t) - x(2\Delta t)\| + \mathcal{O}(\Delta t^2) \\
&= \mathcal{O}(3\Delta t^2)
\end{aligned} \tag{22}$$

After s steps, we get

$$\|\hat{x}(s\Delta t) - x(s\Delta t)\| = \mathcal{O}(s\Delta t^2)$$

To run the simulation up to time t_f the Forward Euler method performs $t_f/\Delta t$ steps, which gives

$$\|\hat{x}(t_f) - x(t_f)\| = \mathcal{O}(\frac{t_f}{\Delta t}\Delta t^2) = \mathcal{O}(\Delta t)$$

Which says that the global error will not grow worse than linear in the size of Δt, as $\Delta t \to 0$. For a more accurate expression of the global error of the Forward Euler method, see [4, 5].

An important conclusion is that, provided that function F is well behaved, i.e., obeys the condition in Eq. (19), the global error can be minimized by simply taking smaller Δt at each step of the simulation using the Forward Euler method. This is called convergence. A numerical method, to be of any use, should be convergent.

Since there is a limit to how small Δt can be made in a digital computer, a numerical method which has an higher order of accuracy than $\mathcal{O}(\Delta t)$, for example, $\mathcal{O}(\Delta t^2)$, will allow for larger steps to be taken. In the next section, we introduce other numerical methods.

6.2 Other Numerical Methods

Backward Euler Method

The Taylor series (Eq. (10)) also works backward from any point, including the point $(t^* + \Delta t)$, as was done in Eq. (14). Neglecting the residual term, we get:

$$x_i((t^* + \Delta t) - \Delta t) \approx x_i(t^* + \Delta t) - x_i'(x(t^* + \Delta t))\Delta t$$

After some simplifications we get a method that resembles the Forward Euler method:

$$x(t^* + \Delta t) \approx x_i(t^*) + F(x(t^* + \Delta t))\Delta t \tag{23}$$

This suggests the following set of equations to numerically approximate the solution:

$$\begin{aligned} x(\Delta t) &\approx x(0) + F(x(\Delta t), \Delta t)\,\Delta t \\ x(\Delta t + \Delta t) &\approx x(\Delta t) + F(x(\Delta t + \Delta t), \Delta t + \Delta t)\,\Delta t \\ x(2\Delta t + \Delta t) &\approx x(2\Delta t) + F(x(2\Delta t + \Delta t), 2\Delta t + \Delta t)\,\Delta t \\ &\dots \end{aligned}$$

Hence we have the backward Euler method:

$$x^{[s+1]} = x^{[s]} + F(x^{[s+1]})\Delta t \tag{24}$$

When compared to explicit methods, the backward Euler requires the solution to an algebraic loop so it will incur some extra computation at each simulation step. Furthermore, the global and local errors of the backward Euler are of the same order as the Forward Euler's. Their difference lies in the fact that the derivative used to make the estimation of $x^{[s+1]}$ is the closest to it. In the Forward Euler, the derivative is an *out-dated* one. This has benefits when dealing with stiff systems. See [5, 4, 7] for more details.

Fig. 17 shows the geometric interpretation of the backward Euler method. Note that F has a similar shape as that of the Forward Euler (in Fig. 14) but this is not necessarily the case as F depends on x, which is being approximated with a different method.

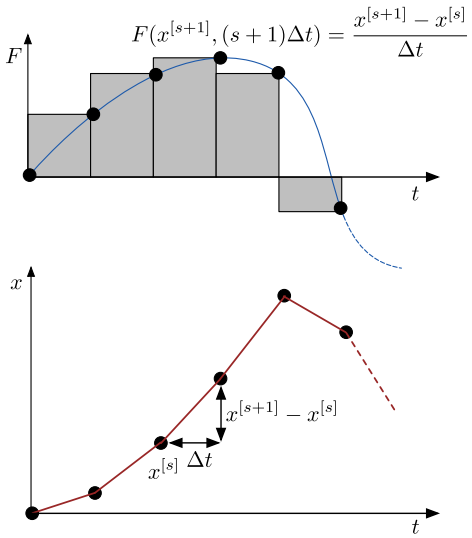


Fig. 17. Geometric Interpretation of the Backward Euler Algorithm

Second Order Taylor Method

Until now we have always neglected the term $\mathcal{O}(\Delta t^2)$ of the Taylor series. Let us see what happens when we neglect higher order terms. For example, the Taylor series, after neglecting the term $\mathcal{O}(\Delta t^3)$, becomes:

$$x(t^* + \Delta t) \approx x(t^*) + F(x(t^*), t^*)\Delta t + \frac{dF\left(x\left(t^*\right), t^*\right)}{dt}\frac{\Delta t^2}{2!}$$

