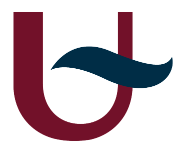

Department of Computer Science

Dynamic Structure Causal Block Diagrams

Master Thesis

Yves Maris

Promotors:
Hans Vangheluwe, University Antwerp
Farnando Barros, University Coimbra

Final version

Antwerp, August 2016

Abstract

Dynamic structure systems can undergo changes to its network structure at any point in time during simulation. This can include the addition, removal or modification of components or relations between components. Allowing this type of behavior can greatly improve expressiveness of a modeling language. We extended causal block diagrams with dynamic structures in a way that is consistent with its standard constructs. Model transformation is allowed trough the addition of a new concept, structure blocks, to the existing language. New components are instantiated and existing ones are removed as specified within the structure block. As a proof of concept we implemented an exhaustive model in a prototype simulator for the new extended language. This work is also a step forward in finding the essence of dynamic structure modeling.

Preface

I would like to thank my promotors Fernando Barros and Hans Vangheluwe for the continued support during my research in the past months.

Contents

Contents vii

List of Figures ix

List of Tables xi

Listings xiii

1 Introduction 1

2 Causal Block Diagrams 3
- 2.1 Background 3
  - 2.1.1 Modeling languages 3
- 2.2 Time bases 4
- 2.3 Abstract Syntax 5
  - 2.3.1 Hierarchical model 5
- 2.4 Denotational semantics 8
  - 2.4.1 Algebraic time model 8
  - 2.4.2 Discrete Time Model 8
  - 2.4.3 Continuous Time Model 8
- 2.5 Operational semantics 10
  - 2.5.1 Flattening process 11
  - 2.5.2 Evaluation order 11
  - 2.5.3 Main simulation algorithm 12

3 Dynamic Structure 13
- 3.1 Related Work 13
- 3.2 Abstract syntax 14
- 3.3 Visual syntax 15
  - 3.3.1 Explicit representation 15
  - 3.3.2 Implicit representation 15
  - 3.3.3 Hybrid representation 17
  - 3.3.4 Completeness of the representations 18
  - 3.3.5 Deciding upon a final representation 19
- 3.4 Dynamic structure CBD semantics 19
  - 3.4.1 Events triggering a structural change 19
  - 3.4.2 The Structure Block 20
  - 3.4.3 Addition of constructs 20
  - 3.4.4 Removal of constructs 21
  - 3.4.5 Reinitialisation 24
  - 3.4.6 Higher order structural change 24
  - 3.4.7 Identifying dynamically created structures 25
  - 3.4.8 Adapted operational semantics 25

3.5 Implementation 26
3.5.1 Python implementation 26
3.5.2 XML parser 27
3.5.3 Visual modeling environment 28
3.5.4 Debugging 29
3.5.5 Unit tests 29
3.5.6 Standard CBD 29
3.5.7 dynamic structure CBD 29
3.5.8 Traceability 30

4 Case Study: Elevator Model **31**
4.1 Ball in a two dimensional space 32
4.1.1 Discretized Position CBD 33
4.1.2 Continuous Position CBD with constant velocity 33
4.1.3 Continuous Position CBD with variable velocity 35
4.1.4 Variable timesteps 37
4.2 Dynamically removing Balls 39
4.3 Dynamically creating Balls 39
4.4 Elevator Model 41
4.4.1 Stopping at floors 41
4.5 Execution plots 43

5 Conclusions **45**

6 Future Work **47**

Bibliography **49**

List of Figures

2.1 Syntax of the addition operation 4
2.2 The abstract syntax of a standard CBD adapted from [8] 6
2.3 Example of an algebraic CBD 6
2.4 Hierarchical CBD without input ports 7
2.5 Hierarchical CBD with input ports 7

3.1 The abstract syntax of a standard CBD adapted from [8] 15
3.2 Explicit visual syntax for simple model 16
3.3 Implicit visual syntax for simple model 17
3.4 Hybrid visual syntax for simple model 18
3.5 Implementation of WCZFB block using basic blocks 20
3.6 CBD illustrating the dynamic addition of structures 21
3.7 CBD illustrating the dynamic removal of structures 22
3.8 CBD illustrating the dynamic addition and removal of structures 23
3.9 CBD illustrating the dynamic Reinitialisation of structures 24
3.10 CBD illustrating the dynamic Reinitialisation of structures 26
3.11 Visual model in AToMPM 29

4.1 Stills of the implementation 32
4.2 Overview of one ball 33
4.3 Discrete time position with constant velocity 34
4.4 Continuous time position with constant velocity 35
4.5 Position with variable velocity 36
4.6 Position that allows variable timesteps 38
4.7 Overview of dynamically removed ball 40
4.8 Overview of dynamically created ball 40
4.9 Variable velocity in elevator model 42
4.10 Y-position of the elevator 43
4.11 Position of ball 1 43
4.12 Position of ball 2 43

List of Tables

2.1 Possible time bases for CBD's 4
2.2 Denotational semantics of Mathematical algebraic time blocks 9
2.3 Denotational semantics of Logical algebraic time blocks 9
2.4 Denotational semantics of delay block 10
2.5 Denotational semantics of continuous time blocks 10

3.1 Comparison of the different visual syntax's 19
3.2 Denotational semantics of event blocks 20

Listings

Chapter 1

Introduction

Causal block diagrams are used in industry for the design of control systems. Yet when we start a simulation with a model we created, the network structure is fixed from that point on. No constructs can be added, removed or modified during simulation. In order to increase expressiveness we propose an adapted formalism in which this is possible. All constructs of the standard CBD formalism will remain the same with the addition of features that provide the dynamic structural changes. We will verify the expressive capabilities as well as the actual response to dynamic structures of this new language with a minimal case study, an elevator model. This elevator will move up and down the floors of a building stopping at each consecutive floor. At this moment doors will open and a ball can enter the elevator. This bal will have a certain starting velocity and will bounce around within the bounds of the elevator. Every time the doors open again they can disappear again when they move outside the scope of the elevator. This model extensively tests the creation and removal capabilities of the model.

This thesis will be structured . Chapter 2 will give a formal definition of (abstract) syntax and semantics of standard causal block diagrams. This is a background chapter meant to give a good understanding of the standard formalism. Chapter 3 will focus on the different ways to develop a dynamic structure formalism starting with a clear definition of the expressiveness we expect. We take an educated approach based on existing dynamic structure formalisms and the method in which they were created. We go in depth about the features of the adapted formalism and introduce a visual syntax to represent it. Section 4 introduced our case study with an iterative approach where features are added each iteration until full functionality is acquired. The model is represented in a comprehensive way by making use of the visual syntax that was earlier introduced.

Chapter 2

Causal Block Diagrams

Causal block diagrams allow us to create a network of blocks that represent mathematical operations [10]. The complete system can thus be described by a set of equations. the simulator provides a way to solve that set of equations. The main application of the formalism lies in the design of control systems. A well known and widely used implementation is by The Mathworks namely Simulink [7]. In this chapter we will discuss the syntax and semantics of the causal block diagram formalism as an introduction to the conducted research on dynamic structures in causal block diagrams. Since its formal introduction under the more general term block diagrams in the late 90's, causal block diagrams have been the topic of a few research projects. The semantics have been earlier formalised by providing an executable implementation in Haskell [6].

2.1 Background

2.1.1 Modeling languages

A modeling language, just like any programming language, can be described by it's syntax and semantics. The syntax of a language can be decomposed in concrete syntax or abstract syntax. The concrete syntax is simply the way an object is represented while the abstract syntax is the essence of this object. This means one abstract syntax can be represented by multiple concrete syntaxes. This gives the developers and users of modeling languages a lot of freedom since the visual representation is completely decoupled from the underlying model. The visual representation can be chosen depending on the circumstances. Probably the simplest example of is situation where this can be of use is providing a gui and a textual interface. While the gui might be more intuitive to use for people that are not familiar with the language, the textual model might be more efficient. Providing both will make sure all users needs are met. To achieve this multiple concrete syntaxes must be mapped to one abstract syntax. The operation where a concrete syntax is translated to an abstract syntax this is called parsing. This will typically be a translation to a in-memory data structure that represents the operation. The inverse of parsing is called printing. This concept is illustrated in Figure 2.1 where the addition operation is represented with two concrete syntaxes

Since the syntax only gives a representation of the of a certain operation and not the actual meaning behind it a description for this is also needed. We call this the semantics. We can consider a translation from the syntax (either abstract or concrete) to the semantics which is called the semantic mapping. The actual meaning is called the semantic domain. Once again we can make a distinction between different types of semantics. First we can consider operational semantics, this will tell you the meaning by giving a simulation algorithm. It tells you how values are computed. Denotational will give you a more mathematical meaning of the operation where the semantics is mapped onto known concepts. While operational semantics is obviously the most suitable for

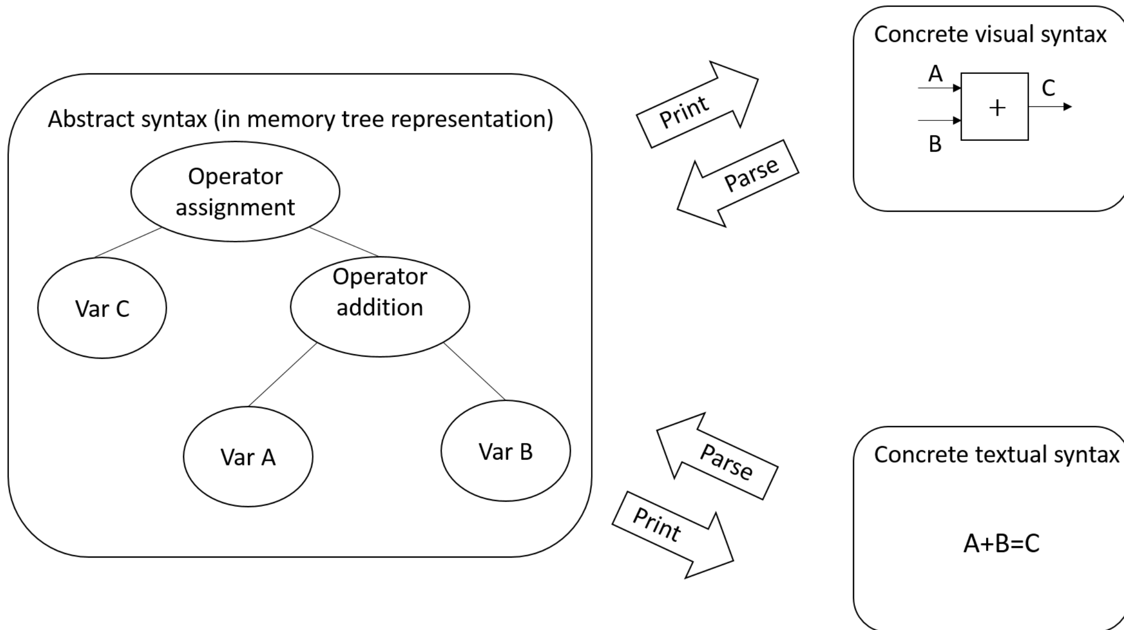

Figure 2.1: Syntax of the addition operation

the implementation and computation of a model, denotational semantics still has its use. Since it gives you a mathematical representation of the model, it can be used to proof certain properties. This makes deeper reasoning about the model possible. While you can have both denotational and operational semantics they will both represent exactly the same meaning.

2.2 Time bases

We can consider CBDs over a number of different time bases. A concise description can be found in Table 2.1.Every time we move down a row in this table the time basis gets more precise meaning the model gets more expressive. In the next sections of this chapter we will go in depth about which blocks allow transitioning from one time base to another. It is also worth mentioning that the continuous time base is not truly continuous but rather a discretized version. This is because computation happens in a discrete way and values are only recalculated at specific points in time. Between two consecutive executions the signal remains constant. With a discrete time approximation the smaller the timestep, the more accurate the approximation. In fact we can view any continuous time model as a discrete time model with an infinitesimally small timestep.

Table 2.1: Possible time bases for CBD's

Time base	Name
$T \in now$	algebraic
$T \in \mathbb{N}$	discrete
$T \in \mathbb{R}$	continuous

2.3 Abstract Syntax

In all the time bases that will be discussed in this chapter some properties will remain the same. Lets start by analysing the name causal block diagram. First of all it tells us there is a sense of causality or in other words input and output of model components are known. This means no causality assignment is required before starting the simulation. This is however the case in acausal block diagrams. In this formalism constraints are imposed by the blocks without a given causality. This leads to even greater expressiveness bu it also means there will be a need for a causality assignment. When this causality assignment is done the acausal model is basically transformed into a causal model. Going on from this point the model can be treated the same as a causal block diagram. This work will focus more on causal block diagrams but expansion to acausal block diagrams is possible yet it will require additional research.

Figure 2.2 gives a representation of the abstract syntax of a CBD. A model consists, as the name suggests, of a number of connected blocks of computation which are contained within a CBD. Each one of these blocks will be derived from a base block and will represent a certain operation. Blocks can generate output or receive input trough ports. Input and output ports are connected to each other trough connections. A signal is send trough an output port and received by an input port. This signal is simply a numeric value that is passed between blocks and used for further computation.

2.3.1 Hierarchical model

A CBD allows a hierarchical composition of a model, where a CBD is embedded within another CBD. This allows to decompose the model into smaller components. With complex models this can be of use to preserve readability of the model. In order for this hierarchical composition to work a CBD must be able to receive input and generate output just like blocks as shown in Figure 2.2. With this adaptation a CBD can be used in exactly the same way as a block would be. The output ports of a CBD can be either connected to the input port of a different CBD or a block. Analogously, the output port of a block can be connected to the input port of a CBD or block.

To explain this concept a simple example will be used. Figure 2.3 shows an algebraic CBD where the sum of two values, two and five, is computed. This result is multiplied with a third value, three. This will be the end result of the computation for this model. The Hierarchy can be introduced into the model in a number of ways. In Section 2.4.1 we will elaborate on the actual semantics of the blocks. Figure 2.4 the complete multiplication is isolated into another CBD. In this case only one input is required for interaction with the rest of the model. The example in Figure 2.5shows that it is also possible for a CBD to have input ports. Here the adder block is isolated and embedded within a different CBD. Because it is only the adder block, without the constant blocks that provide outputs for this, is isolated additional input ports are needed.

