Block Diagrams of Dynamical Systems

Zeke Medley [1]

1 Introduction

We describe a way block diagrams can be used to model dynamical systems. In Section 2, we describe how these diagrams can be executed. To execute a diagram, initial values need to be provided, so in Section 3 we discuss how determine if the provided initial conditions are valid without executing the diagram. We close with a discussion of related work and summary of results in Sections 4 and 5.

1.1 Block Diagrams

A block diagram is a set of blocks B and a relation W over B. If $(a, b) \in W$, then there is an arrow from block a to block b.
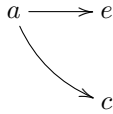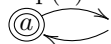

Figure 1: $B = \{a, e, c\}$ and $W = \{(a, e), (a, c)\}$.

When a diagram is executed, initial values are placed on a subset of blocks, then those values are propagated throughout. A block receives a value if all of its inputs do. For a set $I \subseteq B$ of blocks with initial values, we denote the set of blocks which receive values during propagation $\mathrm{p}(I)$.

$\mathrm{p}(I)$ is determined iteratively. For example, in e, we say a has an initial value, so as a is e's only input, $e \in \mathrm{p}(I)$. Having determined e receives a value during propagation, as e is a's only input, $a \in \mathrm{p}(I)$. So, $\mathrm{p}(I) \equiv \{a, e\}$.

To be precise, the set of inputs to block b, denoted N_b, is $\{a \mid (a, b) \in W\}$. b receives a value during

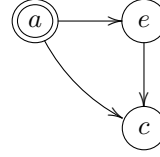Figure 2: A diagram after propagation. Blocks circled once are elements of $\mathrm{p}(I)$, blocks circled twice elements of I. That is, $I = \{a\}$ and $\mathrm{p}(I) = \{e, c\}$.

propagation if $N_b \setminus I \subseteq \mathrm{p}(I)$. Notably, $b \in I \nrightarrow b \in \mathrm{p}(I)$, as shown in Figure 2. We make use of this fact in Section 2 to analyze loops.

1.2 Dynamical Systems

The next state, x^+, of a dynamical system is a function of the previous state. That is, $x^+ = f(x)$. There is a natural translation of dynamical systems to block diagrams, as in Figure 3.

Figure 3: A block diagram of $x^+ = f(x)$

For the dynamical system in Figure 3 to be executable it needs an initial value. Letting x_f denote this value, the first output of the system is $x^+ = f(x_f)$, and thereafter its outputs are $f(f(x_f))$, $f(f(f(x_f)))$, ... Figure 4 shows how this initial value is denoted on a block diagram.

Figure 4: Figure 3 drawn with $I = \{f\}$.

From visual inspection, it is clear Figure 4 will loop forever and Figure 2 will not. A natural question is, how to procedurally determine the order an arbitrary diagram will execute in, and, by extension, if it will loop forever or halt. This is answered in Section 2.

A common extension to the dynamical system of Figure 3 is of the form $x^+ = f(x, u(h(x)))$, illustrated in Figure 5. $h(x)$ denotes what can be observed about

the system and $u(h(x))$ the inputs to the system provided this observation. For example, x might be the state of a boat, $h(x)$ its observed GPS coordinates, and $u(h(x))$ an adjustment to the rudders.
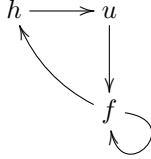

Figure 5: A diagram of $x^+ = f(x, u(h(x)))$.

Care must be taken when placing initial values on Figure 5. As illustrated in Figure 6 for example, consider the case where an initial value is placed on only h, or on both f and u.
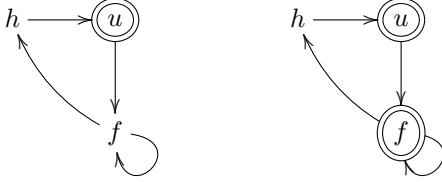

Figure 6: Figure 5 with $I = \{u\}$ (left) and $I = \{u, f\}$ (right).

When a value is placed on only u, what should the next output of f be? $f(\_, x_u)$? As f's next value circularly depends on its previous one, and no initial value is provided, we say f does not take on a value. As not all blocks take on values, we say the diagram is under-determined with these initial conditions.