The derivative $\frac{dF(x(t^*),t^*)}{dt}$ can be expanded with the chain rule [1]:

$$\frac{dF(x(t^*), t^*)}{dt} = \frac{\partial F(x(t^*), t^*)}{\partial x}F(x(t^*), t^*) + \frac{\partial F(x(t^*), t^*)}{\partial t}$$

The second order Taylor series method then becomes:

$$\begin{aligned} x^{[s+1]} = x^{[s]} &+ F(x^{[s]}, s \cdot \Delta t)\Delta t \\ &+ \left(\frac{\partial F(x^{[s]}, s \cdot \Delta t)}{\partial x}F(x^{[s]}, s \cdot \Delta t) + \frac{\partial F(x^{[s]}, s \cdot \Delta t)}{\partial t}\right)\frac{\Delta t^2}{2!} \end{aligned} \tag{25}$$

The local truncation error of this method is the neglected term $\mathcal{O}(\Delta t^3)$, better than the Euler methods. The disadvantage of this method is that it requires the calculation (symbolically or numerically) of the partial derivatives of F – a costly operation. The global error is in the order of $\mathcal{O}(\Delta t^2)$ for well behaved derivatives.

Higher order Taylor methods require even more derivative calculations, making them impractical. There are methods that offer that same global error order with far less computation at each step. We show one next.

Midpoint Method

The backward Euler method makes use of the most up-to-date derivative to estimate the solution at $t^* + \Delta t$ with the disadvantage that it requires more computation to solve the implicit equation. To avoid this, but still trying to be better than Forward Euler, we can try to estimate the derivative at halfway between t^* and $t^* + \Delta t$ and use that derivative to compute $x(t^* + \Delta t)$:

$$x(t^* + \Delta t) = x(t^*) + F(x(t^* + \frac{\Delta t}{2}))\Delta t$$

However, we do not know the value of $x(t^* + \frac{\Delta t}{2})$. We can use Taylor series again to get

$$x(t^* + \frac{\Delta t}{2}) = x(t^*) + F(x(t^*))\frac{\Delta t}{2}$$

Thus we arrive at the midpoint method:

$$x^{[s+1]} = x^{[s]} + F\left[x^{[s]} + F\left(x^{[s]}, s \cdot \Delta t\right)\frac{\Delta t}{2}, \left(s + \frac{1}{2}\right) \cdot \Delta t\right]\Delta t \tag{26}$$

[1] Notice that, to be general, we represent the derivative $F(x(t^*), t^*)$ as a function that depends directly on the time. If this is not the case, then $\frac{\partial F(x(t^*),t^*)}{\partial t} = 0$.

The midpoint method, Eq. (26), can be generalized to

$$x_C^{[s+1]} = x^{[s]} + \beta_{C1} \cdot \Delta t \cdot F^{[s]} + \beta_{C2} \cdot \Delta t \cdot F\left(x^{[s]} + \beta_p \cdot F^{[s]} \cdot \Delta t,\, (s + \alpha_p) \cdot \Delta t\right)$$

where $F^{[s]} = F\left(x^{[s]}, s \cdot \Delta t\right)$, $\beta_p = \alpha_p = \frac{1}{2}$, $\beta_{C1} = 0$, and $\beta_{C2} = 1$.

Expanding $F\left(x^{[s]} + \beta_p \cdot F^{[s]} \cdot \Delta t,\, (s + \alpha_p) \cdot \Delta t\right)$ with the multi-variate version of the Taylor series, we get:

$$\begin{aligned} &F\left(x^{[s]} + \beta_p \cdot F^{[s]} \cdot \Delta t,\, (s + \alpha_p) \cdot \Delta t\right) \\ &\quad \approx F^{[s]} + \beta_p \cdot \frac{\partial F^{[s]}}{\partial x} \cdot F^{[s]} \cdot \Delta t + \alpha_p \cdot \frac{\partial F^{[s]}}{\partial t} \cdot \Delta t \end{aligned}$$

