Constraint Extension to Dataflow Network

Tsang Wing Yee

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

August, 2004

Abstract

Dataflow Network is a model that can specify flows of dynamic data with various operators running in parallel. The model can be specified and solved by a language called Lucid. To extend the functionality of Dataflow Network, we have extended it to a new model called *Extended Dataflow Network*, and we have developed its corresponding language *E-Lucid* to model and solve Extended Dataflow Networks with help of a CSP solver.

摘要

數據流網絡是一個可以用各樣平行運作操作符來表述動態資料流的模型。這模型可用 Lucid 語言來表述和求解。要擴展數據流網絡的功能，我們把它擴展作一個稱爲*延伸數據流網絡*的新模型，並爲它發展了 *E-Lucid* 語言。結合 CSP 求解程序，這語言可以模型化及解決延伸數據流網絡。

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The most common class of programming languages, such as C [28] and Java [15], is based on the "von Neumann" control flow model. The model assumes that a program is a series of instructions, each of which either specifies an operation along with memory locations of the operands or specifies transfer of control to another instruction unconditionally or when some conditions hold. On the contrary, dataflow languages are based on Dataflow Network. The model does not consider flow of control, but considers flow of data. It assumes that a program is a data-dependency network (or Dataflow Network) whose nodes denote operations and whose arcs denote dependencies between operations. The programmer pays his attention in defining the data of the problem rather than the execution of the program. In many situations, it is more simpler and more intuitive to express a problem with dataflow languages.

Dataflow Network[1] [34] has been the basis of several programming languages [21, 8, 17, 4, 16, 7, 22], among which Lucid is the first dataflow language [34]. There are various incarnations of Lucid, including pLucid [1, 12], the first Lucid implementation [34]; GLU [24, 23], a hybrid of Lucid and C; Ferd Lucid [11, 13] and Field Lucid [2], two extensions with array as the data structure; Indexical Lucid [25], an extension with user-defined dimensions and dimensional abstraction; and RLucid [33], an extensions with time stamps.

[1] We refer Dataflow Network to *pipeline dataflow network* [9] from now on

These languages are able to program dynamic problems by defining each variable (flow of data) with exactly one expression in terms of other variables. For example, we can express x, a variable, by the *statement* `x = y + 2 * z`, where y and z are two other variables.

However, none of these languages take the advantages of constraint-solving technique to express general constraints among the variables. The languages cannot define a variable on its own current value, so that a program cannot have the statement `x = x` or both the statements `x = 2 * y` and `y = x - 2`. The languages can only define a variable exactly once, so that a program cannot have both the statements `x = 2 * y` and `x = z + 2`. A statement can only relate a variable with an expression, so that a program cannot have the statement `x * y = y + z`.

In this thesis, we extend both Dataflow Network and Lucid so that the aforementioned limitations are removed. We propose a new model called *Extended Dataflow Network*, which is based on Dataflow Network with addition of *selection operators* and *assertion arcs*. Selection operators allow the definition of possible choices of values, while assertion arcs define constraints among variables. The corresponding language is *E-Lucid*, which extends Lucid with additional implementation of *selection operators* and *assertion variables*. They correspond to selection operators and assertion arcs of Extended Dataflow Network respectively. By making use of selection operators and assertion variables, we can express and reason dynamic problems with useful complex constraints.

For example, solving Fibonacci sequence is a well-known dynamic problem. The sequence *fib* is defined mathematically as follows:

$$fib_t = \begin{cases} t & \texttt{if } t = 0 \texttt{ or } t = 1 \\ fib_{t-1} + fib_{t-2} & \texttt{otherwise} \end{cases}$$

The problem can be coded in Lucid as:

```
fib = 0 fby 1 fby fib + next fib;
```

We can find the whole Fibonacci Sequence by giving the initial condition

that the first two numbers are 0 and 1 respectively. The sequence is 0, 1, 1, 2, 3, 5, 8, 13, and so on. However, if the initial condition is not given, the problem cannot be solved with the Lucid program. On the other hand, E-Lucid program can solve it by giving any two intermediate numbers, such as $fib_4 = 3$ and $fib_5 = 5$.

We enhance the pLucid interpreter to obtain a prototype E-Lucid interpreter. It solves an E-Lucid program by constructing and solving Constraint Satisfaction Problems (CSPs) [27] with the help of ILOG [18] as the external CSP solver.

The rest of the thesis is organized as follows. Chapter 2 introduces Dataflow Network and Lucid as background information, so that we can introduce Extended Dataflow Networks and E-Lucid with examples in Chapter 3. Chapter 4 presents the implementation of the E-Lucid interpreter. Chapter 5 points out the differences between E-Lucid and other models of dynamic problems. We conclude the thesis in Chapter 6 by summarizing our contributions and shedding light on possible directions of future research.