When a value is placed on both u and f, let x_f denote the initial value of f and x_u the initial value of u. As all of f's inputs have values, suppose f is executed next [1]. The first output of f is then $f(x_f, x_u)$, but it is expected to be $f(x_f, u(h(x_f)))$. Without executing the diagram, we can't determine if $x_u \equiv u(h(x_f))$, so we would like to reject these initial conditions and say the diagram is over-determined.

---

[1] One could argue h should be executed next, then u. In which case, what happens to the previous output of u, x_u? Is it overridden? In which case we get $f(x_f, u(h(x_f)))$ as desired. Is the new output added to a queue? Either way, we would like to reject initializations with overridden values and queues.

Section 3 explores how to detect over and under-determined initial conditions. Section 4 discusses related work and how over-determinedness is similar to Synchronous Kahn Networks [3], which disallow multiple values being in a queue as discussed in [1]. However our definition is more restrictive, as a Synchronous Kahn Network would admit the over-determined initialization of Figure 6 using the execution order h, then f, then u.

2 Execution

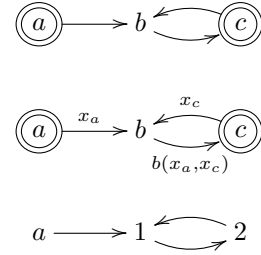Before giving a rigorous description of execution, consider the examples in Figures 7, 8, and 9.

Figure 7: Top: A diagram with $I = \{a, c\}$. Middle: Each wire labeled with its first value. Bottom: Blocks b and c are executed, then execution halts.

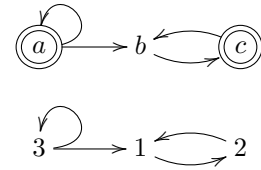Figure 8: Top: Adding (a, a) to W in the diagram from Figure 7 causes it to loop forever. Bottom: A possible execution order.

Figure 7 shows a diagram which does not loop. After propagating the initial values from a and c, b can be executed, b's new output then propagates into c and c is executed. Afterwards, b's input from c has a new value, but b's input from a doesn't, so b cannot be executed again.
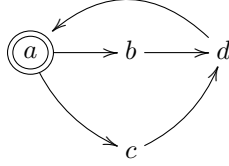
Figure 9: A diagram with multiple possible execution orders. b may be executed before c, or c before b.

Figure 10: Example diagrams and the result of propagation. Blocks circled twice have initial values.

Contrast this with Figure 8 where a feeds back into itself yielding $a(x_a)$ as its new value. In this case, all b's inputs have new values, so b executes again, and this repeats in a loop forever.

In Figures 7 and 8 there is only one sensible execution order. In Figure 9, a then b then c then d, and a then c then b then d are both sensible. As d doesn't output a new value until all its inputs do, either order yields the same result. When multiple orders are valid, our algorithm for determining execution picks one arbitrarily.

We now formally describe execution.

Over a block diagram, for some $I \subseteq B$ of blocks with initial values, $\mathrm{p}(I)$ denotes the set of blocks which receive values during propagation. $\mathrm{p}(I)$ is defined by a calculus $\mathfrak{C}$ with one rule.

$$(\text{P1}) \quad \frac{N_b \setminus I}{b} \quad \text{if } N_b \not\equiv \varnothing$$

The rule (P1) reads, for a block b with a non-zero number of inputs, if $N_b \setminus I \subseteq \mathrm{p}(I)$, then $b \in \mathrm{p}(I)$. Repeated applications of (P1) yield $\mathrm{p}(I)$. Figure 10 shows four examples of $\mathrm{p}(I)$.

Assertions about $\mathrm{p}(I)$ can be proven by means of induction over $\mathfrak{C}$. To show a property F holds for all elements of $\mathrm{p}(I)$, it is sufficient to show F holds when $\mathrm{p}(I) \equiv \varnothing$, then to show if F holds before (P1) is applied then F holds afterwards. In principal, this is the same method of proof as induction over the natural numbers. For a detailed treatment of this method see Chapter II, Section 4 of [5].

Towards an algorithm for determining execution order, we prove two lemmas about propagation.