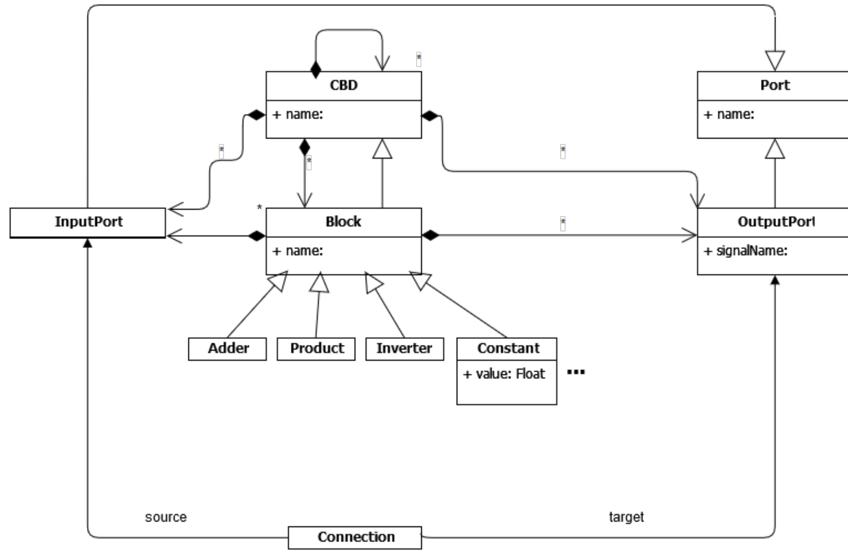

Figure 2.2: The abstract syntax of a standard CBD adapted from [8]

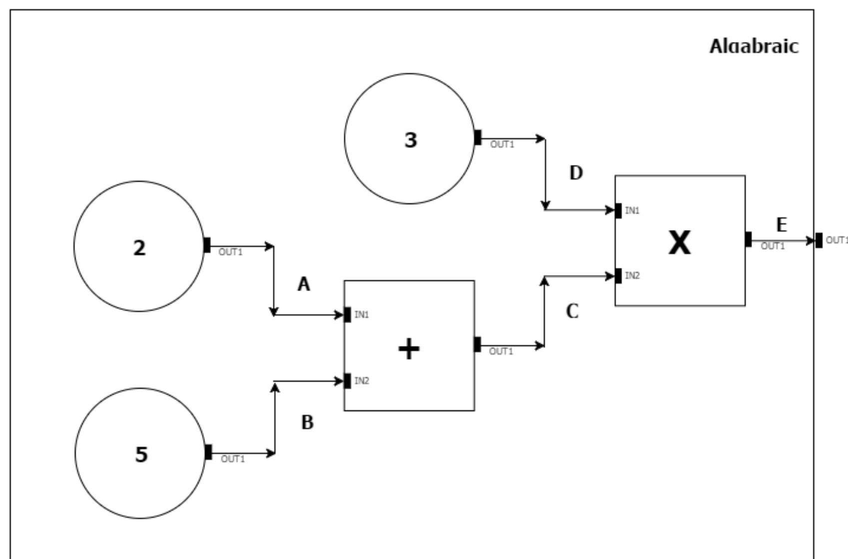

Figure 2.3: Example of an algebraic CBD

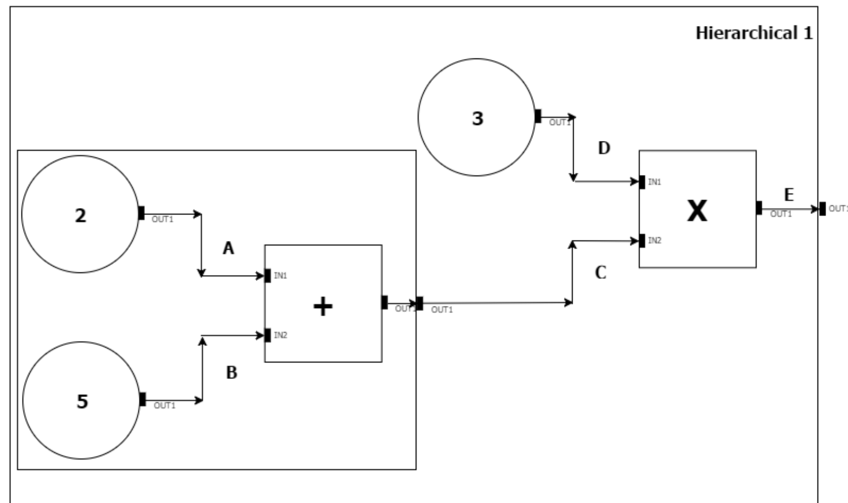

Figure 2.4: Hierarchical CBD without input ports

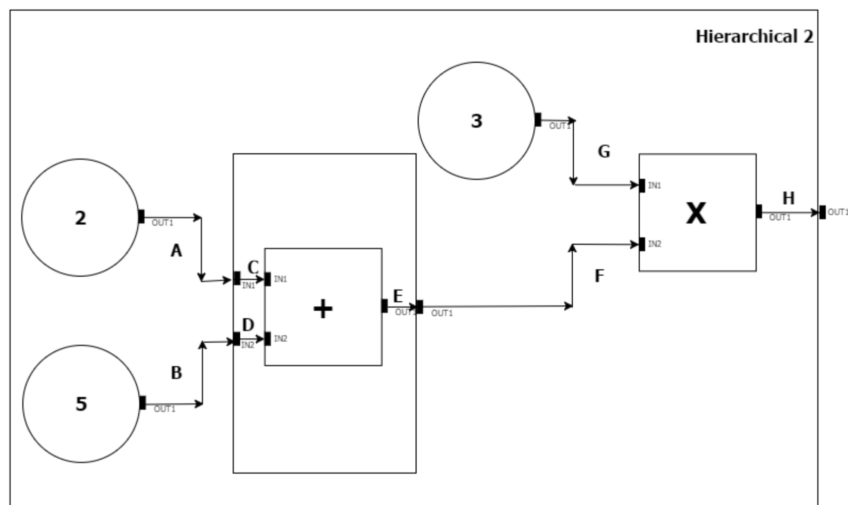

Figure 2.5: Hierarchical CBD with input ports

2.4 Denotational semantics

Previously we defined denotational semantics as the mathematical relations between variables. When we consider a causal block diagram the input and output ports that connect a block to other blocks use signals as a mean of communication. These signals are the can easily be mapped onto variables. When we consider all input and output signals for a given block the relation between them can be expressed mathematically trough equations. To allow the composition of multiple blocks we can simply consider the set of all equations defined by the blocks within the CBD. Since all blocks were represent individual equations, a solver can easily compute the solutions for the complete set. Some blocks might also represent logical operations instead of strict mathematical ones. The underlying principle however remains the same and they will still be treated in the same way mathematical blocks would be. In the remainder of the section we will give an overview of the blocks that make up the different time bases of the CBD formalism.

2.4.1 Algebraic time model

In an algebraic model there is no real sense of time. For each block the output of the respective input is calculated after detected loops have been resolved. The result of a simulation will be that every block has a generated output based on it's input.

Table 2.2 gives the semantics for the most important mathematical blocks while Table 2.3 displays the logical blocks. In the implementation that will be employed for the experiments in this thesis a logical zero will be represented by minus one for convenience sake. The exact reason for this will be discussed in Chapter 4. A logical blocks signal can have two values a logical one or zero. This means the signal will be zero as long as the condition is not satisfied. if the condition is satisfied, the signal changes to one. We call this a piecewise constant signal.

2.4.2 Discrete Time Model

A discrete model introduces a sense of time to the previously discussed model. If the model, as was the case in all the algebraic model, does not depend in any way on values of the past this will not change anything. In every time step the same values will be computed as they would be in the algebraic model.

In order to express a connection between values of the previous and the current iteration, a new block needs to be introduced namely the delay block displayed in 3.2. The delay block feeds a value from a previous time step into the current time step, inducing a need for recomputation. In the first timestep, an initial value is simply used instead of a value from the previous time step.

2.4.3 Continuous Time Model

A continuous time model provides a value for any given point in time. The values computed in the algebraic and discrete time value are exact. No approximations where needed because the time base did not require us to. It is not possible to exactly simulate a continuous model in modern computer architecture. This is why we need some way to discretize the model and simulate it that way. In other words, the continuous model needs to be translated to a discrete model in some way to make it computationally feasible. The two blocks that are added in the continuous time model that where not part of the algebraic or discrete time model. Again an initial value is provided at time zero. These operators also relate a past value to a the current value.

Table 2.2: Denotational semantics of Mathematical algebraic time blocks

Name	Block	Semantics
Adder block	+	$A + B = C$
Product block	X	$A * B = C$
Modulo block	%	$A\%B = C$
NegatorBlock	-	$B = -A$
InverterBlock	1/A	$B = 1/A$
SquarerootBlock	√	$A = B^2$
Ln block	ln	$\ln A = B$

Table 2.3: Denotational semantics of Logical algebraic time blocks

Name	Block	Semantics
AND block	AND	$A\&\&B = C$
OR block	OR	$A\|\|B = C$
Greater than block	>	$A > B = C$
Equals block	==	$A == B = C$
NOT Block	NOT	$!A = B$

Table 2.4: Denotational semantics of delay block

Name	Block	Semantics
Delay block	IC, A, delay, B	$A = B = C$

Table 2.5: Denotational semantics of continuous time blocks

Name	Block	Semantics
Integrator block	IC, A, dt, ∫, B	$if(t = 0) : B = IC$ $else : \int A = B$
Derivator block	IC, A, dt, d/dt, B	$if(t = 0) : B = IC$ $else : \frac{dA}{dt} = B$

2.5 Operational semantics

To develop a simulation algorithm we must first understand the problems that come with it. In order to evaluate a CBD model we want to know the outputs of each block in the model as well as the output of the complete model. In other words values need to be provided to each signal. To achieve this we need to know the input signals for the blocks that generate these output signals.

It is clear that there is a need for some evaluation order that makes sure that the required input signals are always available. A way to achieve this is by starting with the constant blocks. They do not have any input ports and they simply generate a constant output signal. This makes them the ideal starting point. The values generated by the constant blocks can simply be pushed through the CBD until the value for all signals is known. This is an intuitive way to find a correct evaluation order but it isnt an actual algorithm yet.

We can start our reasoning with the observation that a CBD is in actuality a directed graph where the blocks are nodes and the edges are signals. This does not mean that a CBD and a graph are the same thing. A CBD contains information about the operation that a block uses to compute its output. In a graph there is no distinction between the nodes. For example in a graph representation of a CBD it is possible to switch out an adder block for a product block and it will result in the same graph. There is also another way to represent a CBD by a graph that is the inverse of the way we mentioned before, the dependency graph. Here signals are represented by nodes and dependencies between them are represented by signals. A dependency between two signals, A and B, exists if the value of A is needed to compute the value of B.

Up until this point we have assumed a CBD to have a flat structure. If this is not the case a flattening step need to be added before the process is started. This section gives an overview of the steps needed to simulate a CBD model.

2.5.1 Flattening process

The flattening step is crucial in order to achieve correct algebraic loop detection. A loop can be partially embedded within a child block. To be able to detect and resolve this loop the structure needs to be flat, no hierarchy can occur. The flattening process will add the blocks contained within the root CBD to the root CBD. This seems simple enough but there also needs to be a way to handle the output ports of the children. In our case we will choose to do this by replacing all ports by wire blocks. These are simply blocks that pass a signal without any processing. While it is also possible to remove the blocks completely by directly connecting the blocks on either side of the port, this might be less desirable. Traceability to the original model is lost by doing so and additional computation is needed to link the connected blocks.

The first step of the flattening algorithm will detect all the input and output ports and replaces them by wire blocks. In this process all blocks connected to these ports are now carried over to the corresponding wire blocks. Once this is done they can be safely remove from their parent.

The remainder of the flattening process will occur in a recursive way. All CBDs contained within the child CBD will also be flattened resulting in a CBD that only contains computational blocks. All these blocks are finally added to their parent so only one level, the root CBD, remains at the end of execution. For tractability each block will be renamed to the following: newName = parentName + "." + oldName. As you can see this flattening operation is mostly just syntactic sugar, this is to keep overhead of computation to a minimum.

2.5.2 Evaluation order

By using a topological sort algorithm on the dependency graph of a CBD a correct ordering can be generated, the basis of this algorithm is a simple depth first sort traversal of the nodes. In case the dependency cycle contains cycles, this algorithm will give an incorrect order. For example the output of a product block is connected to the input of an adder block. Operationally this will give an incorrect result yet denotational this is perfectly solvable. For this reason it is required to detect these loops and resolve them before starting the actual simulation. In actuality a loop represents a set of equations that can be solved by a Gaussian solver. Yet before this can be done we must check if the set of equations has a unique solution. This is not the case if the equations are a linear combination of each other. This can be checked by determining whether the determinant is 0.

Algorithm 1: Loop detection

```
   Data: graph
   Result: The strong components within graph
 1 topSort(graph);
 2 revGraph = reverseEdges(graph);
 3 strongComponents=[];
 4 for node in revGraph do
 5     Mark node as visited;
 6 while !isEmpty(revGraph) do
 7     startNode = highestOrderNumber(revgraph);
 8     component = dfsCollect(startNode,revNode);
 9     appendResult(component);
10     revGraph.remove(component);
```

In the previous section we described the topological sort of the graph. After this we must detect the algebraic loops by reversing the order. If there are loops present we will determine whether the corresponding equations are linear. If they are not linear we will not be able to

solve it, otherwise we feed the equations in the Gaussian solver. In the last step the remainder of the model is simulated. Result of strong component detection algorithm which can be found in Algortihm 2 is a set of sets. If all sets are singletons, no strong components are present. In a strong component every node is reachable form any other node meaning a loop exists. When this is the case we need to use the Gaussian solver.