Where the quadratic term was neglected. Plugging it into the previous equation gives:

$$\begin{aligned} x_C^{[s+1]} &= x^{[s]} + \beta_{C1} \cdot F^{[s]} \cdot \Delta t + \\ &\qquad \beta_{C2} \cdot \left[F^{[s]} + \beta_p \cdot \frac{\partial F^{[s]}}{\partial x} \cdot F^{[s]} \cdot \Delta t + \alpha_p \cdot \frac{\partial F^{[s]}}{\partial t} \cdot \Delta t\right] \cdot \Delta t \\ &= x^{[s]} + (\beta_{C1} + \beta_{C2})\, F^{[s]} \cdot \Delta t + \\ &\qquad \beta_{C2} \left[\beta_p \cdot \frac{\partial F^{[s]}}{\partial x} \cdot F^{[s]} + \alpha_p \cdot \frac{\partial F^{[s]}}{\partial t}\right] \cdot \Delta t^2 \end{aligned}$$

To find the local truncation error, let us find the Taylor series expansion of the true solution and then compare it to the previous equation. The true solution can be expanded as:

$$\hat{x}^{[s+1]} = \hat{x}^{[s]} + F^{[s]} \cdot \Delta t + \frac{1}{2} \frac{\partial F^{[s]}}{\partial x} \Delta t^2 + \mathcal{O}(\Delta t^3)$$

Applying the chain rule to the derivative yields:

$$\hat{x}^{[s+1]} = \hat{x}^{[s]} + F^{[s]} \cdot \Delta t + \frac{1}{2} \cdot \left[\frac{\partial F^{[s]}}{\partial x} F^{[s]} + \frac{\partial F^{[s]}}{\partial t}\right] \cdot \Delta t^2 + \mathcal{O}(\Delta t^3)$$

Comparing $\hat{x}^{[s+1]}$ with $x_C^{[s+1]}$, and assuming that these start from a true solution $\hat{x}^{[s]}$ gives:

$$\begin{aligned} &x_C^{[s+1]} = \hat{x}^{[s+1]} \leftrightarrow \\ &\hat{x}^{[s]} + (\beta_{C1} + \beta_{C2})\, F^{[s]} \cdot \Delta t + \beta_{C2} \left[\beta_p \cdot \frac{\partial F^{[s]}}{\partial x} \cdot F^{[s]} + \alpha_p \cdot \frac{\partial F^{[s]}}{\partial t}\right] \cdot \Delta t^2 \\ &\qquad = \hat{x}^{[s]} + F^{[s]} \cdot \Delta t + \frac{1}{2} \cdot \left[\frac{\partial F^{[s]}}{\partial x} F^{[s]} + \frac{\partial F^{[s]}}{\partial t}\right] \cdot \Delta t^2 + \mathcal{O}(\Delta t^3) \end{aligned}$$

When solved for the parameters, the above equation gives:

$$\begin{cases} \beta_{C1} + \beta_{C2} = 1 \\ 2\beta_{C2}\beta_p = 1 \\ 2\beta_{C2}\alpha_p = 1 \end{cases}$$

As long as the parameters $\beta_p, \alpha_p, \beta_{C1}, \beta_{C2}$ obey the above system of equations, the generic method will have a local truncation error of order $\mathcal{O}(\Delta t^3))$, without having to compute any derivative of F. This also shows that the midpoint method is but an element of a family of methods, all with different sets of parameters, called the two stage Runge Kutta methods.

By the same argument as the Forward Euler (in Section 6.1), we conclude that the global error of the two stage Runge Kutta method is of order $\mathcal{O}(\Delta t^2)$

6.3 Adaptive-Step Size

The numerical integration schemes introduced until now use a step size Δt assumed to be constant throughout the simulation process. These type of numerical algorithms are computationally expensive in systems where the dynamic behavior changes slowly except in some limited regions.

Recall that the order of growth of the global error ultimately depends on the Lipschitz constant K_f, in Eq. (19). This constant represents the worst case deviation of function F as a response to deviations in its parameters [2], for all possible values of $\hat{x}(t)$.

A larger K_f indicates that the global error *may* grow faster, which means that the step size Δt should be smaller. To clarify: if a system has a large K_f, it means that there is at least one pair of values $x(t)$ and $\hat{x}(t)$ for which $\|F(\hat{x}(t)) - F(x(t))\|$ is large. This does not imply the deviations of F are large for all possible pairs of values $x(t)$ and $\hat{x}(t)$. Furthermore, it does not imply that, if the system were to be simulated in a bounded region (e.g, for $0 < t < t_f$), the Lipschitz constant in that region would be smaller. A smaller Lipschitz constant means that the Δt can be larger.

For a given derivative F, it is hard to find the proper K_f in order to pick the right Δt.