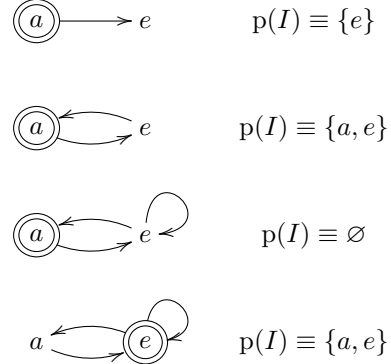**Lemma 1.** $I \subseteq J \rightarrow p(I) \subseteq p(J)$.

Proof. Intuitively, decreasing the number of blocks with initial values will not increase the number of blocks which receive values during propagation. Formally, let F be the property $b \in \mathrm{p}(I) \rightarrow b \in \mathrm{p}(J)$. Evidently, if F is true, then $\mathrm{p}(I) \subseteq \mathrm{p}(J)$.

Base case, when $\mathrm{p}(I) \equiv \varnothing$, F is vacuously true. For the inductive step, assume F holds before the application of (P1) and (P1) subsequently admits b.

$$
\begin{aligned}
b \in \mathrm{p}(I) &\rightarrow N_b \setminus I \subseteq \mathrm{p}(I) & (\text{P1}) \\
&\rightarrow N_b \setminus I \subseteq \mathrm{p}(J) & (\text{Induction}) \\
&\rightarrow N_b \setminus J \subseteq \mathrm{p}(J) & (I \subseteq J) \\
&\rightarrow b \in \mathrm{p}(J) & (\text{P1})
\end{aligned}
$$

$\square$

To represent a dynamical system, let V_t denote the set of blocks which have values propagated to them at time t. For a set of initial conditions I, $V_0 := \mathrm{p}(I)$ and $V_{t+1} := \mathrm{p}(\{b \mid b \in I \wedge b \in V_t\})$. That is, if a block with an initial value receives a new value during propagation, the new value is its value at time $t+1$ and propagation is performed again to compute the values of other blocks at $t+1$.

Lemma 2. $\exists t V_{t+1} \equiv V_t \leftrightarrow \forall n > 0, V_{t+n} \equiv V_t$.

Proof. Propagation has reached a fixed point. By

3

induction.

$$\exists t V_{t+1} \equiv V_t \leftrightarrow V_{t+2} \equiv p(\{b \mid b \in I \wedge b \in V_{t+1}\})$$
$$\equiv p(\{b \mid b \in I \wedge b \in V_t\})$$
$$\equiv V_{t+1}$$
$$\equiv V_t$$

$$V_{t+n+1} \equiv p(\{b \mid b \in I \wedge b \in V_{t+n}\})$$
$$\equiv p(\{b \mid b \in I \wedge b \in V_t\})$$
$$\equiv V_{t+1}$$
$$\equiv V_t$$

$\square$

If a fixed point is reached during propagation and $V_t \not\equiv \varnothing$, then blocks will take on values for the rest of time, so we say the diagram does not halt. Otherwise, for a fixed point $V_t \equiv \varnothing$, no block will take on a value again, so we say the diagram halts.

Algorithm 1 $p(I)$

1: $S \leftarrow \{b \mid N_b \subseteq I \wedge N_b \not\equiv \varnothing\}$
2: $L \leftarrow$ empty list
3: **while** $S \not\equiv \varnothing$ **do**
4:     $b \leftarrow$ any element of S
5:     $L \leftarrow L \cup b$
6:     $S \leftarrow S \setminus b$
7:     **for** each b' where $b \in N_{b'}$ **do**
8:         **if** $b' \notin L \wedge N_{b'} \setminus I \subseteq L$ **then**
9:             $S \leftarrow S \cup b'$
10:         **end if**
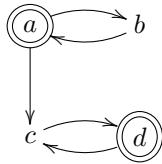11:   **end for**
12: **end while**
13: **return** L

Figure 12: Algorithm 1 returns [c, b, a, d] when executed on the diagram.

Algorithm 1 describes a concrete implementation of the propogation function based on Kahn's algorithm for topological storting [9].

S is initialized with all blocks satisfying P1 due to initial values alone. During each iteration of the loop, a block is removed from S and appended to L; Then, all new blocks satisfying P1 are added to S. Once no more applications of P1 are possible, L is returned.

Notably, L is a valid execution order for the diagram: If a block b is added to L either all its inputs have initial values or all its inputs have received values during propagation.