2.5.3 Main simulation algorithm

The simulation algorithm is displayed in Algorithm 2, subsequent timesteps are computed in a loop. In each iteration a dependency graph is created and algebraic loops are detected, lifted out and resolved and the rest of the model is computed. When a delay block is present in a loop, it breaks this loop. This is the case because it is dependent on a value from a previous timestep and not the current one. Each individual block is computed via the evaluation order that was calculated and finally the time is increased at the end of the iteration.

Algorithm 2: Operational semantics CBD (adaptation from [8] and [15])

```
Data: cbd
Result: Behaviour trace
1 logicalTime = 0;
2 flatcbd = FLATTEN(cbd);
3 while not end_condition do
4     schedule = LOOPDETECT(DEPGRAPH(flatcbd));
5     for block in schedule do
6         COMPUTE(block)
7     logicalTime = logicalTime + δt
```

Chapter 3

Dynamic Structure

Now that the standard CBD syntax and semantics has been discussed we can add concepts to this in order to achieve an extended formalism which we will call dynamic structure Causal Block Diagram (DSCBD). Section 3.3 will introduce a visual syntax that will be used to represent the examples throughout this chapter. Section 3.4 will elaborate on the actual semantics of the extended formalism, this includes the addition of two new blocks allowing the structural change. The first one will detect when a signal crosses zero from below, generating an event when a certain condition is satisfied. The generated event will serve as a trigger for the structural change. The second block we need will contain the actual instructions for the changes that need to occur. We will call this block a structure block. Four operations are allowed inside this structure block. The first one being identification of existing blocks or connections based on their identifiers. This is necessary for the second operation to be possible which is the removal of constructs. The third operation that is allowed is the instantiation of new blocks, which have earlier been declared within the structure block. Lastly,these new structures can be initialized using values that have been generated during execution. Section 3.5 will give some insight in the implementation used to achieve a running prototype of the dynamic structure CBD formalism.

3.1 Related Work

Different approaches can be taken to add the semantic constructs needed to support dynamic structures to a formalism that originally does not allow this type of behaviour. It is possible to use a second formalism that does allow this type of dynamic behaviour and interleave it with the original formalism. The addition of new constructs can still be needed as the semantic glue between the two individual formalisms. Statecharts and Class diagrams have been combined into a dynamic system formalism, SCCD, that has a proposed xml standard to go with it [13]. CBDs have also been incorporated in hybrid systems with timed finite state automata yet they do not support the type of dynamic expressiveness we are looking for [8]. they do however give an indication on how to glue two formalisms together and the additional constructs needed for this.

Alternatively one could provide a structured way in order to achieve dynamic structures and add this to the formalism. Similar methods have been used in DEVS [17][14] to allow the addition of dynamic structures (DSDEVS) [3] [2]. A model executive is defined for the structural change, this is basicaly a modified model that contains some structure function to formalise structural change. Efficient implementations for the DSDEVS formalism have already been accomplished, proving this is a method worth considering [9]. Another formalism has been introduced by Barros using the same method to provide dynamic structures namely heterogeneous flow systems [4][5]. A great example of this method over building a hybrid system is the fact that we can stay very close to the semantics of the original formalism since the model executive is literally just an extension of the original model. For this reason this method will be employed on causal block diagrams.

3.2 Abstract syntax

The abstract syntax of the dynamic structure CBD formalism will be very similar to the one of a standard CBD. As displayed in Figure 3.1 only two blocks need to be added compared to the original. The first one is the when crossing zero from below (WCZFB) block that provides the trigger for a structural change. The second block is the structure block, that provides the actual structural change trough a structure function. As you can see it can contain blocks and cbds. The reasons behind it will be discussed in Section 3.4.

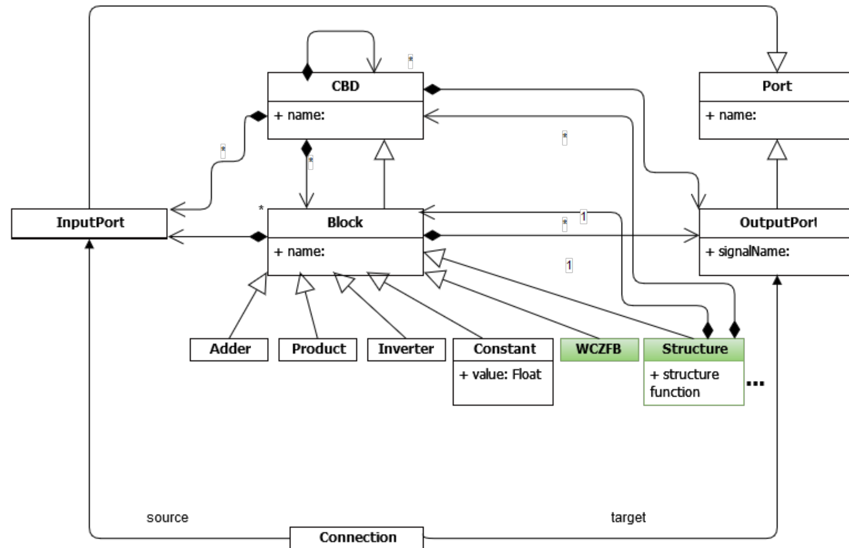

Figure 3.1: The abstract syntax of a standard CBD adapted from [8]

3.3 Visual syntax

Next to the xml representation a visual syntax can greatly aid the user in understanding the dynamic-CBD formalism. While it might not always be the most efficient way of representing a model, it gives a more clear overview of a model. The xml representation might still be readable but it requires a lot of analysis unlike the visual syntax. For readability we would like to keep the visual models as compact as possible while still being complete and able to represent all possible xml models. We will argue about different visual representations and finally decide on the one that will be consistently used for the remainder of the models.

3.3.1 Explicit representation

What actually happens when a structural change occurs is a transformation from one model to another one. This is inherently connected to matching the part of the model where the change occurs, when we consider a rule based transformation this is displayed on the left hand side. On the right hand side we specify the change that happens in the previously matched construct. Being this explicit leads to a number of possibilities one being graph rewriting. In the current implementation we always match specific blocks based on their identifier, which is the name of the block. In this visual syntax a more general approach could be taken where patterns are matched instead of these specific blocks. Every time this pattern occurs it is transformed in the same way. This feature is useful for expanding the current implementation of the formalism.

Figure3.2 gives an example of this representation. This example will be used throughout this section to compare the different visual representation. As u can see the CBD consists of one adder block that calculates the sum of two constants. On the grey box we see the definition of the rule with on the left hand side the current structure with the adder block a. These blocks are matched and transformed to the right hand side where the adder block is replaced by a product block. The constant blocks remain unchanged.

3.3.2 Implicit representation

In the implicit representation the simulated model will be interleaved with dynamic constructs leading to a more compact notation. Constructs to be removed from the current structure are

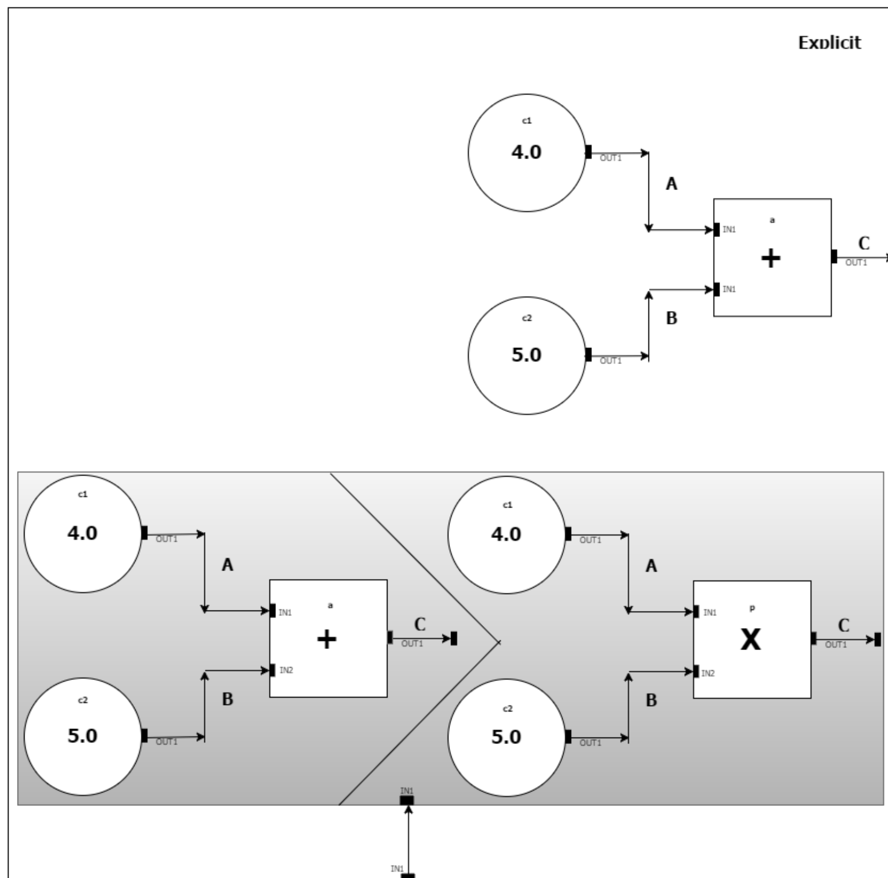

Figure 3.2: Explicit visual syntax for simple model

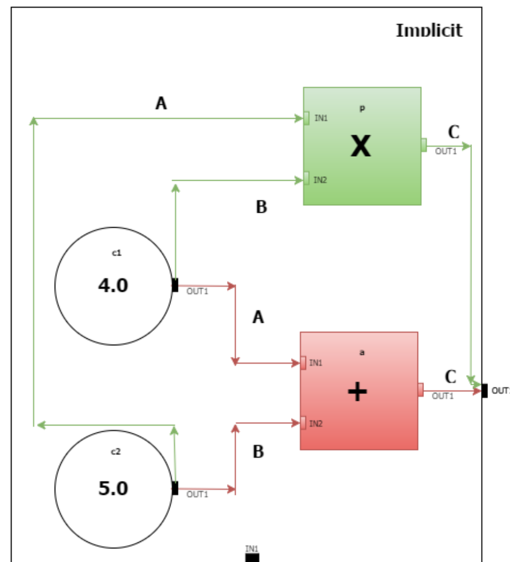

Figure 3.3: Implicit visual syntax for simple model

displayed in red while elements that are added are displayed in green. The use of this model might seem way more appealing at first because less drawing and detailing is needed for what might seem an equally expressive visual syntax. This however is not completely true. Since we are working with specific structures in the model we use the option to do pattern matching. While this is not supported in the current implementation it is something to keep in mild. Also expressing higher order dynamic structures is not possible in in this visual syntax. On top of that reinitialisation would not be supported without changing the semantics of the constant block by allowing input ports to provide the new values.

Figure4.1 shows the representation of our running example. As we can see right ahead, compactness has been greatly promoted. Yet this compactness has had an obvious cost, readability has somewhat deteriorated since it is harder to make a distinction between the current model and the one we will become after the structural change. Also the structural change must be triggered trough a value that is passed to the containing CBD.

3.3.3 Hybrid representation

Both an implicit and explicit representation have advantages and disadvantages. Ideally we would have a model that maintains the expressiveness and separation of structures of the explicit representation while at the same time remaining as compact as the implicit representation. Unfortunately this is not possible since increasing the explicitness will decrease compactness of the model since this requires some level of redundancy for matching the constructs that we want to structurally change followed by the changes that we want to apply as is the case in a rule based transformation. Separating these pre and post conditions is something that might not be desirable in our situation since we want to keep the notation as compact as possible. This is where the implicit representation comes in. Instead of first identifying the structures of interest followed by the modifications we can do this in one step. There is still a marching process, the same as the explicit representation, but structural changes will be specified using a color code. Red elements will be removed form the model while green elements will be added. This is completely analogue to the implicit representation.

Figure3.4 shows the adaption of the running example. As you can see he have embedded within the simulated CBD is a gray box containing the implicit representation displayed Figure4.1. A

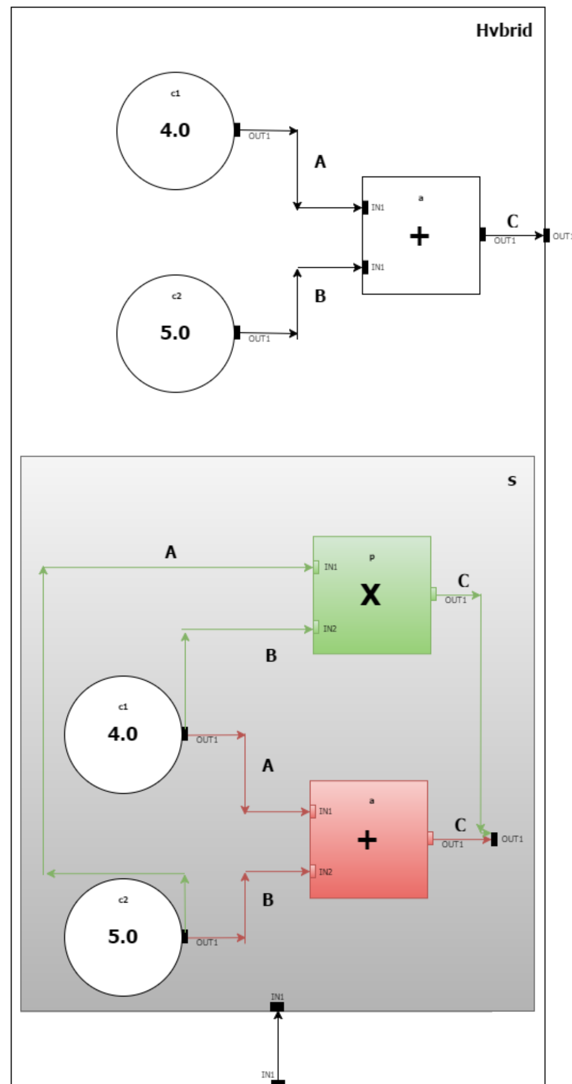

Figure 3.4: Hybrid visual syntax for simple model

clear distinction is made between the simulated model and the structural changes. The trigger for this structural change is simply passed trough the input port of the grey box, the same way as i the explicit representation.