An algorithm that can change the Δt throughout the simulation, not only leverages the features of each region in the state space to improve the run-time performance of the simulation but also frees the user from the burden of picking an appropriate Δt. All of this without sacrificing accuracy.

The change of the Δt has to be triggered by some estimate of the error being committed at each simulation step. Assuming the estimate is available, Δt is increased if the error becomes too small and decreased if the error is too large.

The challenge is to come up with a good estimate of the error being committed. Suppose we are given two methods, with local truncation errors $\mathcal{O}(c\Delta t^v)$ and $\mathcal{O}(c'\Delta t^{v'})$, respectively, with c, c', v, v' positive constants. Formally, let $x(t)$

[2] $F(x(t)) = F(\hat{x}(t) + e(t))$, with e being the approximation error

be the solution computed by the first method, $\tilde{x}(t)$ by the second, and $\hat{x}(t)$ be the real solution. Then, after one inaccurate step, solutions $x(t+\Delta t)$ and $\tilde{x}(t+\Delta t)$ can be written as:

$$\begin{aligned} x(t+\Delta t) &= \hat{x}(t+\Delta t) + \mathcal{O}(c\Delta t^{v}) \\ \tilde{x}(t+\Delta t) &= \hat{x}(t+\Delta t) + \mathcal{O}(c'\Delta t^{v'}) \end{aligned} \tag{27}$$

Comparing $x(t+\Delta t)$ with $\tilde{x}(t+\Delta t)$ yields

$$\|x(t+\Delta t) - \tilde{x}(t+\Delta t)\| = \left\|\mathcal{O}(c\Delta t^{v}) - \mathcal{O}(c'\Delta t^{v'})\right\|$$

The bigO notation tells that there exist constants K_1 and K_2 such that, in the limit $\Delta t \to 0$,

$$\left\|\mathcal{O}(c\Delta t^{v}) - \mathcal{O}(c'\Delta t^{v'})\right\| = \left\|K_1 c\Delta t^{v} - K_2 c'\Delta t^{v'}\right\|$$

Assuming that $c' > c$ and that $v' < v$ – the other cases are similar– we have, as $\Delta t \to 0$:

$$\|x(t+\Delta t) - \tilde{x}(t+\Delta t)\| = \mathcal{O}(c'\Delta t^{v'}) = \|\tilde{x}(t+\Delta t) - \hat{x}(t+\Delta t)\|$$

Thus proving that comparing the solutions of the two methods gives an estimate of the error in the same order as the local truncation error of the least accurate method.

From the previous sections, there are two approaches to affect the accuracy of a method: (a) use a smaller step-size and (b) use an higher order approximation method (e.g., the midpoint).

The approach (a) is widely used because of its simplicity: simply take any existing numerical method, compute the solution twice (once with two half steps, and once with a single step), and compare the two estimates.

For an example of approach (b), use the midpoint method to compute the solution $x(t)$, and, at each step, compare it with the result $\tilde{x}(t)$ of the Forward Euler method. It is easy to see that there is some redundant computation in this approach. Fortunately, higher order Runge-Kutta methods can be combined, reusing most of the redundant computation. These are called the Runge-Kutta Fehlberg methods.

6.4 Logic blocks

Decision blocks are widely used in CBDs to increase the expressiveness of the language. The most common decision block is the switch block. The switch block, shown in Fig. 18, outputs the value $u(t)$ or $v(t)$ depending on the value of $c(t)$. If $c(t) \geq 0$, $u(t)$ is the output, otherwise $v(t)$. The translational semantics are:

$$y(t) = \begin{cases} u(t), & \text{if } c(t) \geq 0 \\ v(t), & \text{otherwise} \end{cases}$$

As will be presented shortly, the operational semantics of this block introduce interesting challenges.

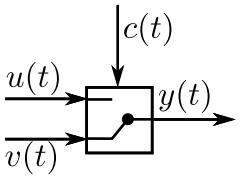


Fig. 18. Switch block

Discontinuity Handling – Zero-Crossing Detection and Location

Recall that the simulation of continuous time CBDs can only be performed approximately in a digital computer. See Section 5.2. This means that the simulation of a continuous CBDs is actually a discrete set of points

$$x(0), x(1\Delta t_1), x(2\Delta t_2), \ldots$$

computed with an adaptive step size method (see Section 6.3).