Algorithm 2 Execution Order

1: $V_t \leftarrow p(I)$
2: $L \leftarrow$ empty list
3: **loop**
4:     $V_{t+1} \leftarrow p(\{b \mid b \in I \wedge b \in V_t\})$
5:     **if** $V_{t+1} \equiv \varnothing$ **then**
6:         **return** (L + V_t, empty list)
7:     **end if**
8:     **if** $V_{t+1} \equiv V_t$ **then**
9:         **return** (L, V_t)
10:     **end if**
11:     $L \leftarrow L + V_t$
12:     $V_t \leftarrow V_{t+1}$
13: **end loop**

Algorithm 2 builds off Algorithm 1 to determine the order a diagram executes. Per Lemma 2, the condition on line 5 being true means execution halts, and line 8 means execution will loop forever. A tuple containing a sequence of blocks to execute once and a sequence of blocks to execute in an infinite loop is returned.

Lemma 3. *Algorithm 2 halts on finite diagrams.*

Proof. Per Lemma 1, $V_{t+1} \subseteq V_t$ after execution of line 4. If the condition on line 8 is false, $V_{t+1} \subset V_t$. So each each iteration of the loop reduces the size of V_t by at least one. And when V_t is empty Algorithm 2 halts, so Algorithm 2 halts on finite diagrams. $\square$

For $b \in B$ let $b(\dots)$ denote the result of evaluating b on some inputs. If $b \in I$, let x_b denote b's initial
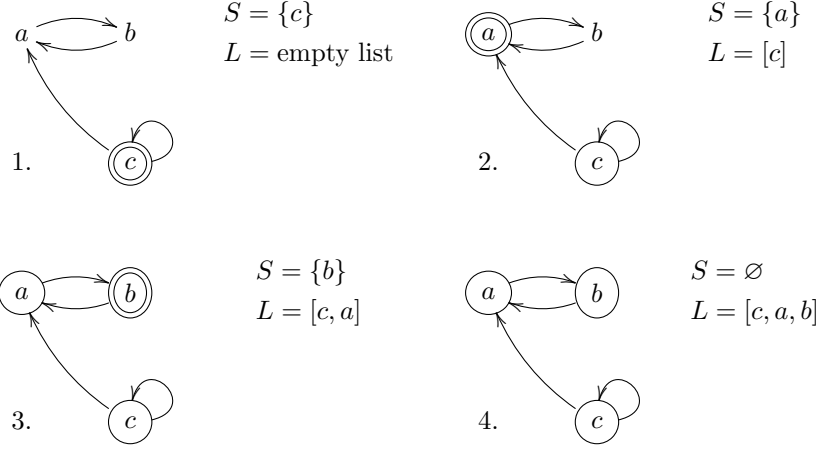
Figure 11: Visualizing Algorithm 1's execution.

value. Using this notation, diagrams can be compiled into imperative code using the execution order from Algorithm 2. We provide two examples for Figures 13 and 14.
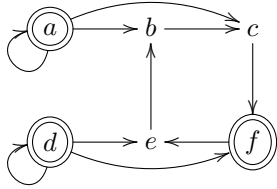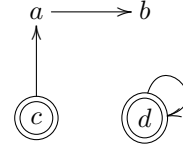

Figure 14: Algorithm 2 returns ([a, b, d], [d]) when executed on the diagram. A non-empty list is returned in the first position when analyzing diagrams with disjoint sequential and looping components.

Figure 13: Algorithm 2 returns (empty list, [a, d, e, b, c, f]) when executed on the diagram.

Algorithm 4 Figure 14 Compiled

1: $x_a \leftarrow a(x_c)$
2: $x_b \leftarrow b(x_a)$
3: $d^+ \leftarrow d(x_d)$
4: $x_d \leftarrow d^+$
5: **loop**
6:      $d^+ \leftarrow d(x_d)$
7:      $x_d \leftarrow d^+$
8: **end loop**

---

3 Soundness

As discussed in Section 1.2, the canonical representation of a dynamical system is an equation of the form

---

Algorithm 3 Figure 13 Compiled

1: **loop**
2:      $a^+ \leftarrow a(x_a)$
3:      $d^+ \leftarrow d(x_d)$
4:      $x_e \leftarrow e(x_d, x_f)$
5:      $x_b \leftarrow b(x_a, x_e)$
6:      $x_c \leftarrow c(x_a, x_b)$
7:      $f^+ \leftarrow f(x_d, x_c)$
8:      $x_a, x_d, x_f \leftarrow a^+, d^+, f^+$
9: **end loop**

$x^+ = f(x)$. That is, the next value of the system is a function of the previous one. Some initial conditions stop this from being the case. When this happens we say the diagram is unsound.