3.3.4 Completeness of the representations

The dynamic structure CBD formalism allows for a set of new concepts that are added on top of standard CBDs. In order for the visual syntax to be complete it must be possible to represent all models that can be created within the formalism. This is only true when the complete set of concepts is supported. As displayed in Table 3.1 the implicit representation is not complete because of its setbacks. It can not represent reinitialisation without change of the semantics of the Constant Block. Also higher order structural change and pattern matching is not possible because of the in-model nature of this representation. Both the hybrid and explicit representations turn out to be complete since they support all concepts.

Table 3.1: Comparison of the different visual syntax's

	Implicit	Explicit	Hybrid
Removing connections	X	X	X
Adding connections	X	X	X
Removing Blocks	X	X	X
Adding Blocks	X	X	X
Reinitialising new structures	-	X	X
Higher order structural change	-	X	X
Pattern matching	-	X	X

3.3.5 Deciding upon a final representation

For the sake of this study we are going to use a hybrid representation. The implicit representation might be the most space efficient but it does leave more room for confusion. It mixes blocks of the currently simulated CBD with blocks that will be created by the structural change causing the user to lose sight of the simulated model. On top of that it is not complete. While it might suffice for simple models used to explain the concepts, in more complex ones such as our case study, the bouncing ball model, discussed in Chapter 4 it's shortcomings like the inability to express higher order structural change would become clear.

The explicit representation might be very clear with its left and right hand side rules, it will lead to models that are often very large even for fairly simple ones. Most of the models will be included this document for visual reference so models that are to big to fit on one page are highly undesirable. In order to tackle this problem the hybrid representation has been introduced, combining the strong points of both the implicit and explicit representation, leading to a visual syntax that is both complete and more compact than the original explicit representation.

3.4 Dynamic structure CBD semantics

For a formalism to allow dynamic structure some way has to be provided to change the model during simulation. In CBDs this would mean finding some way to manipulate the blocks and connections between blocks. This chapter will introduce a way for the CBD formalism to generate some kind of events triggering the structural change and the block responsible for the actual structural change. This new entity will be called a structure block and will allow the identification, removal, addition and reinitialisation of structures. Some interesting features such as higher order structural change will be discussed as well.

3.4.1 Events triggering a structural change

As we discussed in Section 2.2, a CBD will generate a discrete signal over time. This however is not ideal as a trigger for a structural change because we want the structural change to occur once an event occurs. The closest thing we have in the standard CBD implementations are the logical blocks, introduced in Section 2.4. The piecewise constant signal generated by these logical blocks must in some way be translated to an event. We want this event to occur only wen the signal crosses from a logical zero to a logical one. This can be enforced by a pre and post condition. in each timestep the pre condition will check if the previous value of the signal was zero. The post condition will check if the current value of the signal equals one. Only if both the pre and post condition are satisfied, a one signal is generated. The generated one signal can be viewed as an event which will only occur if a signal crosses zero from below. This is never the case in two consecutive timesteps. Figure **??** shows how the WCZFBBlock can be implemented using basic blocks. In some situations it might be desirable to generate an event after a fixed number of

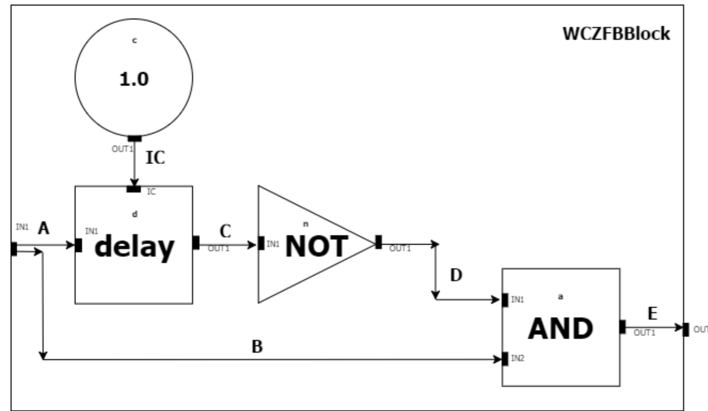

Figure 3.5: Implementation of WCZFB block using basic blocks

Table 3.2: Denotational semantics of event blocks

Name	Block	Semantics
WCZFB block		$if(prev(A) == 0)and(A == 1)) : 1$ $else : 0$
After block	After	$if(self.value == 0) : 1$ $else : -1$

timesteps have passed. Table 3.2 shows the semantics of this block, called the after block as well as the semantics of the WCZFB block.

3.4.2 The Structure Block

We earlier introduced the notation we will use for structural change. In this notation we allow the identification, removal, addition and reinitialisation of new structures. This section will give an overview of how these operations are supported. The new entity that will support this set of operations will be called a structure block and just like every object within a causal bock diagram it is a block itself. This new block will have a minimum of one input port trough which the event is passed. Additional input ports are used for reinitialisation as discussed in Section 3.4.5.

The scope of the structure block is limited to the structure in which it is embedded. In other words structural changes specified by this block will be limited to one CBD and will not be applied to the CBDs parent or child. This has the advantage that we can decide where the change will occur trough manipulating the structure.

3.4.3 Addition of constructs

Blocks or connections between blocks might need to be added during simulation. The first part of this is declaring the connections and blocks needed within the structure block. This happens in the same way you would a normal blocks and connections. This entire declaration is embedded within a structure function, which will be called every time a event is received by the structure

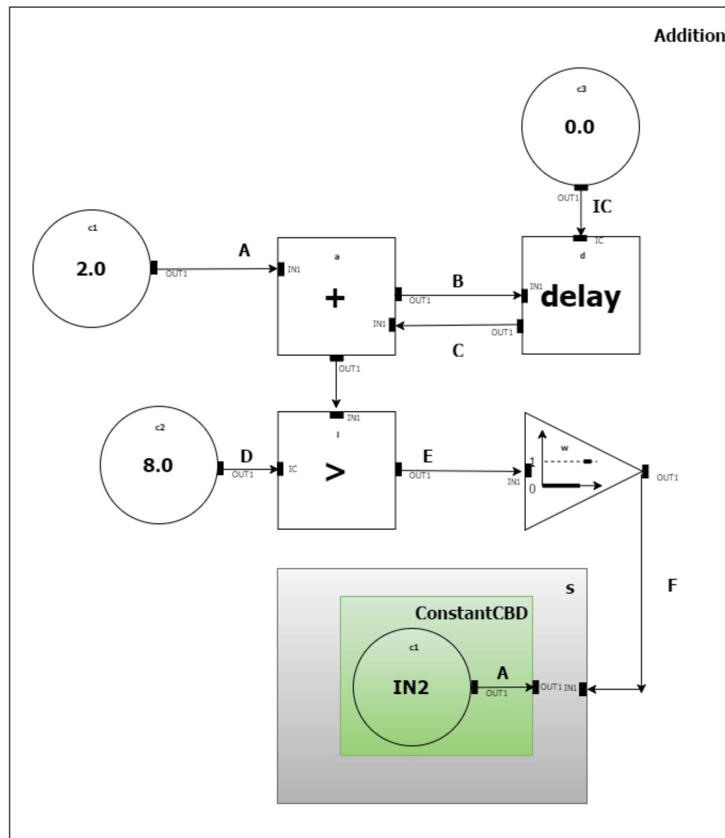

Figure 3.6: CBD illustrating the dynamic addition of structures

block. In other words, a structure block instantiates the blocks and connections in its containing CBD every time a event is received. This event must be passed to the structure block trough its first input port.

Figure 3.6 gives an example of a model where an instance ConstantCBD, displayed in green, is added to the CBD Addition every time an event is received by the structure block s.

3.4.4 Removal of constructs

In addition to adding new structures, removing existing ones must be possible to unwanted blocks and connections from the containing CBD. This is done using their identifier which in our case will be a unique name within that CBD. Whenever a block gets removed all the connections to this block get removed as well because in our definition of a connection it can only exist between a pair of ports. Once one port ceases to exist, the connection disappears. The removal statements are simply added to the structure function and the removal operation is executed together with the instantiation of the new structures when an event is received.

In Figure 3.7 one a structure block s with a red marker inside can be seen. Every time this block receives an event it removes its entire containing structure, in this case ConstantCBD. This is one way of removing a structure, the other one is displayed in Figure **??**. Blocks get detected using their identifier and marked in red for removal. In this example the constant block C1 gets removed and replaced by the structure in green.

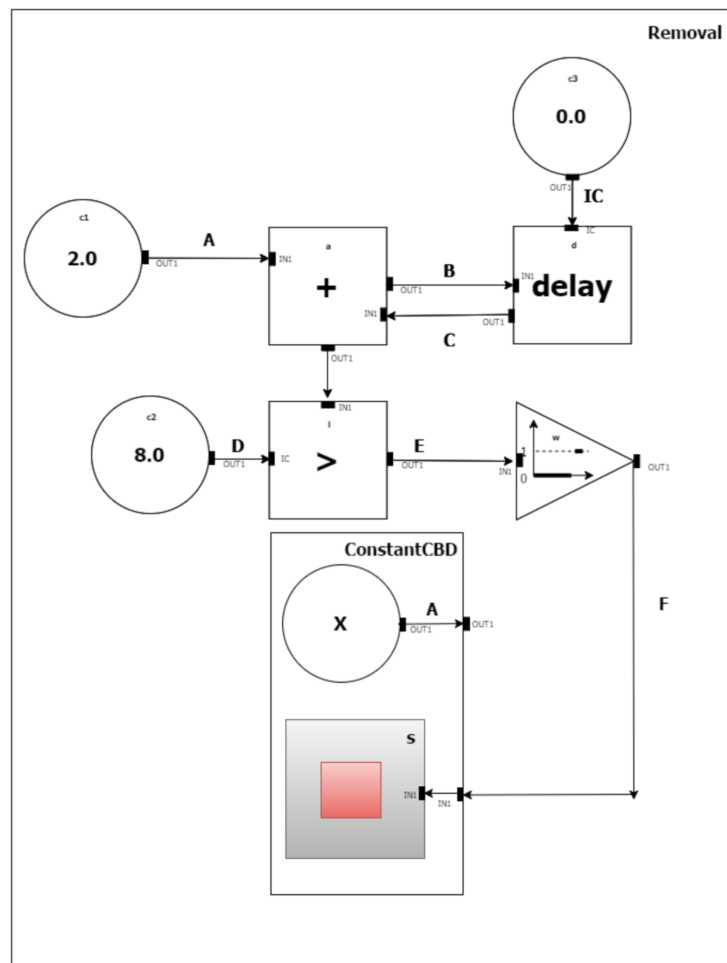

Figure 3.7: CBD illustrating the dynamic removal of structures

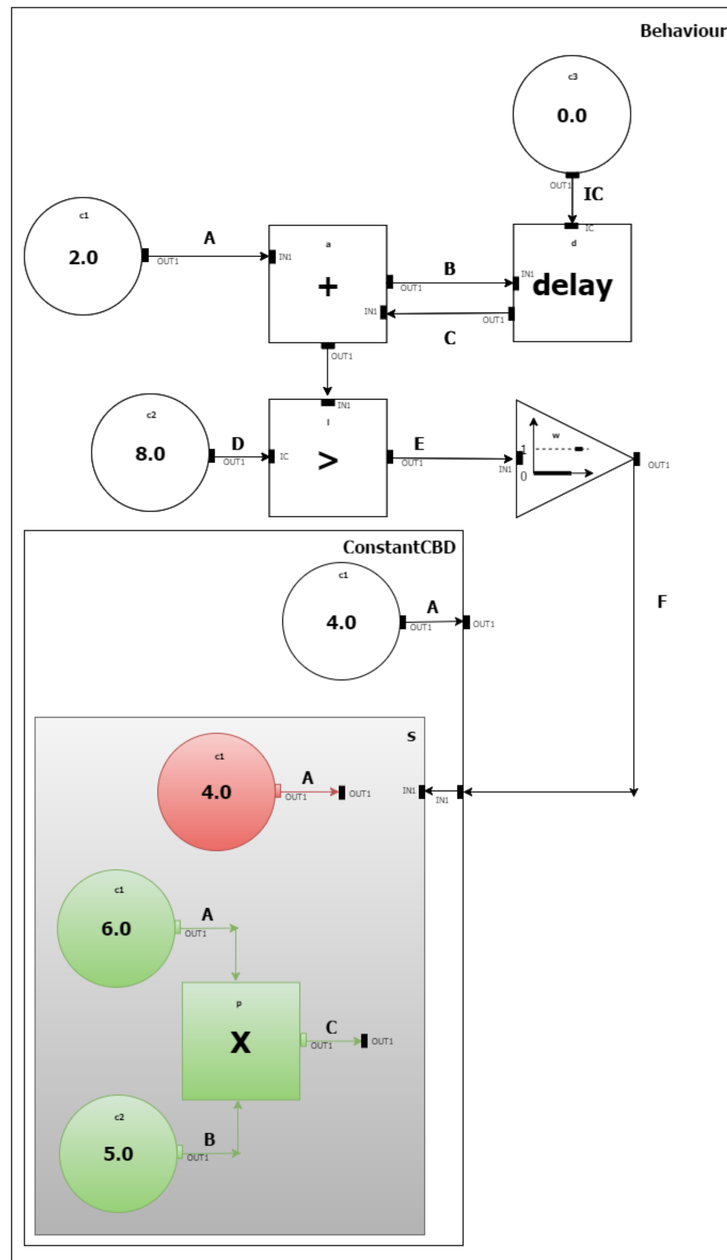

Figure 3.8: CBD illustrating the dynamic addition and removal of structures

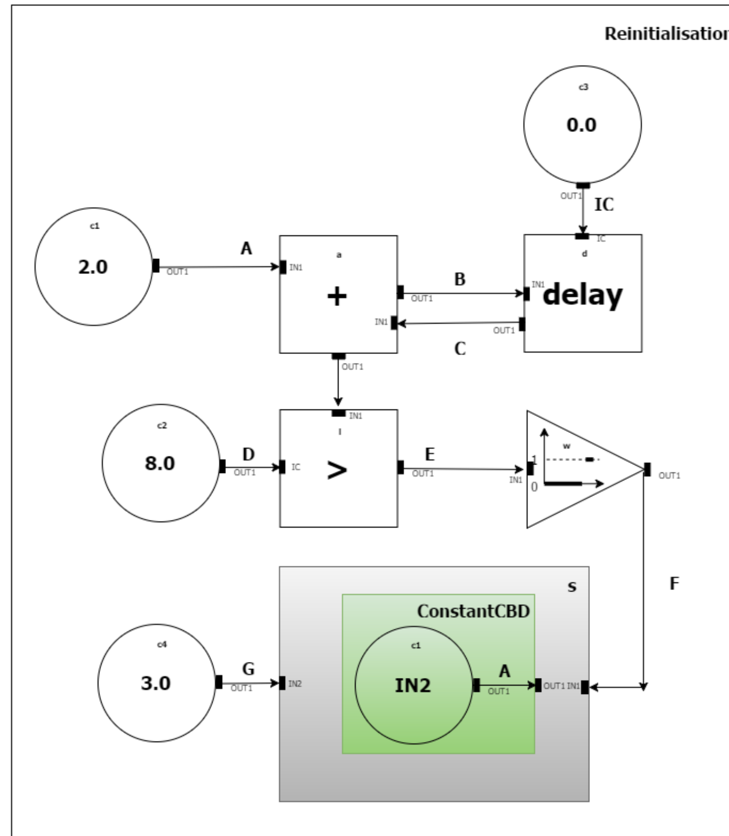

Figure 3.9: CBD illustrating the dynamic Reinitialisation of structures

3.4.5 Reinitialisation

Sometimes the initial values of a dynamically created structure must depend on certain output values that have been generated during execution. In case of CBD's initial values are always represented by constant blocks, meaning we have to link the value of a constant block to a certain input value. To achieve this we mus create additional input ports for the structure block. Every input port except for the first will be linked to some constant block value. This does not happen directly but the input signal is simply passed as an argument to the structure function which will use the arguments for proper instantiation of the constant blocks.