Chapter 2

Preliminaries

This chapter provides the theoretical background of this work. We present the basic definitions of *Constraint Satisfaction Problems* (CSPs) [27]. In addition, we present Dataflow Networks [6, 26], which can model flows of dynamic data with various operators running in parallel. Dataflow networks make the expressions of iterations and data flows natural and simple. Lucid [3] is a programming language to specify and execute dataflow networks.

2.1 Constraint Satisfaction Problems

A *constraint* [31] c can be considered as a predicate that maps to `true` or `false`. We define $\texttt{var}(c)$ as the set of variables involved in constraint c.

A *valuation* θ for a set V of variables is an assignment of values to the variables in V. Suppose $V = \{v_1, \ldots, v_n\}$ then θ is written as $\{v_1 \mapsto d_1, \ldots, v_n \mapsto d_n\}$, meaning that each v_i is assigned the value d_i.

A constraint c over variables V is given a value $\theta(c)$ over variables V. $\theta(c)$ is obtained by replacing each variable by its corresponding value. If $\texttt{var}(c) \subseteq V$ and $\theta(c)$ holds, θ *satisfies* c and is a *solution* of c. A constraint c is *satisfiable* if it has a solution. Otherwise it is *unsatisfiable*. Given a set of constraints C, interpreted as a conjunction, if θ is a solution of all $c \in C$, we abuse terminology by saying that θ *satisfies* C and is a *solution* of C. A set of constraints C is

satisfiable if it has a solution. Otherwise it is *unsatisfiable.*

A *constraint satisfaction problem*(CSP) [31] p is a tuple $\langle X, D, C\rangle$, where $X = \{x_1, \ldots, x_n\}$ is a finite set of variables, D is a finite or infinite set containing, for every $x \in X$, an associated domain D_x giving the set of possible values of x, and C is a finite set of constraints, where $\texttt{var}(c) \subseteq X$ for all $c \in C$. p is *satisfiable* if $C \wedge x_1 \in D_{x_1} \wedge \ldots \wedge x_n \in D_{x_n}$ has a solution. Otherwise it is *unsatisfiable.*

2.2 Dataflow Networks

Suppose there is a sequence of points, representing discrete equally spaced time points, indexed by $t \in \mathcal{Z} \geq 0$. The value x associating with each of these points is called a *datastream*, and the points are called *datons.* We use $x[t]$ to denote the $(t+1)^{th}$ daton of x, where t is the *index* of $x[t]$. A *valuation* θ over a set U of datons is an assignment of values to the datons. Suppose $U = \{u_1, \ldots, u_n\}$, θ is written as $\{u_1 \mapsto d_1, \ldots, u_n \mapsto d_n\}$ meaning that each u_i is assigned the value d_i. We use a number with a pair of double quotes to denote a daton which is assigned the value of the number. For example, "7" denotes a daton assigned the value 7. If $x[i] \mapsto a_i$ for all $i \in \mathcal{Z} \geq 0$, we can denote the values of the first $i+1$ datons of x by $x = \langle$ "a_0", "a_1", "a_2", ..., "a_i", ... $\rangle$, or simply $x = \langle a_0, a_1, a_2, ..., a_i, ...\rangle$. For example, x is $\langle 4, 1, 4, 7, 2, ...\rangle$ if we have $\{x[0] \mapsto 4, x[1] \mapsto 1, x[2] \mapsto 4, x[3] \mapsto 7, x[4] \mapsto 2\}$.

Each datastream has an infinite number of datons. Each daton can take value from its associated *daton domain*, which defines the value type of the datons. There are various daton domains, such as integer, real, boolean, or any other user-defined daton domains. All daton domains must include the special value `eod` which denotes the "end of data" and marks the end of a datastream. For example, if a datastream x has integer daton domain, then x is an integer datastream and $x[t]$ must be an integer or `eod` for all $t \in \mathcal{Z} \geq 0$.

If a daton $x[t]$ of a datastream x takes the value `eod`, $x[t']$ must take the value `eod` for all integers $t' > t$.

Constant datastreams are datastreams with identical datons, which are denoted with boldface hereafter. For example, all datons of the datastream **7** take the value 7, and all datons of the datastream **eod** take the value `eod`.