Two ways to think intuitively about unsound diagrams are given in Figures 15 and 16. In Figure 15, the next value of block a is not a function of its previous value, which isn't consistent with a dynamical system. In Figure 16, the next value of a is a function of its previous value, but block c has multiple output values. As we want each block to have a one value at a time, this diagram is also unsound.
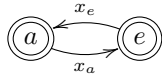

Figure 15: A diagram with unsound initial conditions where $x_a^+ = a(x_e)$.

The problems of Figures 15 and 16 occur when a block both has an initial value and a value computable from other blocks with initial values. Thus, we can ensure every block has a single value by requiring

$$b \in I \to b \notin \mathrm{p}(I \setminus b).$$

However, a diagram like $\textcircled{a} \longrightarrow c$ satisfies this condition but has no feedback, and thus is an unsatisfactory model of a dynamical system of the form $x^+ = f(x)$. To address this, we introduce the notion of an *uninterrupted path* (abbreviated up) between
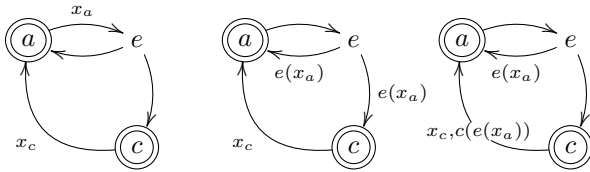

Figure 16: From left to right, three possible steps of execution. After step 2, c has a new value on its inputs, yielding a new output after step 3, but then c has multiple outputs.

two blocks. Over a diagram with initial conditions I, there is an uninterrupted path between a and b if there is a path between a and b which does not pass through any elements of I. Formally,

$$\mathrm{up}(a, b, I) \leftrightarrow (a, b) \in W \vee \exists b' \in N_b \setminus I, \mathrm{up}(a, b', I).$$

For example, $\textcircled{a} \rightleftarrows c$ has an uninterrupted path from a to a, but not from c to c. In the sequel we prove two lemmas which relate uninterrupted paths to soundness, building up to a proof that

$$(b \in I \to \mathrm{up}(b, b, I)) \to (b \in I \to b \notin \mathrm{p}(I \setminus b)).$$

This being true means requiring each block with an initial value to feed back into itself is sufficient to ensure each block has one value at a time, making the diagram a satisfactory model of a dynamical system.

Lemma 4. $(a \notin I \wedge up(a, b, I)) \to (b \in p(I) \to a \in p(I))$

Proof. Intuitively, in $\textcircled{e} \longrightarrow a \longrightarrow b$, if b has a value propagated to it, then a must also. The reason for the requirement $a \notin I$ is to address diagrams like $\textcircled{a} \longrightarrow b$ where b receives a value during propagation due to a having an initial value, but a does not. To prove this, we use induction which allows us to reason about cases where a and b are connected via a long uninterrupted path, as in $\textcircled{e} \longrightarrow a \longrightarrow \ldots \longrightarrow b$.

By induction on the length of the path. For a path of length one, $(a, b) \in W$ so by definition $a \in N_b$. So as $a \notin I$, $a \in N_b \setminus I$. Thus, by (P1) $b \in \mathrm{p}(I) \to N_b \setminus I \subseteq \mathrm{p}(I) \to a \in \mathrm{p}(I)$. For a path of length greater than one, $(a, b) \notin W$ so

$$\mathrm{up}(a, b, I) \wedge b \in \mathrm{p}(I)$$
$$\to \exists b' \in N_b \setminus I, \mathrm{up}(a, b', I) \wedge N_b \setminus I \subseteq \mathrm{p}(I)$$
$$\to \mathrm{up}(a, b', I) \wedge b' \in \mathrm{p}(I)$$
$$\to a \in \mathrm{p}(I) \qquad \text{(Inductive hypothesis)}.$$

$\square$

Lemma 5. *If* $b \in I \to up(b, b, I)$, *then* $b \in I \to b \notin p(I \setminus b)$.