As you can see in the example in Figure 3.9 the structure block s has two input ports. IN1 gets used for the event as usual and IN2 gets inserted into the constant block c1. So after an event occurs, the constant block c1 within ConstantCBD gets instantiated with value 3.0.

3.4.6 Higher order structural change

When defining a structure block it might be of interest for the developer to allow the creation of additional structure blocks within this definition. This kind of recursive definition can have great advantages in certain types of models. Let us for example define a model of a rocket. Initially this can be represented by a normal causal block diagram but when it explodes the structure will change. The initial rocket will decompose into several smaller pieces of shrapnel that will all have their individual behaviour. As you see no recursive definitions needed so far, the structure blocks simply introduces Causal block diagrams that represent the behaviour of the pieces of shrapnel to the original model. But what if the smallest fragments would burn in the atmosphere after

some amount of time that is directly correlated to their size. An additional structure block would be needed to satisfied this requirement. The only way to do this in the dynamic structure CBD formalism is by creating a structure block that will trigger the removal of the shrapnel CBD after some time. The CBD representing the shrapnel has already been dynamically created by a structure block. The further dynamic behaviour must be specified within this structure block hence the need for recursion.

The same can be said for our case study: the bouncing ball model. When introducing a new ball to the structure, this ball will have a limited lifetime. A lifetime that is determined by some pre defied rule that can be either behaviour related or time bound. Just like the shrapnel of the rocket, the ball will disappear after some time which will induce the need for a recursive structure block definition. The case study will be discussed in depth in the next section.

3.4.7 Identifying dynamically created structures

When instantiating a dynamic structure it will be named uniquely by means of appending an integer number to a name that is provided in the declaration of the model. This mechanism gives us the opportunity to create an arbitrary number of equivalent structures without ever having to worry about redundancy in the name giving process. The newly added structure can however be subjected to CRUD operations again by the containing CBD which leads us to the problem of identification. Using the name for identification is impossible since the naming process takes place at run time when the structure is instantiated by its containing structure block. The way to handle this is by doing any operation on the dynamically created structure from within. Lets say we would want to remove the entire structure after some vent occurs. In a situation where the structures name is known before execution, a structure block containing the remove statement with the specified name could simply take care of this. When this name is not known the signal must be tunneled to the inside of the structure. An additional input port must be provided trough which the event passes. The event is then fed into the corresponding structure block that, instead of a remove tag with the name specified, contains an empty remove statement. This specifies the removal of the entire containing structure.

3.4.8 Adapted operational semantics

In order for the structural changes to be applied to a CDB we need the original hierarchical structure. Since we earlier defined a syntactical flattening operation, the inverse operation needs to precede the structural changes. Algorithm 3 shows the changes needed to the main simulation algorithm. After the needed structural changes are applied local time is increased and simulation is continued. Because of the need for unflattening in each time step, the flattening is no longer in pre processing step but rather in the main simulation loop.

Algorithm 3: Operational semantics dynamic structure CBD

```
   Data: cbd
   Result: Behaviour trace
1  logicalTime = 0;
2  while not end_condition do
3  |  flatcbd = FLATTEN(cbd);
4  |  schedule = LOOPDETECT(DEPGRAPH(flatcbd)) structureBlocks =
   |    COLLECTSTRUCTBLOCKS(schedule);
5  |  for block in schedule do
6  |  |_ flatcbd = COMPUTE(block, flatcbd)
7  |  for block in structureBlocks do
8  |  |_ cbd = COMPUTE(block, cbd)
9  |_ logicalTime = logicalTime + δt
```

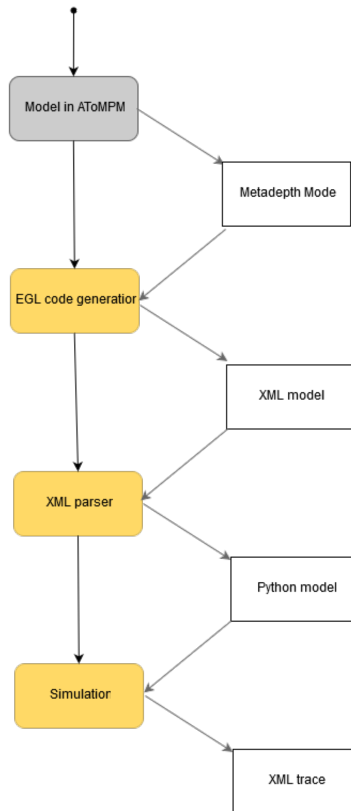

Figure 3.10: CBD illustrating the dynamic Reinitialisation of structures

3.5 Implementation

As a proof of concept a prototype simulator was developed in python. Figure 3.10 shows the steps that are required to simulate a visual model. A user can define a visual model using the web based modeling provided by ATomPM. Using the Epslon Generation Language (EGL), this model can be exported to a structured textual model in xml. An xml parser will use this textual model to generate the required python code needed to start the simulation.

3.5.1 Python implementation

The actual simulator used for the standard CBDs described in Section 2 has been extended to support the new blocks discussed in Section 3.4. The WCZFB block has simply been implemented using the available blocks from the standard CBD as previously discussed. The structure block required a bit more changes, a structure function must be passed as an argument to make instantiation of the new blocks and connections possible. This function can use a number of arguments that assign specific values to the constant blocks inside. Every timestep the value of the first input port is checked. If this equals zero, nothing happens. If the incoming signal equals one on the other hand, the structure function is called with the structure blocks parent and the signal values of the other input ports as arguments.

For the main simulation loop changes are limited. The usual flattening step, loop detection and computation occurs just like it would for simulation in a standard CBD. At the end of the simulation step however all structure blocks are computed and structural changes are executed when needed. This step requires the model to be unflattend, in our prototype implementation this is simply done by name analysis of the blocks that will allow for us to reconstruct the unflattend

structure. Basically it is exactly the inverse operation of the flattening step, The flattening as well as the unflattening operations are merely syntactical. All the examples shown in the Figures in section 3.4 have been used for the automated tests used in the implementation.

3.5.2 XML parser

In the implementation an xml parser is used for the declaration of CBD models. This section explains why we made this choice and gives a concrete example of what a model could look like.

Implicitly creating structure function

A structure block needs a structure function containing the declaration of the new blocks and connections as well as the removal instructions. In order to make this implicit we allow a user to declare a CBD just like he would a standard CBD within a structure block. The only difference is that removal operations can also be specified using the reserved remove tags. The listing below shows a XML representation of the model that was earlier displayed in Figure 3.9

```
<CBD name = "StructureBlockReinitTest" num_output_ports = "1">
  <Block type = "DelayBlock" block_name = "d"/>
  <Block type = "AdderBlock" block_name = "a"/>
  <Block type = "ConstantBlock" block_name = "ic" value = "0.0"/>
  <Block type = "ConstantBlock" block_name = "ac" value = "2.0"/>
  <Block type = "ConstantBlock" block_name = "nv" value = "3.0"/>
  <Block type = "ConstantBlock" block_name = "cond" value = "8.0"/>
  <Block type = "WhenGreaterThanBlock" block_name = "g"/>
  <Connection from_block = "c1" to_block = "OUT1" input_port_name="IN1"
      output_port_name="OUT1"/>
  <Block type = "StructureBlock" block_name = "s" num_inputs = "2">
      <CBD name = "ConstantCBD" num_output_ports = "1">
      <Block type = "ConstantBlock" block_name = "c" value = "IN2"/>
    </CBD>
  </Block>
    <Connection from_block = "ic" to_block = "d" input_port_name="IC"
        output_port_name="OUT1"/>
  <Connection from_block = "a" to_block = "d" input_port_name="IN1"
      output_port_name="OUT1"/>
  <Connection from_block = "d" to_block = "a" input_port_name="IN1"
      output_port_name="OUT1"/>
    <Connection from_block = "ac" to_block = "a" input_port_name="IN2"
        output_port_name="OUT1"/>
  <Connection from_block = "cond" to_block = "g" input_port_name="IC"
      output_port_name="OUT1"/>
  <Connection from_block = "a" to_block = "g" input_port_name="IN1"
      output_port_name="OUT1"/>
  <Connection from_block = "g" to_block = "s" input_port_name="IN1"
      output_port_name="OUT1"/>
  <Connection from_block = "nv" to_block = "s" input_port_name="IN2"
      output_port_name="OUT1"/>
    <Connection from_block = "a" to_block = "OUT1" input_port_name="IN1"
        output_port_name="OUT1"/>
</CBD>
```

Conversion to canonical form

Since XML is completely made up out of tags and their containing attributes, it is easy to read a representation of a CBD into memory and transform it into a form that is correct while at the same time consistent. WC3 introduced standards for XML canonisation [16] which is employed by the lxml module we used which allows a user to use an in memory tree representation of the XML structure to write its corresponding canonical form [1]. This however does not take into account tag order. To ensure this, each CBD that is written tot an XML file is processed in a particular order. First all block declarations including structures blocks and child CBDs are added to their patent in alphabetical order only when this is done connections are added in alphabetical order as well. Not only does this guarantee a consistent order, it also ensures proper execution since the operation that connects individual blocks can never refer to blocks that were only later defined.

3.5.3 Visual modeling environment

AToMPM [12] is a web based modeling environment that allows us to create a visual editor for the dynamic structure CBD formalism. The first step in doing so is creating an abstract syntax. We previously defined this in Section 3.2. Based on the abstract syntax a concrete visual syntax can be defined, this has been extensively explored in Section 3.3. When a model is created it can be exported to metadepth. Since metadepth has support for the epsilon generation language (EGL), the metadepth model can be used to generate the corresponding xml model [11]. Figure 3.11 shows an example of an implemented visual model in AToMPM.

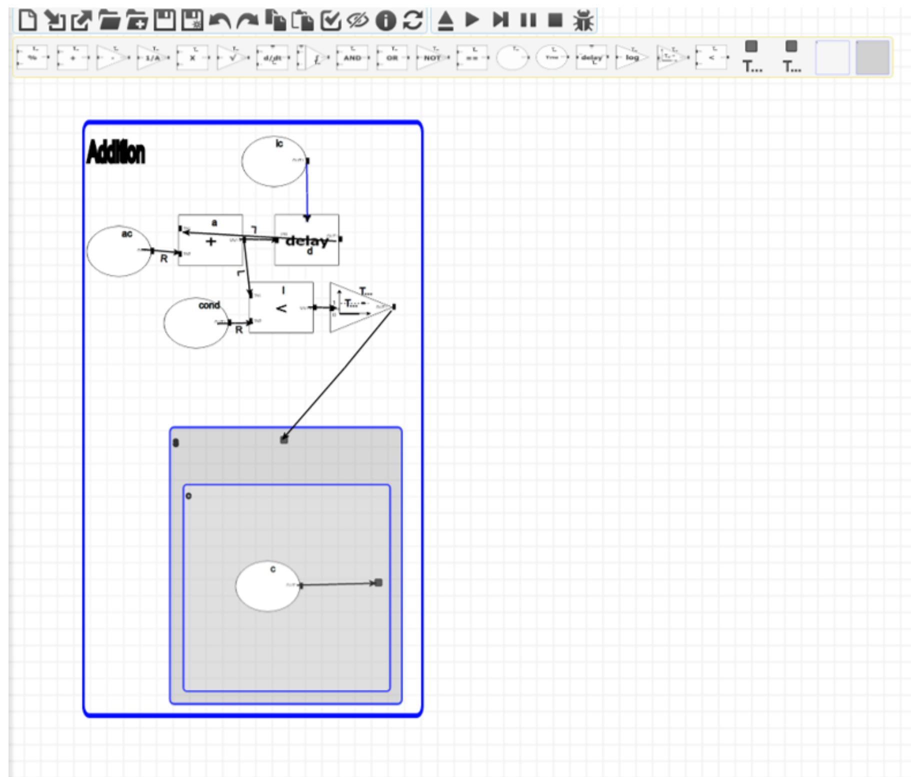

Figure 3.11: Visual model in AToMPM

3.5.4 Debugging

Debugging of standard causal block diagrams is fairly straight forward. Unit tests can be written to test core functionality. To get a clear overview of more complex models a trace of the computed values can be used. Dynamic structure CBDs however introduce some additional complexity that will be discussed in this section.