Suppose the time is t and the simulator is going to compute the solution to the output of the switch block $y(t + \Delta t)$. Furthermore, assume that $y(t) = u(t)$, that is, $c(t) \geq 0$. If $c(t + \Delta t) \leq 0$, then two issues can be identified:

1. $y(t + \Delta t) = v(t + \Delta t)$ may be very different than $y(t)$, because $v(t + \Delta t) \neq u(t + \Delta t)$.
2. $t + \Delta t$ may not represent the exact time at which the signal $c(t)$ crossed the zero. That is $c(t + \Delta t) = 0 - \delta$, for some $\epsilon > 0$.

The second issue implies that, by the intermediate value theorem, there exists at least one point $t^* \in [t, t + \Delta t]$, at which $c(t + \Delta t) = 0$. Ideally Δt should be picked in a way such that $t + \Delta t \approx t^*$, thus minimizing δ, for two reasons:

- Accuracy is improved since all the blocks that depend on the solution $x(t + \Delta t)$ will produce outputs that are close to the switching point t^*;
- Integrator blocks, which apply the numerical methods presented in Section 6.2 may need to be aware of the discontinuity in their inputs, caused by the discontinuity of y around the point t^* (issue 1 above).

To see why this can be a problem, consider the abstract CBD shown in Fig. 19. It can be written as a differential equation (recall Fig. 13):

$$x'(t) = F(x(t))$$

At the time of the discontinuity t^*, in the limit $\epsilon \to 0$, $x(t^* - \epsilon) = x(t^* + \epsilon)$, but $F(x(t^* - \epsilon)) \neq F(x(t^* + \epsilon))$ because of the switch block. This causes a fundamental assumption about the behavior of F – the Lipschitz condition, Eq. (19) – to be violated. Formally,

$$\|F(x(t^* + \epsilon)) - F(x(t^* - \epsilon))\| \leq K_f \|x(t^* + \epsilon) - x(t^* - \epsilon)\| \leq 0$$

is a contradiction.

Without the Lipschitz condition assumption, it is hard to guarantee an order for the growth of the global error. There are multiple ways to address this

problem, once the exact time of the discontinuity is located. See [9, 15]. We focus here in the location of the time of the discontinuity (also called root-finding, or zero crossing location in the literature).

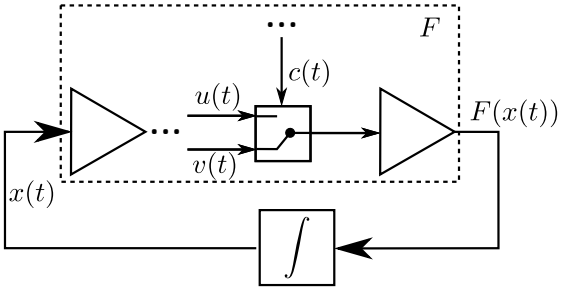


Fig. 19. Abstract CBD which may violate the condition in Eq. (19).

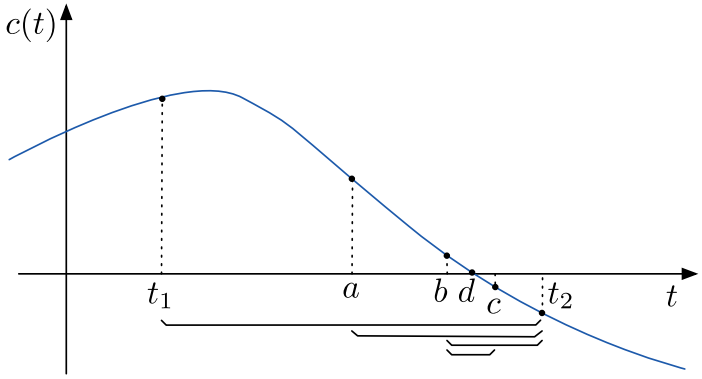


Fig. 20. The bisection method

Different algorithms have been proposed over the years (see [4, 7]). The essence is always the same: locate t^* in an interval $[t, t + \Delta t]$ such that the condition of the switch $c(t + \Delta t) = 0$.

A robust yet simple algorithm is the bisection method. As the name hints, the method works by iteratively bisecting the interval. At each iteration it selects the subinterval where the zero-crossing is present to search for the zero location. The algorithm is shown in Fig. 20. The initial steps detects a zero-crossing in the interval between t_1 and t_2. The iterative procedure evaluates first point a, then point b, then point c and finally point d that is within the tolerance bounds.

Other algorithms are described in the literature. For example, the false position method (or regula-falsi) method tries to reduce the number of iterations by using a linear approximation between the two calculated points. For example, referring to Fig. 20, the point a is calculated as $a = t_2 - \frac{t_1 * c(t_2) - t_2 * c(t_1)}{c(t_2) - c(t_1)}$.