Proof. Intuitively, in $\textcircled{e}\longrightarrow b$ because there is an uninterrupted path between e and itself, Lemma 4 approximately tells us, for e to have a value e must have a value, i.e. there is a cyclic dependency. As a result, removing the initial value on e will cause e to no longer have a value. If this is the case for all blocks with initial values, then the diagram is sound. To formally prove this, we once again use induction on the length of the uninterrupted path.

For some $I \subseteq B$ and $b \in I$, let $J = I \setminus b$ and let $p(J)$ be defined by $\mathfrak{C}$. We prove the predicate $b \notin p(J)$ via induction on $\mathfrak{C}$: For the base case, if $p(J) \equiv \varnothing$ then the predicate holds vacuously. For the inductive step we once again consider paths of length one and greater than one.

For a path of length one,

$$\begin{aligned} b \in I \wedge (b,b) \in W &\to b \in N_b \\ &\to b \in N_b \setminus (I \setminus b). \\ &\to b \in N_b \setminus J. \end{aligned}$$

Per the inductive hypothesis $b \notin p(J)$, so as $b \in N_b \setminus J$, $N_b \setminus J \not\subseteq p(J)$, so b will not be admitted by (P1).

For a path of length greater than one, as $J \subseteq I$ note

$$\exists b' \in N_b \setminus I, \mathrm{up}(b, b', I) \to \exists b' \in N_b \setminus J, \mathrm{up}(b, b', J).$$

Towards a contradiction, before b is admitted by (P1) suppose $N_b \setminus J \subseteq p(J)$. Thus, for the same b' as above, $b' \in p(J)$. So by Lemma 4 $b \notin J \wedge \mathrm{up}(b, b', J) \wedge b' \in p(J) \to b \in p(J)$. But this contradicts the inductive hypothesis, $b \notin p(J)$. So $N_b \setminus J \not\subseteq p(J)$ and thus b will not be admitted by (P1). $\square$

To conclude our discussion of soundness, per Lemma 5, to check if a diagram is sound we can check if $b \in I \to \mathrm{up}(b, b, I)$ as follows: For each $b \in I$, check that b is reachable from b without passing over any elements of I. This can be accomplished with a graph traversal (for example, breadth-first search), which is modified to not pass over elements of I. If every $b \in I$ is reachable from itself via this traversal, the diagram is a satisfactory model of a dynamical system.

4 Related Work

Block diagrams as a notation for describing computation has found success in many domains from shader programming [2] to signal processing [1]. In fact, representing and simulating block diagrams of dynamical systems has found commercial success in, for example, Simulink by MathWorks [11], released in 1984. However, these systems are proprietary, so it is difficult to know precisely what static analysis they perform.

In the literature, our execution and static analysis methods resemble Synchronous Kahn Networks [3] and synchronous programming languages [7] [6]. To sketch this equivalence, the blocks here can be thought of a processes in a Kahn Network which wait on all of their inputs and then yield exactly one output, and initial conditions can be modeled with the `pre` operator. Among other things, these models of computation make possible static analysis to ensure execution avoids multiple values being on a channel at once, as in Figure 16; static analysis to detect unbroken feedback loops [4], like the one in Figure 3; And static analysis to compile diagrams to imperative code [3].

However, to the best of our knowledge this analysis has not been extended to detecting over-determined initialization, as in Figures 6 (right) and 15. Though this is not to say we are the first to identify and address this problem. Without purchasing a copy we cannot verify, but Simulink likely performs similar checks with consistency checking enabled. [2] And as we will show, Wolfram System Modeler's [8] notion of over-determined initialization is likely nearly identical to ours.

To see this, note how the Wolfram System Modeler documentation explains over-determined initialization.

> "In overdetermined initialization, a subset of variables in the initialization problem is determined by more equations than the number of variables." [10]

Casting this in terms of our diagrams, we can view

---

[2] See this <u>MATLAB Central answer</u>.

the initialization problem as solving for the value on each block's output, provided with known values from I. To show this by means of example, let x_b denote the output of block b and consider the over-determined diagram of Figure 17.
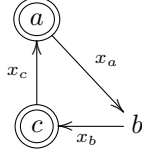

Figure 17: An over-determined diagram where the output of block b is labeled x_b.