3.5.5 Unit tests

3.5.6 Standard CBD

It is most appropriate to start the testing at the lowest level which are the basic blocks. Simple tests can be written to assure a certain input generates the correct output. We simply check if the block performs the correct operation.

Once we know all the blocks have the intended behaviour, more complex cases can be tested starting with models without algebraic loops. This will put the implementation of the topological sort algorithm to the test. Models with an algebraic loop will test the strong component algorithm and the Gaussian solver that resolves the loops. Ofcourse a CBD can be hierarchical but for computation it will be (syntactically) flattened. This can be tested with hierarchical models of varying depths.

3.5.7 dynamic structure CBD

The same tests as the ones needed for a standard CBD formalism remain valid for dynamic structure CBD formalism. However some additional tests are needed because the signals can change during simulation. To fully test the implementation fully we must test all the features

including removal and creation of structures and initialisation of new structures. We must check if the structural change occurs at the right time and and if the right blocks get instantiated. After this is done we must check is simulation is resumed correctly by checking the new models.

3.5.8 Traceability

To verify more complex models we must be able to display simulation steps and results in a structured way. This can be achieved by a trace of the values generated by the composing blocks. To do this in a structured way we do this using xml. In each iteration we display the generated values and the new blocks that get instantiated by the structure blocks. The listing below shows the computed values at the first simulation step by the model shown in Figure 3.6. As you can see no new blocks are instantiated since this happened before starting simulation.

```
<TimeStep iteration = 0>
  <Output type = "InverterBlock" block_name = "ic" value = 0.0/>
  <Output type = "DelayBlock" block_name = "d" value = 0.0/>
  <Output type = "InverterBlock" block_name = "ac" value = 2.0/>
  <Output type = "AdderBlock" block_name = "a" value = 2.0/>
  <Output type = "InverterBlock" block_name = "cond" value = 8.0/>
  <Output type = "GreaterThanBlock" block_name = "g" value = -1/>
  <Output type = "WireBlock" block_name = "w.IN1" value = -1/>
  <Output type = "InverterBlock" block_name = "w.delayic" value = -1/>
  <Output type = "DelayBlock" block_name = "w.d1" value = -1/>
  <Output type = "NotBlock" block_name = "w.n" value = 1/>
  <Output type = "AndBlock" block_name "w.a" value = -1/>
  <Output type = "WireBlock" block_name = "w.OUT1" value = -1/>
  <Output type = "WireBlock" block_name = "OUT1" value = 2.0/>
<TimeStep />
```

In time step four shown in the listing below the condition for the structural change is satisfied which means some new blocks are instantiated. The instantiation is displayed in the same way as you would declare the block in xml representation. Only difference being that there is a time$_{o}ffsetthatisusedtoadjustthesimulationtimefortheseblocks$.

```
<TimeStep iteration = 4>
  <Output type = "InverterBlock" block_name = "ic" value = 0.0/>
  <Output type = "DelayBlock" block_name = "d" value = 8.0/>
  <Output type = "InverterBlock" block_name = "ac" value = 2.0/>
  <Output type = "AdderBlock" block_name = "a" value = 10.0/>
  <Output type = "InverterBlock" block_name = "cond" value = 8.0/>
  <Output type = "GreaterThanBlock" block_name = "g" value = 1/>
  <Output type = "WireBlock" block_name = "w.IN1" value = 1/>
  <Output type = "DelayBlock" block_name = "w.d1" value = -1/>
  <Output type = "NotBlock" block_name = "w.n" value = 1/>
  <Output type = "AndBlock" block_name = "w.a" value = 1/>
  <Output type = "WireBlock" block_name = "w.OUT1" value = 1/>
  <Output type = "WireBlock" block_name = "OUT1" value = 10.0/>
  <Output type = "InverterBlock" block_name = "w.delayic" value = -1/>
  <Block type = "OutputPortBlock" block_name = "OUT1" time_offset = 5/>
  <CBD name = "ConstantCBD0" time_offset = 5/>
  <Block type = "ConstantBlock" block_name = "c" value = 2.0 time_offset = 5/>
<TimeStep />
```

Chapter 4

Case Study: Elevator Model

This section will discuss our case study which will be an elevator model. We will start by implementing a bouncing ball in a two dimensional space, a case study that was previously used in testing a hybrid CBD system [8] as discussed in Section 4.1. To more thoroughly test the system dynamics we extended the case to an elevator model similar to a case study that was used to test the heterogeneous flow system formalism [2]. This will be discussed in Section 4.4. The elevator will be a bounded box that will move in only one dimension. In the implementation the elevator is embedded in a building of one floor but conceptually this remains the same for an arbitrary number of floors.The elevator will stop at each floor and when it stops doors will open immediately and balls will be able to enter and leave the elevator.Balls are generated at an arbitrary rate and balls leave every time the wall with the open door is "hit". This gives us a semi realistic scenario where balls are created and removed, an ideal test for the dynamic structure formalism. Figure 4.1 shows some stills of the implementation of the model. The elevator is represented by the red box and it can move up or down each timestep. The left side shows the elevator at a certain timestep with a single ball inside. On the image on the right side you can see a additional ball has entered the elevator. the door is displayed by the black on the left side. As you can see it has opened when the ball enters.

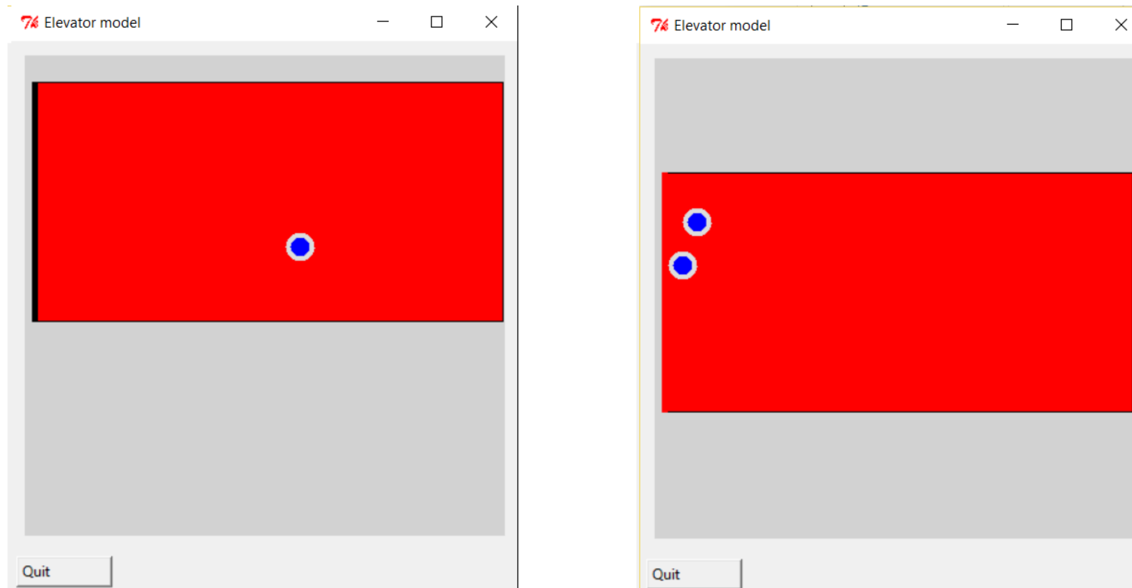

Figure 4.1: Stills of the implementation

4.1 Ball in a two dimensional space

To explore the expressiveness of the unaltered causal block diagram formalism we propose a simplified bouncing ball model which will consist of a ball with a constant velocity that will move around in a rectangular space. The ball will interact with the walls and in case of a collision the velocity of the ball will be changed accordingly. There are two scenario's that can occur since we are dealing with a rectangular space:

1. The ball collides with a horizontal wall: this will inverse the velocity in the Y dissection. A collision with the top wall will induce a downwards motion while a collision with the bottom wall will induce a upward motion.

2. The ball collides with a vertical wall: this will inverse the velocity in the X dissection. A collision with the left wall will induce a motion to the right while a collision with a bottom wall will induce a motion to the left.

For simplicity purposes a ball will first be assumed to be dimensionless. This means the only variable factor that has to be taken into account for collision detection is the position of the ball at a certain point in time. A single ball will require two distinct outputs to represent the x and y coordinates at a given time during simulation. The x and y positions will be calculated by their own dedicated CBD which is contained within the ball CBD. This kind of modular approach allows for easy augmentation to higher dimension models. Simply adding or removing a component and their associated input and output ports will delete or add a dimension to the model. The x and y positions will depend on the layout of the surrounding space which is constructed out of a top, bottom, left and right wall. This tells us the Ball CBD will have a need for four input ports each one representing a wall. The input representing the wall position will need to be redirected to the right position CBD. In a two dimensional model model this will mean connecting the bottom and top wall with the CBD calculating the y position. The left and right wall will be connected to the CBD calculating the x position. The walls will be represented as CBD's as well. although this might not have allot of added value in this version of the model, it allows for new constructs, like moving walls, to be incorporated more easily. The general construct of this design can be found in Figure 4.2. Here BallX and BallY are both instances of position.

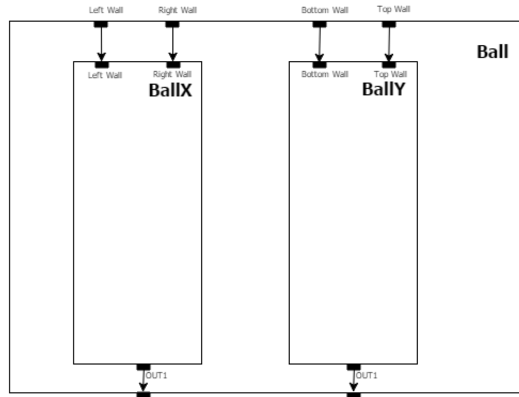

Figure 4.2: Overview of one ball

4.1.1 Discretized Position CBD

The position CBD in Figure 4.3 will be the actual functional part of the model that represents the ball. The goal will be to get the position of a ball in a certain dimension at a given point in time. To establish a position we first must first know the velocity of the ball. The Velocity will be provided by an other dedicated CBD witch is contained within the position CBD. Even tough the speed is constant in the first iteration of the model, the velocity is not. As discussed before, a collision with a wall will inverse the current velocity. To accommodate to this phenomenon two inputs are needed for the velocity CBD. Each of these inputs will get a signal that represents a Boolean value that is true every time the ball is within the bound of a certain wall and false when it is out of bounds. Considering a ball can collide with two walls in each dimension in a rectangular space each input port will correspond to a wall. For example if we take a look at the x position, collisions with the left and right wall are possible. Because we chose to represent True as one and False as minus one, the transformation of the velocity is simple. We just need to take the latest value of the velocity and multiply it by the incoming boolean values. When the ball has gone out of bounds, the direction of the velocity is inversed and the ball starts moving in the opposite direction. The latest value of the velocity is acquired from a delay block. The initial value of this the delay block is the starting velocity which is decided upon when before simulation is started.

Since we are first developing a discrete time model, we need to use a delay block to calculate the different positions. Each time step the value generated by the velocity CBD is multiplied with the passed time since the previous timestep δt. This results in the change in position. To get the new position all that remains is to add this difference to the previous value of the position which is the output of the delay block.

This approach does allow the ball to go outside the bounds of the containing space depending on the starting position, speed and timestep. If this is not desired the bounds of a model must be checked before the position at the start of each time step instead of at the end. We chose not to do this because this model gives a clear representation of the influence of varying the time step, start position and speed.

4.1.2 Continuous Position CBD with constant velocity

The continuous model with constant velocity will in reality simply be an alternate representation of the previously mentioned discrete time model. Instead of explicitly calculating the position by multiplying the velocity with the size of the timestep, we can now simply use an integrator to do the work for us. In Figure 4.4 the velocity block is simply connected to this integrator and its output is compared to the output positions to apply collision detection. Notice the velocity block