A *Dataflow Network* is represented by a directed graph with datastreams as arcs. The nodes represent dataflow operators (or simply operators) for transforming datastreams. Each n-ary *dataflow operator* defines a function with n incoming arcs and one outgoing arc, except that the operator `split` (which will be described later) has one incoming arc and more than one outgoing arc. The datastreams at the incoming arcs are called *operand datastreams*, datons of which are called *operand datons*. The datastreams at the outgoing arcs are called *output datastreams*, datons of which are *output datons*. An operator *fires* a daton to its outgoing arc(s) by evaluating the result with the operand daton(s) and assigning the result to the output daton(s). Each operator runs in parallel independently. An operator fires immediately if all the required operand datons are available, regardless of whether other operators have fired or not. The unique *solution* of a Dataflow Network is the valuation θ over all the datons, so that the values of all the datons are the same as the result of the transformation of the operators.

Figure 2.1 is a simple dataflow network. In this example, + is a binary operator, which is denoted by a node labelled + with two incoming arcs and one outgoing arc. The datastream **1** supplies infinite datons of "1", meaning that $a[t] \mapsto 1$ for all $t \in \mathcal{Z} \geq 0$. The "0" at the arc b is a daton meaning that $b[0] \mapsto 0$.

The dataflow operator + requires two datons to operate at any time. At the 1^{st} firing of +, as $a[0] \mapsto 1$ and $b[0] \mapsto 0$, + reads the datons and fires an output daton "1" to b. At the 2^{nd} firing, as $a[1] \mapsto 1$ and $b[1] \mapsto 1$, + fires "2" to arc b. At the 3^{rd} firing, as $a[2] \mapsto 1$ and $b[2] \mapsto 2$, + fires "3" to the

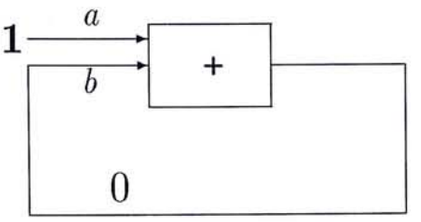


Figure 2.1: A simple dataflow network

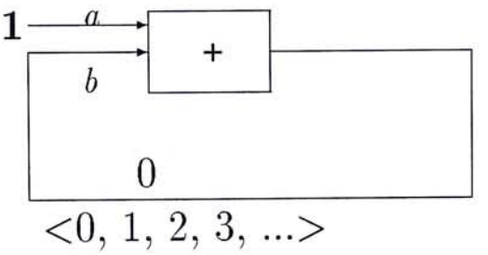


Figure 2.2: Generating a datastream with dataflow network

arc b. The process continues as shown in Figure 2.2. We can get the solution $\bigcup_{t \in \mathcal{Z}_{\geq 0}} \{b[t] \mapsto t+1\}$ by reading the arc b.

We can use dataflow networks to generate datastreams of more complicated patterns. Figure 2.3 shows the dataflow network solving a simplified version of the "Hamming's problem", which is to generate the sequence of all numbers of the form $2^i 3^j$ in increasing order without repetitions for all non-negative integers i and j: $1, 2, 3, 4, 6, 8, 9, 12$, and so on. The dataflow operators perform the following actions:

- `double` fires output datons which are equal to the operand datons times 2.
- `triple` fires output datons which are equal to the operand datons times 3.
- `split` fires multiple identical copies of its operand datons.
- `merge` fires an output datastream with datons obtained by sorting the

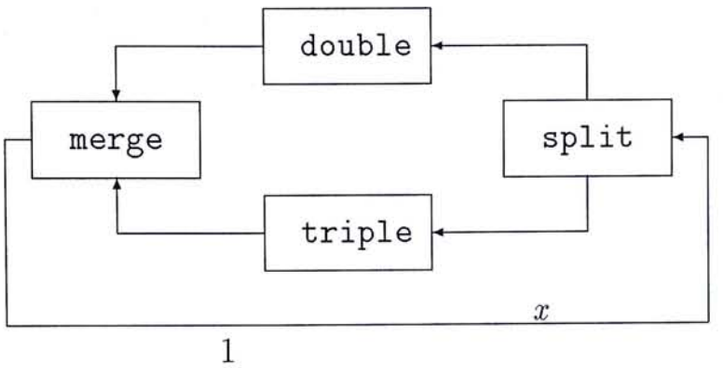


Figure 2.3: The dataflow network of the Hamming's problem

operand datons in increasing order with duplicated datons removed, provided that each of the operand datastreams is in increasing order.