7 Conclusions

Causal Block Diagrams represent a formalization of the intuitive graphical notation of blocks and arrows. This chapter introduced the different variants of this formalism, in a gradual manner.

Algebraic CBDs represent algebraic systems, i.e., where there is no notion of passing time. Discrete time CBDs mixes in the passage of time, although at discrete points. These are analogous to difference equations. Finally, continuous time CBDs, were time is a continuum, correspond to differential equations.

To connect these three variants, a running example of a cruise control system was used. The most typical use cases are:
Algebraic CBDs – study the steady state behavior of systems;
Discrete time CBDs – represent computation and software components;
Continuous time CBDs – modeling physical systems;

In order to perform simulation, discrete time CBDs can also be used to represent a discretized approximation of continuous time CBDs, just like difference equations are used to approximate differential equations.

The advantage of CBDs over plain difference/differential equations is the natural support for hierarchical descriptions of complex systems, providing a way to manage complexity.

The disadvantage is in the ability to reuse models of physical components, represented as CBDs. Physical objects do not have a notion of inputs and outputs. They are best modeled with equations where any variable can be an input/output, depending on whether it is known. This way, the same component can be reused in many different settings, with its input/outputs defined upon instantiation. In CBDs, the modeler is forced to think of the possible instantiations of the model, and define the inputs/outputs accordingly.

CBDs are widely used in the development of embedded systems. Understanding their semantics and the numerical techniques employed provides the reader with a stepping stone into understanding other modeling languages.

8 Further Reading

Among the references already cited, we highlight: [7] provides an extensive overview of the simulation of continuous systems. [12] gives a good introduction to continuous system modeling and simulation, for the reader with a background in Computer Science. [4] provides a mathematically oriented description of multiple numerical techniques. Last but not least, [9] and [15] provide an overview of the the challenges involved in hybrid system simulation, of which CBDs with logic blocks are part of.

Acknowledgment

This work was partially funded with PhD fellowship from the Agency for Innovation by Science and Technology in Flanders (IWT). Partial support by the Flanders Make strategic research center for the manufacturing industry is also gratefully acknowledged.

References

1. Karl Johan Åström, Hilding Elmqvist, and Sven Erik Mattsson. Evolution of Continuous-Time Modeling and Simulation. In *ESM*, pages 9–18, 1998.
2. L A Belady, M W Blasgen, C J Evangelisti, and R D Tennison. A computer graphics system for block diagram problems. *IBM Systems Journal*, 10(2):143–161, 1971.
3. R D Brennan. Digital simulation for control system design. In *Proceedings of the SHARE design automation project on - DAC '66*, DAC '66, pages 1.1–1.12, New York, New York, USA, 1966. ACM Press.
4. Richard L. Burden and John Douglas Faires. *Numerical Analysis*. Cengage Learning, 9 edition, 2010.
5. J.C. Butcher. *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons, Ltd, Chichester, UK, jun 2003.
6. François Edouard Cellier. *Continuous system modeling*. Springer Science & Business Media, 1991.
7. François Edouard Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer Science & Business Media, 2006.
8. F Gracer and R A Myers. Graphic Computer-assisted Design of Optical Filters. *IBM Journal of Research and Development*, 13(2):172–178, mar 1969.
9. Pieter J. Mosterman and Gautam Biswas. A theory of discontinuities in physical system models. *Journal of the Franklin Institute*, 335(3):401–439, apr 1998.
10. Ernesto Posse, Juan de Lara, and Hans Vangheluwe. Processing causal block diagrams with graphgrammars in atom3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*, pages 23–34, 2002.
11. Robert Tarjan. Depth-first search and linear graph algorithms. *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, 1(2), jun 1971.
12. Job van Amerongen. *Dynamical Systems for Creative Technology*. Controllab Products B.V., Enschede, 2010.
13. P P J Van den Bosch and P Bruijn. The directed digital computer as a teaching tool in control engineering; interactive instruction and design. In *Proceedings of the IFAC Symposium on Trends in Automatic Control Education*, pages 260–271, 1977.
14. Herman Van der Auweraer, Jan Anthonis, Stijn De Bruyne, and Jan Leuridan. Virtual engineering at work: the challenges for designing mechatronic products. *Engineering with Computers*, 29(3):389–408, 2013.
15. Fu Zhang, Murali Yeddanapudi, and Pieter Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. In *IFAC World Congress*, pages 7967–7972, 2008.