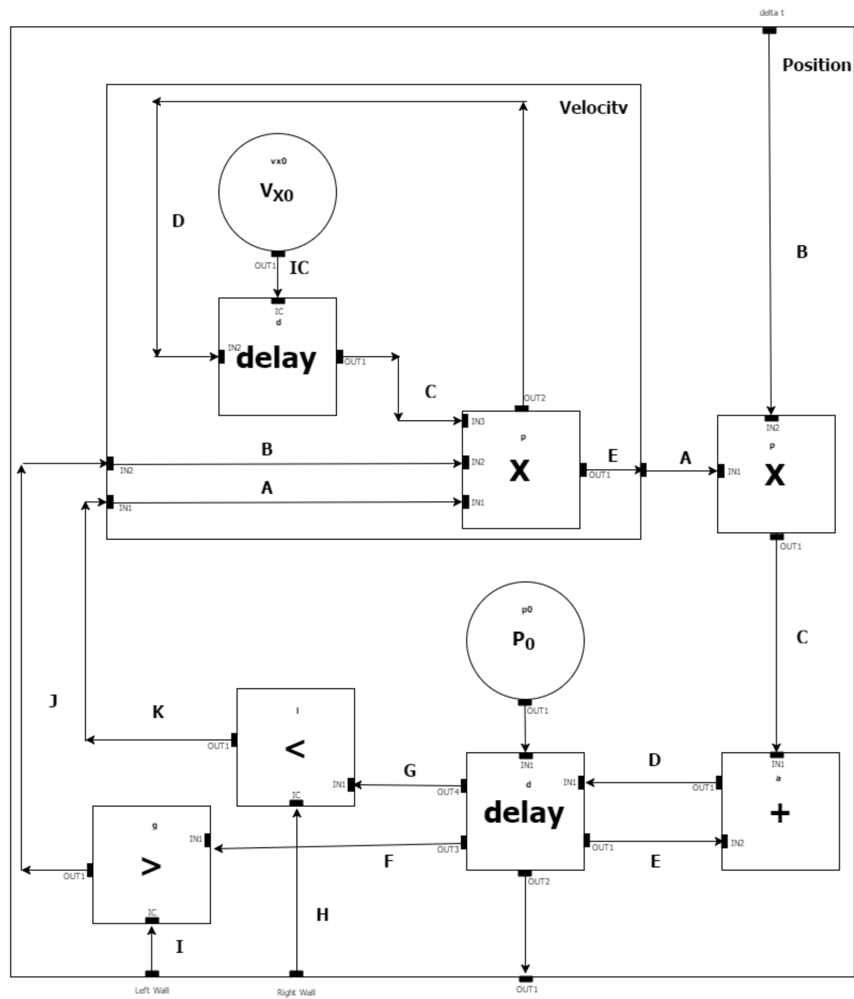

Figure 4.3: Discrete time position with constant velocity

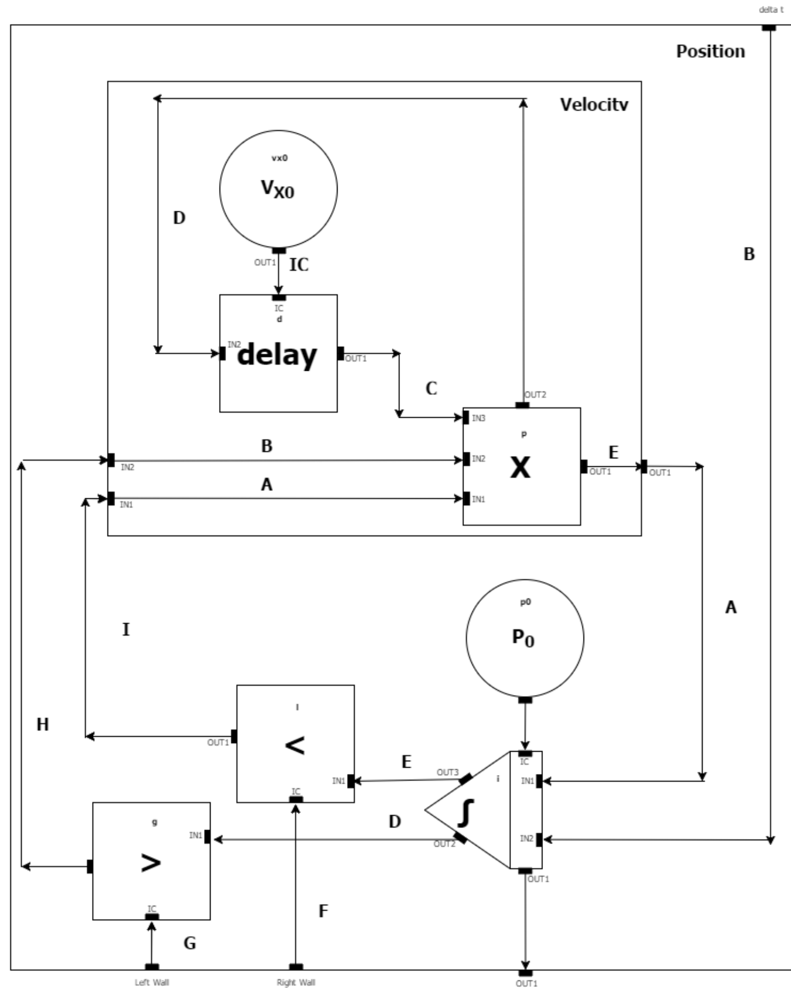

Figure 4.4: Continuous time position with constant velocity

remains completely unchanged since we are still working with a constant velocity. The initial condition of the integrator block will simply be a constant block representing the initial position in that dimension.

4.1.3 Continuous Position CBD with variable velocity

When we start varying the velocity it becomes an equation of time. As Newtonian physics dictates the rate of change in velocity is described by the acceleration a as seen in Equation 4.1. Hence the velocity can easily be calculated by taking the integral of the acceleration meaning we only have to change the velocity block in a way that it uses an integrator block. It would get a constant a representing the acceleration a as input and an other constant representing the starting velocity v_0 as initial condition. This new model is displayed in Figure 4.5

$$a = \frac{dv}{dt} \tag{4.1}$$

Up until now the change seems straightforward but collision detection adds some complexity that will force us to use the newly introduced structure blocks. This is the first time we deviate from the standard CBD formalism. Every time a ball collides with a wall it will cause us to invert the current velocity. To achieve this the integrator needs to be reset using the negation

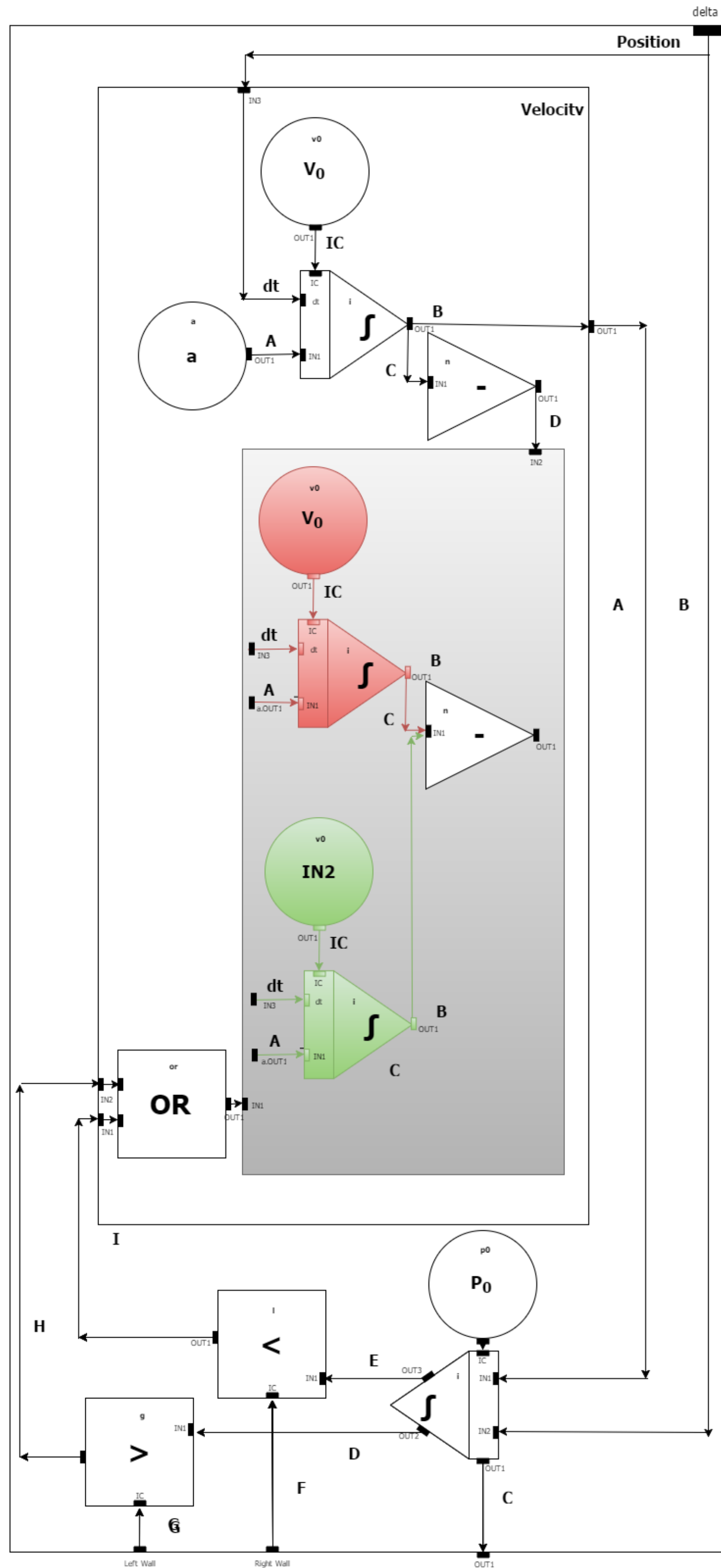

Figure 4.5: Position with variable velocity

of the last velocity value. To reset an integrator Block we first need to remove it together with it's corresponding initial condition. At the same time the new integrator block which will take the old ones place is instantiated together with a new initial velocity v_0 which is represented by a constant block. Since the value of v_0 will depend on the last output value of the velocity block, some reinitialisation is required. The negated velocity will be fed into the structure block using its second input port $IN2$. By using this name instead of a distinct value inside of the constant block, the value of the input port is used for reinitialisation.

The variable velocity model should only be used to calculate the y dimension of the ball since we are going for a realistic model. The X dimension will still employ the constant velocity model discussed in the previous section.

4.1.4 Variable timesteps

The model described in the previous section is not completely correct since we are using signals where events are required. More specifically when the collision detection is applied by comparing the current position of the ball to the position of the wall, a signal is generated. This signal is than passed to the velocity bock where the direction is inverted every time a one signal is received. when we apply fixed time steps this will not impose any problems because every time a ball passes the boundary it will be inside the boundary again in the next one. This is the case because we consider the case where the ball has a collision with the floor without loss of energy meaning the ball will travel equal distance before and after the collision.

In the case where the timestep can become smaller in case of a collision or a sufficient loss of energy occurs on impact, it is possible for the ball to satisfy the collision condition two timestamp in a row. This would lead to the ball oscillating because of two consecutive inversions of direction. This unwanted situation can be avoided by only generating one event in the situation where the signal crosses zero from below. Using The WCZFB Bock defined in the previous chapter we can easily enforce this. We just need to add these blocks between the conditions for the collision and the velocity block.

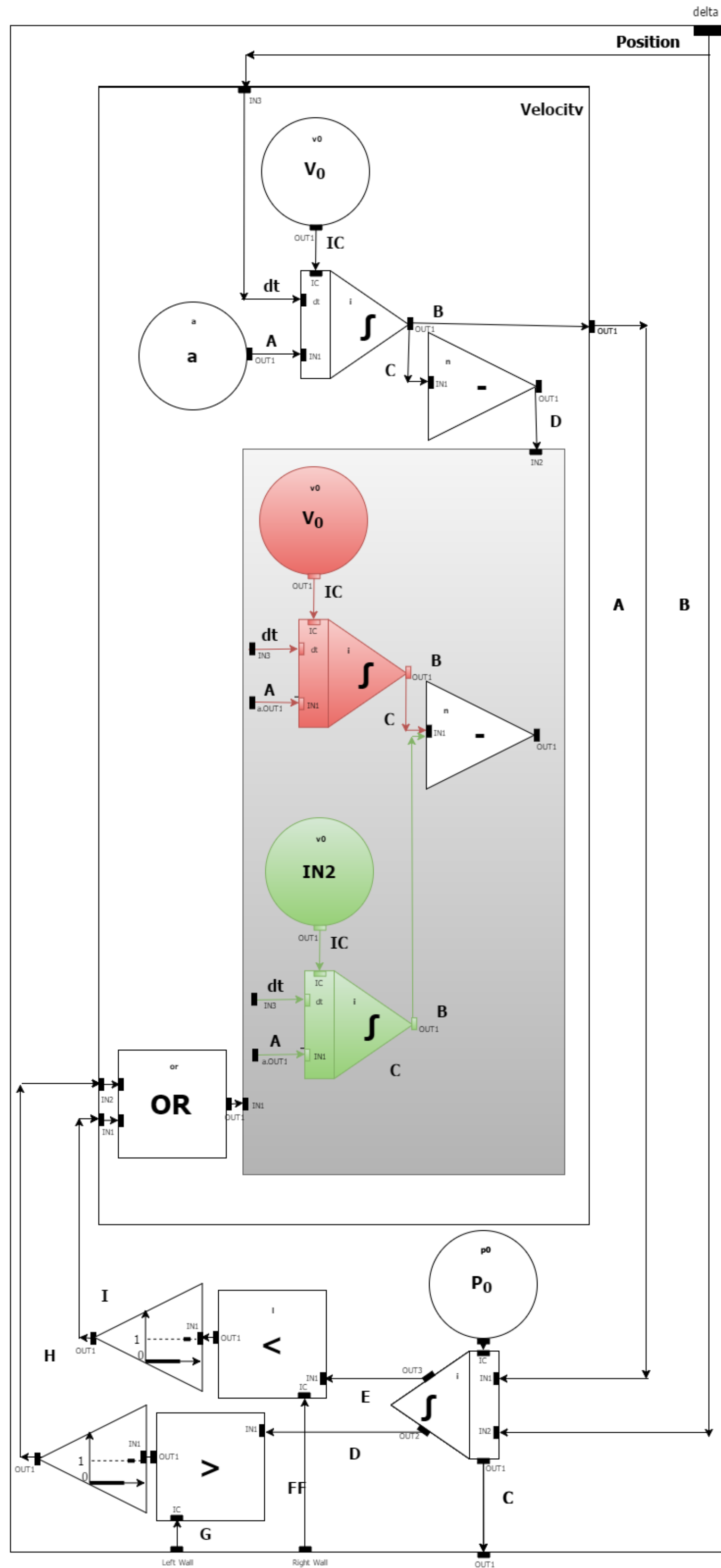

Figure 4.6: Position that allows variable timesteps

4.2 Dynamically removing Balls

Now that we have created a model that can represent a ball we need to be able to remove them. The first thing we need for this is the event that will trigger the removal of the respective ball. A separate CBD will generate this event based on some logical condition. This can be anything for example a counter that registers the number of hits on a specific wall and removes once a certain amount is registered.

In Figure **??** you can see the adjustments that need to be made to a ball, to be able to remove it after some time. Since balls can be dynamically created, the structure block that contains the instruction to actually remove the ball is embedded within that ball instance. This will ensure identification will never be a problem. The event generated by the previously described CBD is than tunneled to the inside of the ball and connected to the BallRemover. The remainder of the CBD will remain the same as we discussed in the previous Section.

4.3 Dynamically creating Balls

Adding balls happens in a very similar way to removing balls. again an event needs to be generated by some separate CBD that contains the logic for adding the balls. The event is passed to a structure block that creates the ball including the CBD that generates the removal event. In Figure 4.8 a scenario is displayed where an event is generated based on the positions of the already created balls. An example could be whenever the existing balls have hut the right wall a number of times, create a new ball. The event generated by this ball is then simply tunneled to BallAdder which contains the declaration of a ball.