Initially, all arcs are empty except the daton "1" in the arc x, implying that $x[0] \mapsto 1$. At the 1^{st} firing of `split`, it makes two copies of "1" as operand datons of `double` and `triple`, which fire "2" and "3" to arc y and z respectively. The smaller daton is "2", which is fired by `merge`, so that $x[1] \mapsto 2$. The daton "3" is waiting at the arc z. At the 2^{nd} firing, `split` makes two copies of "2" as operand datons of `double` and `triple`, which fire "4" and "6" to arc y and z respectively. Recall that "3" is waiting at arc z, the smaller daton among "4" and "3" is "3", which is fired by `merge`, so that $x[2] \mapsto 3$. The datons "4" and "6" are waiting at the arcs y and z respectively. At the 3^{rd} firing, `split` makes two copies of "3" as operand datons of `double` and `triple`, which fire "6" and "9" to arc y and z respectively. Recall that "4" and "6" are waiting at the arcs y and z respectively, the smaller daton is "4", which is fired by `merge`, so that $x[3] \mapsto 4$. The daton "6" is waiting at arc y, while "6" and "9" are waiting at the arc z. At the 4^{th} firing, `split` makes two copies of "4" as operand datons of `double` and `triple`, which fire "8" and "12" to arc y and z respectively. Recall that two "6"s are waiting at the arcs y and z respectively, both of them are consumed as they are the same. The daton "6" is fired by `merge`, so that $x[4] \mapsto 6$. The daton "8" is waiting at

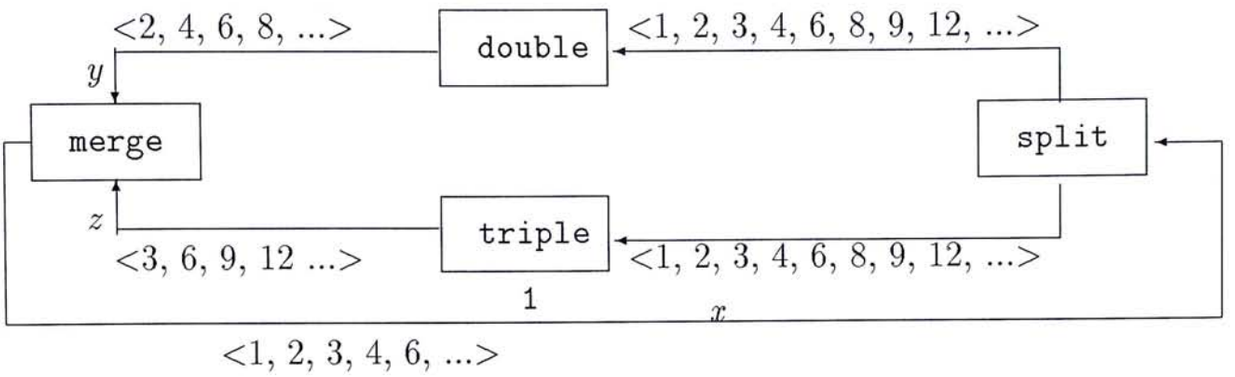


Figure 2.4: Generating datastreams with the dataflow network of the Hamming's problem

arc y, while "9" and "12" are waiting at the arc z. The process continues, as shown in Figure 2.4. We can get the sequence $\langle$ 1, 2, 3, 4, 6, ... $\rangle$ by reading the arc x.

The Dataflow Network solves the simplified Hamming's problem intuitively: $2^i 3^j$ implies the usage of `double` and `triple`. In fact, the problem is difficult to formulate in imperative programming languages because we want *all* numbers [34]. The advantages of Dataflow Networks are that they are easy and natural to express iterations and data flows directly.

2.3 The Lucid Programming Language

Lucid [3] is a dataflow programming language for specifying and executing Dataflow Networks. The following example Lucid program[1] gives a brief overview of the language.

```
a = 1;
b = a + 2;
```

A Lucid program consists of *statements*, each of which is an equation ended

[1]The Lucid code is shown in teletype font. Variables of the same name in teletype font and italic font should be understood interchangeably.

by a semicolon ";". There is a *variable* on the left hand side of an equation, and an *expression* on the right hand side. An expression is composed of *constants*, *variables*, *operators*, and *functions*. For example, `a = 1` is a statement, where `a` is a variable, and `1` is an expression with a constant `1`. Another example statement is `b = a + 2`, where `a` and `b` are variables and `a + 2` is an expression with one operator `+` and two operands: a variable `a` and a constant 2. Detail descriptions of the terms are given in the following subsections.

2.3.1 Daton Domain

Recall that each daton of a datastream can take a value from the daton domain. Daton domains of datastreams in Lucid can be numbers, boolean or any other types. There is no explicit declaration for daton domains. They are implicit and determined at compile time using type analysis of the associated expressions [24]. The type analysis is based on the types of the constant datastreams and operators.

2.3.2 Constants

Lucid provides numeric constants, **true**, **false**, and the special constant **eod**. As mentioned earlier, a constant is an infinite datastream of datons where all the datons are identical. For example, **3.2** denotes the real constant datastream with 3.2 as the values of all the datons, **false** denotes the boolean constant datastream with `false` as the values of all the datons, and **eod** denotes the special constant datastream with `eod` as the values of all the datons.