For a block $b \in I$, let k_b denote its initial value. With this notation, the initialization problem for Figure 17 becomes the problem of writing each variable in the following system of equations in terms of initial values and function applications.

$$\begin{cases} x_a - a(x_c) \equiv 0 \\ x_b - b(x_a) \equiv 0 \\ x_c - c(x_b) \equiv 0 \\ a(x_c) \equiv k_a \\ c(x_b) \equiv k_c \end{cases}$$

However, in this system of equations we can express the value of x_c two ways: $x_c \equiv c(b(k_a))$ and $x_c \equiv k_c$. Thus, we say the diagram is over-determined as "a subset of variables (x_c) in the initialization problem is determined by more equations than the number of variables." This is equivalent to saying $c \in \mathrm{p}(I \setminus c)$, which corresponds to our definition of soundness.

5 Conclusion

We've discussed how to represent and execute block diagrams of dynamical systems. Section 2 explored how to statically determine the execution order of a diagram, then showed how to compile it to an imperative program. Afterwards, Section 3 showed how to statically detect problematic initial conditions on a diagram. Together, this provides the conceptual basis for an implementation of a software library for simulating dynamical systems. Though practical considerations and remain.

1. Not all blocks can be connected. A block which outputs numbers can't be wired to one which takes a string as input. However, in the case where blocks are implemented in a typed language, the type system ought to catch these errors when compiling the imperative program generated in Section 2.

2. It is difficult to ensure execution order doesn't change the behavior of the system. For example, consider Figure 18. If blocks b and c append to a file, the order they are executed in will change its contents.

3. When multiple execution orders are possible as in Figures 9 and 18, blocks can be executed in parallel. When and how to exploit these opportunities is an open question.

4. In a predator-prey model like fish ⟶ shark where wires carry the location of a fish and shark, our initialization rules say specifying both locations is over-determined. However, this is resolved by observing the fish and shark update their positions based on their current state as well as their counterpart's, resulting in

Thus, internal state should represented as feedback in a software implementation.
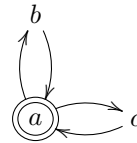

Figure 18: Blocks b and c can be executed in parallel.

References

[1] Cycling '74. *Max: A Playground for Invention*. Accessed: 2024-07-21. 2024. URL: https://cycling74.com/products/max.

[2] Blender Foundation. *Blender Shader Nodes*. Accessed: 2024-07-11. 2024. URL: https://docs.blender.org/manual/en/latest/render/shader_nodes/index.html.

[3] Paul Caspi and Marc Pouzet. "Synchronous Kahn networks". In: *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*. ICFP '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 226–238. ISBN: 0897917707. DOI: 10.1145/232627.232651. URL: https://doi.org/10.1145/232627.232651.

[4] Pascal Cuoq and Marc Pouzet. "Modular Causality in a Synchronous Stream Language". In: *Programming Languages and Systems*. Ed. by David Sands. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 237–251. ISBN: 978-3-540-45309-3.

[5] Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical Logic*. 3rd. Springer, 2021. DOI: 10.1007/978-3-030-73839-6.

[6] Stephen A. Edwards and Edward A. Lee. "The semantics and execution of a synchronous block-diagram language". In: *Science of Computer Programming* 48.1 (2003), pp. 21–42. ISSN: 0167-6423. DOI: https://doi.org/10.1016/S0167-6423(02)00096-5. URL: https://www.sciencedirect.com/science/article/pii/S0167642302000965.

[7] N. Halbwachs. *Synchronous Programming of Reactive Systems*. The Springer International Series in Engineering and Computer Science. Springer US, 1992. ISBN: 9780792393115. URL: https://books.google.com/books?id=-q9MNBEqTekC.

[8] Wolfram Research Inc. *System Modeler, Version 14.0*. Champaign, IL, 2024. URL: https://www.wolfram.com/system-modeler.

[9] Arthur B. Kahn. "Topological Sorting of Large Networks". In: *Communications of the ACM* 5.11 (1962), pp. 558–562. DOI: 10.1145/368996.369025.

[10] Wolfram Research. *Model Initialization*. Accessed: 2024-07-23. 2024. URL: https://reference.wolfram.com/system-modeler/GettingStarted/ModelInitialization.html.

[11] The MathWorks, Inc. *MATLAB Simulink*. Accessed: 2024-07-11. 2024. URL: https://www.mathworks.com/products/simulink.html.