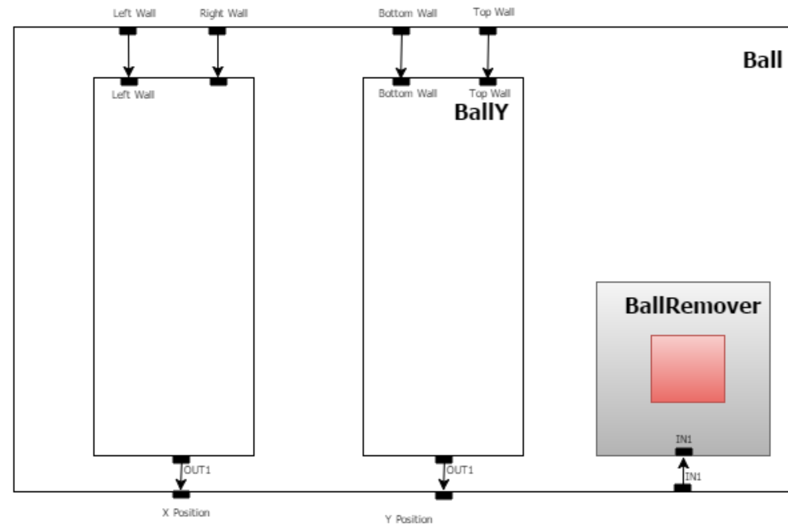

Figure 4.7: Overview of dynamically removed ball

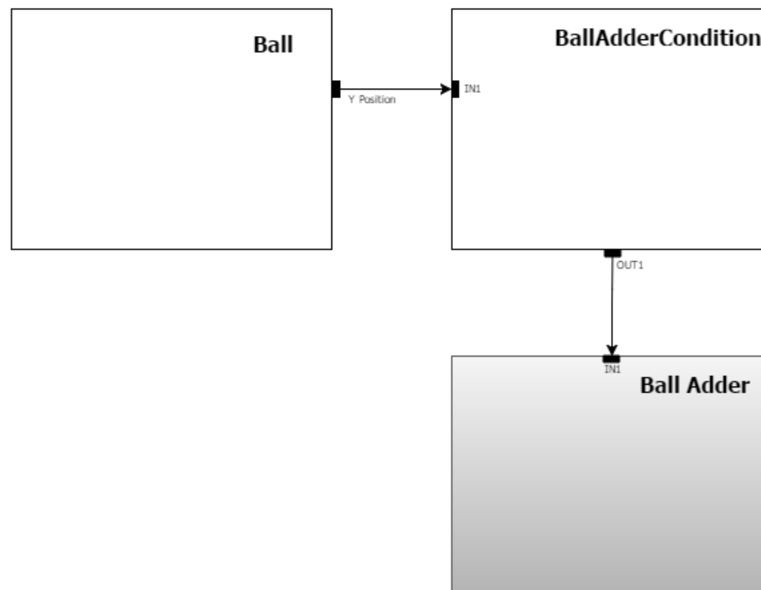

Figure 4.8: Overview of dynamically created ball

4.4 Elevator Model

We want the balls modeled in the previous sections to be created and removed in a more realistic manner. To achieve this we define an elevator that contains bouncing balls [2]. The elevator will move from floor to floor opening its door every time it comes to a halt. At this point it is possible for new balls to enter the elevator and existing balls to leave the elevator. Entering the elevator happens immediately when the door opens while leaving only happens when the door is open and a ball moves trough it.

In order for an elevator model to be created the top and bottom wall must be created. Both the top and bottom wall will have to move up and down, every time a the bound is encountered the direction is changed. This might seem very familiar because it is exactly the same as the CBD that calculates the position of the ball with constant velocity meaning that model can be completely reused. The only difference being that the input values are the top and bottom bounds of the respective top and bottom wall of the elevator.

4.4.1 Stopping at floors

To make the elevator stop some adjustments need to be made to the velocity block of the position CBD Figure 4.4. The new velocity block is displayed in Figure 4.9. A first major change that can be noticed is the addition of a condition fCheck which will compare the current position to the position of the respective floors. This will send an event to the structure block s1 which will disconnect the integrator block from the output and connect a zero constant block instead. This makes the elevator stop instantly. It will also activate a new structure block s2 after a fixed number of timesteps which will restore the original model again and make the elevator resume its movement.

The door will simply be represented by an equals block that checks if the velocity is zero. This will make the doors open instantaneously. Also this simulation is only run for the bottom wall since the top wall will simply be at a fixed length above the bottom wall.

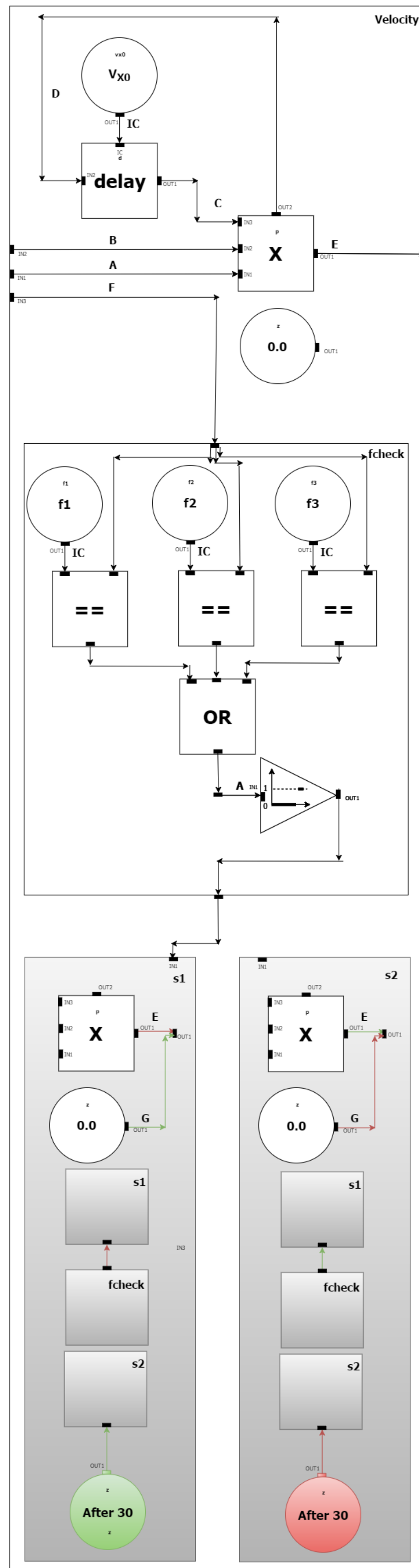

Figure 4.9: Variable velocity in elevator model

4.5 Execution plots

To give an illustration of how a possible execution would look, some execution plots where generated. Figure 4.10 shows the movement of the elevator, as you can see it starts at the top and moves down to two floors. At the first stop of the elevator, a ball gets created, meaning the plot of this ball only starts at this time, as can be observed in Figure 4.11. After the elevator stops at the second floor an additional ball gets created, this plot is displayed in Figure 4.12.

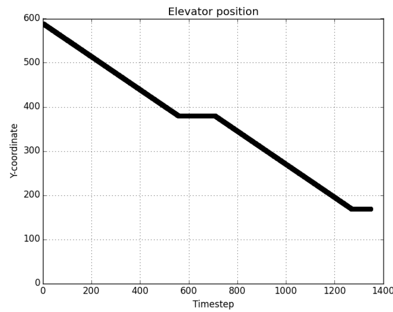

Figure 4.10: Y-position of the elevator

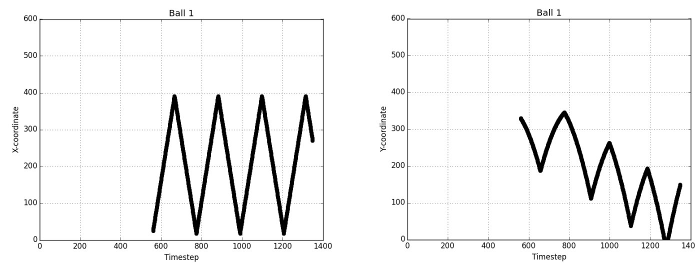

Figure 4.11: Position of ball 1

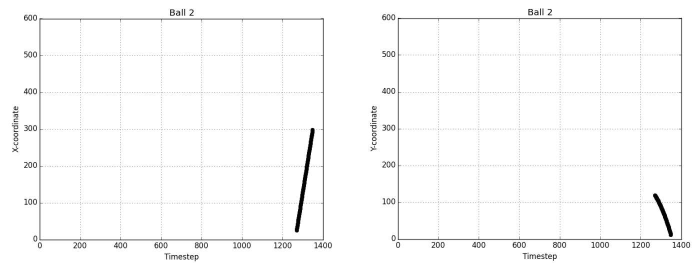

Figure 4.12: Position of ball 2

Chapter 5

Conclusions

We started off by giving a definition of the standard CBD formalism in Section 2 based on the syntax and semantics. In Chapter ?? we have boiled down the essence of dynamic structure to a two phase system where first there is a trigger needed to identify when the structural change needs to occur. This trigger will in our case be an event. Since causal block diagrams are not an event based formalism but rather discrete or democratized continuous some modifications where necessary to generate these events. To stay true to the nature of causal block diagrams we decided to implement this using the existing blocks. By simply checking if a signal crossed zero from below, a piecewise constant signal can be translated to an event.

An event is passed to a structure block for further processing. In order to define structural charge we needed to allow the following operations that were related to it:

- Identification of existing structures
- Instantiation of new structures
- Removal of existing structures
- Reinitialisation of new structures

All these individual operations are embedded within the structure function that is called by a structure block when it receives a one signal. Structures are identified based on a unique identifier which in our implementation is their name. A special remove operation can get rid of these identified structures and new structures can be declared just like they would be in standard CBD. values generated during simulation can be passed to the structure block via additional input ports to initialise the new structures. Simulation changes compared to standard CBDs because at the end of each time step the structure blocks are checked to determine whether structural change is needed. When this is the case, the model is transformed and simulation is resumed.

We put a prototype implementation of the dynamic structure CBD formalism to the test with an extensive case study, the elevator model described in Chapter 4. The elevator contains some balls that can bounce around freely and it moves from floor to floor in a building stopping at each floor. When the elevator stops, the door opens and balls can enter and leave the elevator. This case study is ideal to test a dynamic structure system because it contains a great number of variable elements such as the speed of the elevator, the number of balls and the behaviour of the balls.

Chapter 6

Future Work

Future work includes some additional research that could be done on performance of the new formalism since this was not the focus of this thesis as well as further increasing expressiveness. Performance could be possibly improved by not making a structure block instantiate each structure but rather running multiple simulations at the same time. Not only would it cut down simulation time but it would get rid of the overhead of instantiating these new structures. This would however come with some limitations such as the fact that no further change could be supported since this would disrupt the multiple simulations that are being run on that one instance. This optimisation technique however remains untested.

The dynamic structure CBD formalism could also be tested against hybrid formalisms in both performance and expressiveness to determine which method is best suitable to adapt existing formalisms. Expressiveness of dynamic structure CBD could also be further expanded by using the more general technique of graph rewriting inside the structure blocks. This allows us to match a great number of structures at ones without using their identifier. Whether this has any use outside of academic purposes also needs to be further researched.

The final part that is worth mentioning is scheduling. In the current implementation scheduling of structure block is done in a vary basic way where the blocks that do not induce change are triggered first and the other ones second. This could introduce some problems since one block might alter objects that are created by another block. An other situation is that a certain structure block alters the input values of a second structure block. These are both examples of situations where advanced scheduling is needed.

Bibliography

[1] *lxml*, 2016 (accessed August 17, 2016). 28

[2] Fernando J Barros. The dynamic structure discrete event system specification formalism. *Transactions of the Society for Computer Simulation International*, 13(1):35–46, 1996. 13, 31, 41

[3] Fernando J Barros. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(4):501–515, 1997. 13

[4] Fernando J Barros. Modeling and simulation of dynamic structure heterogeneous flow systems. *Simulation*, 78(1):18–27, 2002. 13

[5] Fernando J Barros. Dynamic structure multiparadigm modeling and simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 13(3):259–275, 2003. 13

[6] Ben Denckla and Pieter J Mosterman. Formalizing causal block diagrams for modeling a class of hybrid dynamic systems. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 4193–4198. IEEE, 2005. 3

[7] Mathlab Mathworks. Simulink. *Dynamic System Simulation for MATLAB, Version*, 7, 2008. 3

[8] Sadaf Mustafiz, Bruno Barroca, Claudio Gomes, and Hans Vangheluwe. Towards modular language design using language fragments: The hybrid systems case study. In *Information Technology: New Generations*, pages 785–797. Springer, 2016. ixix, ixix, 6, 12, 13, 15, 31

[9] Alexander Muzy and James J Nutaro. Algorithms for efficient implementations of the devs & dsdevs abstract simulators. In *1st Open International Conference on Modeling & Simulation (OICMS)*, pages 273–279, 2005. 13

[10] Ernesto Posse, Juan De Lara, and Hans Vangheluwe. Processing causal block diagrams with graph-grammars in atom3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*, pages 23–34, 2002. 3

[11] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. The epsilon generation language. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 1–16. Springer, 2008. 28

[12] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. Atompm: A web-based modeling environment. In *Demos/Posters/StudentResearch@ MoDELS*, pages 21–25. Citeseer, 2013. 28

[13] Simon Van Mierlo, Yentl Van, Bart Meyers Tendeloo, Joeri Exelmans, and Hans Vangheluwe. Sccd: Scxml extended with class diagrams. 13

[14] Hans Vangheluwe. The discrete event system specification (devs) formalism. *Course Notes, Course: Modeling and Simulation (COMP522A), McGill University, Montreal Canada*, 2001. 13

[15] Hans Vangheluwe, Joachim Denil, Sadaf Mustafiz, Daniel Riegelhaupt, and Simon Van Mierlo. Explicit modelling of a cbd experimentation environment. In *Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative*, page 13. Society for Computer Simulation International, 2014. 12

[16] W3C. *W3C recommendation Canonical XML*, 2001 (accessed August 17, 2016). 28

[17] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems.* Academic press, 2000. 13