2.3.3 Variables

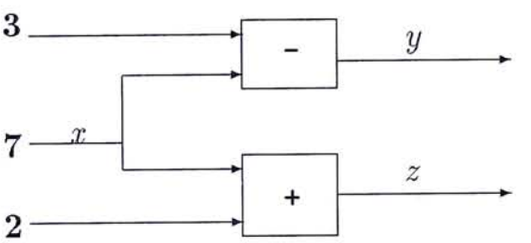
A variable in Lucid is simply a named expression. Its name can be any alphanumeric identifier starting with a letter and can include the special character "_" (underscore). For example, *root*, *_value*, *etc.*

2.3.4 Dataflow Operators

Lucid has several predefined dataflow operators that can be divided into two general categories: pointwise and non-pointwise operators. With *pointwise* operators, computing an output daton at a certain index in the datastream requires only the datons at the same index from the operand datastreams. All operators for which this is not true are *non-pointwise* operators.

An operator must have at least one incoming arcs and one outgoing arc regardless of its class. Lucid does not need the operator `split`. Instead, copies of a daton in an arc will automatically be made when necessary. It is achieved by labelling the required arc with a variable. Referring to the variable for multiple times will automatically make copies of the datons of the variable. For example, in the following program and the corresponding graph:

```
x = 7;
y = 3 - x;
z = x + 2;
```



The arc with operand datastream **7** is labelled x. Variable x is referred in `y = 3 - x` and `z = x + 2`, so that two copies of datons of x are made.

Arithmetic, Relational, and Logical Operators. The arithmetic, relational, logical, and bitwise logical operators in Lucid are similar to those in the C language [28] and they are all pointwise. The difference is that dataflow operators work on datastreams instead of scalars. The arithmetic operators are `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and unary `~` for negation. The relational operators are `>` for "greater than", `>=` for "not less than", `<` for "less than", `<=` for "not greater than", `eq` for equality, and `ne` for inequality. The logical operators are `and` for conjunction, `or` for disjunction, and `not` for negation. The following are some example expressions,

in where the left hand sides of = are expressions and the right hand sides are the corresponding values of datons in the datastreams.

$$\begin{aligned} A &= \langle\ 0, & 1, & 2, & 3, & \ldots\rangle \\ B &= \langle\ 1, & 3, & 5, & 7, & \ldots\rangle \\ A+B &= \langle\ 1, & 4, & 7, & 10, & \ldots\rangle \\ A<2 &= \langle\ \texttt{true}, & \texttt{true}, & \texttt{false}, & \texttt{false}, & \ldots\rangle \end{aligned}$$

Operators `first` and `next`. The `first` and `next` operators are the two simplest non-pointwise unary Lucid operators. Operator `first` returns a datastream, each daton of which is the first daton of the operand datastream. Operator `next` returns a datastream that consists of all but the first daton of its operand datastream with the order preserved. In other words, the output datastream is the operand datastream with two modifications: (a) the first daton is removed and (b) the indices of the rest of the datons are shifted by -1.

$$\begin{aligned} A &= \langle\ 0, & 1, & 2, & 3, & \ldots\rangle \\ \texttt{first}\ A &= \langle\ 0, & 0, & 0, & 0, & \ldots\rangle \\ \texttt{next}\ A &= \langle\ 1, & 2, & 3, & \ldots\rangle \\ \texttt{next next}\ A &= \langle\ 2, & 3, & \ldots\rangle \\ \texttt{first next next}\ A &= \langle\ 2, & 2, & 2, & 2, & \ldots\rangle \end{aligned}$$

Operator `fby`. The operator `fby` (pronounced "followed by") is a binary non-pointwise operator that accepts two operand datastreams and produces an output datastream, first daton of which is from the first daton of the first operand datastream and subsequent datons of which correspond to all the datons of the second operand datastream with order preserved.

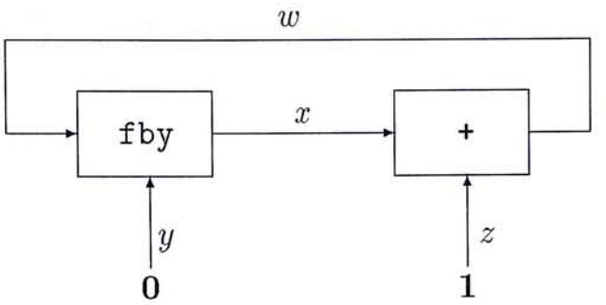


Figure 2.5: `x = 0 fby x + 1`

$$
\begin{aligned}
A &= \langle\ 0,\ 1,\ 2,\ 3,\ \ldots\rangle \\
B &= \langle\ 11,\ 21,\ 31,\ 41,\ \ldots\rangle \\
A\ \texttt{fby}\ B &= \langle\ 0,\ 11,\ 21,\ 31,\ 41,\ \ldots\rangle \\
B\ \texttt{fby}\ A &= \langle\ 11,\ 0,\ 1,\ 2,\ 3,\ \ldots\rangle \\
A\ \texttt{fby}\ B\ \texttt{fby}\ A + B &= \langle\ 0,\ 11,\ 11,\ 22,\ 33,\ \ldots\rangle \\
A\ \texttt{fby next}\ B &= \langle\ 0,\ 21,\ 31,\ 41,\ \ldots\rangle
\end{aligned}
$$

Figure 2.5 shows the dataflow network of the example Lucid code:

```
x = 0 fby x + 1;
```

The dataflow operator `fby` gets the operand daton "0" fired by the datastream **0**. Thus `fby` is able to fire daton "0" and $x[0] \mapsto 0$. The operator `+` reads the daton "0" and the "1" produced by the datastream **1**, so that `+` fires output daton "1". The output is read by `fby` which fires "1" and thus $x[1] \mapsto 1$. The operator `+` reads the daton "1" and the "1" produced by the datastream **1** to fire output daton "2". The output is read by `fby` which fires "2" and thus $x[2] \mapsto 2$. The process continues and we can see that $x[t] \mapsto t$ for all $t \in \mathcal{Z} \geq 0$.

Operators `attime`. The `attime` operator takes two operand datastreams, a base datastream and an index datastream to create an output datastream by using each daton of the index datastream as an index or a context into the base datastream.

$$
\begin{aligned}
A &= \langle\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots\rangle \\
B &= \langle\ 1,\ 2,\ 4,\ 8,\ 7,\ 0,\ 5,\ 2,\ 3,\ 9,\ \ldots\rangle \\
A\ \texttt{attime}\ B &= \langle\ 1,\ 2,\ 5,\ 34,\ 21,\ 1,\ 8,\ 2,\ 3,\ 55,\ \ldots\rangle
\end{aligned}
$$

Operator `if-then-else-fi`. The `if-then-else-fi` operator takes three datastreams, a boolean datastream and two base datastreams: namely P, A, and B respectively. At the $(t+1)^{th}$ firing, `if-then-else` fires daton of $A[t]$ if $P[t]$ is `true`, and $B[t]$ otherwise. Here is an example.

$$
\begin{aligned}
P &= \langle\ \texttt{true},\ \texttt{false},\ \texttt{false},\ \texttt{true},\ \ldots\rangle \\
A &= \langle\ 10,\ 1,\ 2,\ 3,\ \ldots\rangle \\
B &= \langle\ 21,\ 2,\ 4,\ 8,\ \ldots\rangle \\
\texttt{if}\ P\ \texttt{then}\ A\ \texttt{else}\ B\ \texttt{fi} &= \langle\ 10,\ 2,\ 4,\ 3,\ \ldots\rangle
\end{aligned}
$$

Operators `asa`, `wvr`, `upon`. The Lucid operator `asa` (shorthand for "as soon as") accepts two operand datastreams and produces an output datastream. This operator can be thought of as producing an operand dependent constant datastream. The daton of this constant is the k^{th} daton of the first operand datastream such that the k^{th} daton of the second operand datastream is `true` and all datons prior to it are `false`. Here is an example.

$$
\begin{aligned}
A &= \langle\ 10,\ 11,\ 29,\ 23,\ 13,\ \ldots\rangle \\
P &= \langle\ \texttt{false},\ \texttt{false},\ \texttt{false},\ \texttt{true},\ \texttt{true},\ \ldots\rangle \\
A\ \texttt{asa}\ P &= \langle\ 23,\ 23,\ 23,\ 23,\ 23,\ 23,\ 23,\ \ldots\rangle
\end{aligned}
$$

Operator `asa` can be defined in terms of `first`, `next` and `if-then-else`. The statement `y = x asa p` can be rewritten into:

```
y = first (if p then x else next y fi);        (2.1)
```

The operator `wvr` (shorthand for "whenever") has two operand datastreams and one output datastream. This operator selects those datons from the first operand datastream whose corresponding datons in the second operand datastream are `true`. The output datastream, in a sense, is a filtered version of

the first operand datastream where the extent of filtering is determined by the second operand datastream.

$$\begin{aligned} A &= \langle \quad 11, \quad 29, \quad 23, \quad 13, \quad 21, \quad 34, \quad \ldots \rangle \\ P &= \langle \ \texttt{false}, \ \texttt{false}, \ \texttt{true}, \ \texttt{true}, \ \texttt{false}, \ \texttt{true}, \ \ldots \rangle \\ A \ \texttt{wvr} \ P &= \langle \quad 23, \quad 13, \quad 34, \quad \ldots \rangle \end{aligned}$$

Operator `wvr` can be defined in terms of `attime`, `fby`, and `if-then-else`. The statement `y = x wvr p` can be rewritten into:

```
    y  =  x attime t2;
   t2  =  t1 fby (t1 attime t2+1);
   t1  =  if p then time else next t1 fi;
 time  =  0 fby time + 1;
```

(2.2)

The binary operator `upon` "stretches" the first operand datastream is based on the second datastream. Specifically, the k^{th} daton of the output datastream is the p^{th} ($p \leq k$) daton of the first operand datastream such that p of the first k datons of the second operand datastream are `true`.

$$\begin{aligned} A &= \langle \quad 10, \quad 11, \quad 29, \quad 23, \quad 5, \quad 8, \quad 13, \quad 21, \quad \ldots \rangle \\ P &= \langle \ \texttt{false}, \ \texttt{false}, \ \texttt{true}, \ \texttt{true}, \ \texttt{true}, \ \texttt{false}, \ \texttt{true}, \ \texttt{false}, \ \ldots \rangle \\ A \ \texttt{upon} \ P &= \langle \quad 10, \quad 10, \quad 10, \quad 11, \quad 29, \quad 23, \quad 23, \quad 5, \quad \ldots \rangle \end{aligned}$$

Operator `upon` can be defined in terms of `attime`, `fby`, and `if-then-else`. The statement `y = x upon p` can be rewritten into:

```
    y  =  x attime t1;
   t1  =  0 fby if p then t1+1 else t1 fi;
```

(2.3)

Operator `iseod`. Operator `iseod` tests if its daton operand is "eod". It returns `true` if it is and returns `false` otherwise.

$$\begin{aligned} X &= \langle \quad 1, \quad 4, \quad \texttt{eod}, \quad \texttt{eod}, \quad \texttt{eod}, \quad \ldots \rangle \\ \texttt{iseod} \ X &= \langle \ \texttt{false}, \ \texttt{false}, \ \texttt{true}, \ \texttt{true}, \ \texttt{true}, \ \ldots \rangle \end{aligned}$$

Associativity and Precedence Rules The following table gives the associativity of infix operators in Lucid:

Associativity	Operators
left	`+, -, *, /, or, and, asa, wvr, upon, if-then-else-fi`
right	`fby`

We list here the hierarchy of precedences amongst Lucid operators. Operators with lowest precedences are at the top of the list, and ones with highest precedences are at the bottom.

`fby`
`asa, upon, wvr`
`if-then-else-fi`
`or`
`and`
`not`
`eq, <, <=, >, >=`
`+, -`
`*, /`
`first, next, iseod`

2.3.5 Functions

Other than operators, nodes of Dataflow Network can also be represented by Lucid functions. Lucid has several predefined functions, including `sin` for sine, `cos` for cosine, `tan` for tangent, `log` for logarithm, `sq` for square, `sqr` for square root, and `pow` for power. Each function must have one outgoing arc and at least one incoming arcs.

2.3.6 Expression and Statement

An *expression* is comprised of constants, variables, operators, and functions. An expression can be just a constant datastream, *e.g.* `7`; or a variable, *e.g.* `x`; or it can be operators / functions with their corresponding operands, *e.g.* `a+7`. A *statement* is an equation for a variable and an expression, *e.g.* `x = a+7`. In a statement s: x `=` E, s and E are the *defining statement* and *defining expression* of x respectively, while x is their *defined variable.* In a Lucid program, each variable has a unique defining statement and a unique defining expression.

2.3.7 Examples

We use some examples to illustrate applications that can be modelled by Lucid.

Fibonacci Sequence

A Fibonacci sequence *fib* is defined mathematically as follows:

$$fib_t = \begin{cases} t & \texttt{if } t = 0 \texttt{ or } t = 1 \\ fib_{t-1} + fib_{t-2} & \texttt{otherwise} \end{cases}$$

The problem can be coded in Lucid as:

```
fib = 0 fby 1 fby fib + next fib;
```

The statement states that the first and second datons of *fib* are 0 and 1 respectively. The subsequent datons are computed by adding the preceding two datons. To compute *fib*[3], one needs to compute the sum of *fib*[2] and *fib*[1]. We know that *fib*[1] is assigned 1 and *fib*[2] is the sum of *fib*[0] and *fib*[1], which are already assigned values.

From this example, we can see that the problem can be coded in Lucid naturally and intuitively.

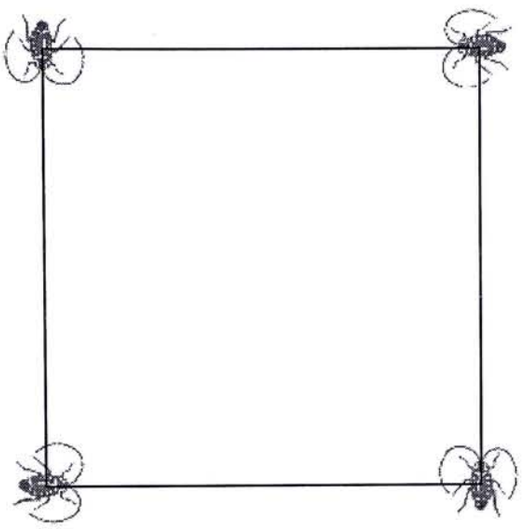


Figure 2.6: The Four Cockroaches Problem

Four Cockroaches Problem

There are four cockroaches at the four corners of a square, one facing another in a cyclic manner in the anti-clockwise direction, with corresponding start coordinates [0.0, 0.0], [10.0, 0.0], [10.0, 10.0], and [0.0, 10.0] as shown in Figure 2.6. Each cockroach walks towards the next cockroach in anti-clockwise direction, and they can move distance unit d in each time unit. We would like to find out the traces of the four cockroaches. We model the problem with the following Lucid code, where the datastreams xi and yi represent the x and y coordinates of cockroach i respectively.

The step distance d is defined as:

```
d = 0.01;
```

The distance between cockroaches i and j is $\sqrt{(xj - xi)^2 + (yj - yi)^2}$. The coordinates are defined as:

```
x1 = 0.0 fby x1 + d*(x2 - x1) / sqr(sq(x2 - x1) + sq(y2 - y1));
y1 = 0.0 fby y1 + d*(y2 - y1) / sqr(sq(x2 - x1) + sq(y2 - y1));
x2 = 10.0 fby x2 + d*(x3 - x2) / sqr(sq(x3 - x2) + sq(y3 - y2));
```

t	x1[t]	y1[t]	x2[t]	y2[t]	x3[t]	y3[t]	x4[t]	y4[t]
0	0.0	0.0	10.0	0.0	10.0	10.0	0.0	10.0
1	0.01	0.0	10.0	0.01	9.99	10.0	0.0	9.99
2	0.02	1.001e-05	9.99999	0.02	9.98	9.99999	1.001e-05	9.98
3	0.03	3.004e-05	9.99997	0.03	9.97	9.99997	3.004e-05	9.97
4	0.0399999	6.01001e-05	9.99994	0.0399999	9.96	9.99994	6.01001e-05	9.96
5	0.0499998	0.0001002	9.9999	0.0499998	9.95	9.9999	0.0001002	9.95
6	0.0599997	0.000150351	9.99985	0.0599997	9.94	9.99985	0.000150351	9.94
7	0.0699995	0.000210561	9.99979	0.0699995	9.93	9.99979	0.000210561	9.93
8	0.0799993	0.000280842	9.99972	0.0799993	9.92	9.99972	0.000280842	9.92
9	0.089999	0.000361203	9.99964	0.089999	9.91	9.99964	0.000361203	9.91

Table 2.1: Generating a datastream with dataflow network

```
y2 = 0.0 fby y2 + d*(y3 - y2) / sqr(sq(x3 - x2) + sq(y3 - y2));
x3 = 10.0 fby x3 + d*(x4 - x3) / sqr(sq(x4 - x3) + sq(y4 - y3));
y3 = 10.0 fby y3 + d*(y4 - y3) / sqr(sq(x4 - x3) + sq(y4 - y3));
x4 = 0.0 fby x4 + d*(x1 - x4) / sqr(sq(x1 - x4) + sq(y1 - y4));
y4 = 10.0 fby y4 + d*(y1 - y4) / sqr(sq(x1 - x4) + sq(y1 - y4));
```

The solution of the coordinates at the first 10 time units are shown in Table 2.1.

As shown in Figure 2.7, by plotting the solution at the first several thousand time units, we can find that the four cockroaches will finally meet at the center of the squares.

2.3.8 Implementation

Figure 2.8 shows the pseudo-code of the main function of the pLucid interpreter, `LUCIDinterpreter`. Instead of printing the values of all variables, `LUCIDinterpreter` prints only the values of variables of interest. The parameter of `LUCIDinterpreter` is P, the set of variables of interest. At Line 1 in the algorithm, ζ is the valuation consisting of assignments we have obtained,

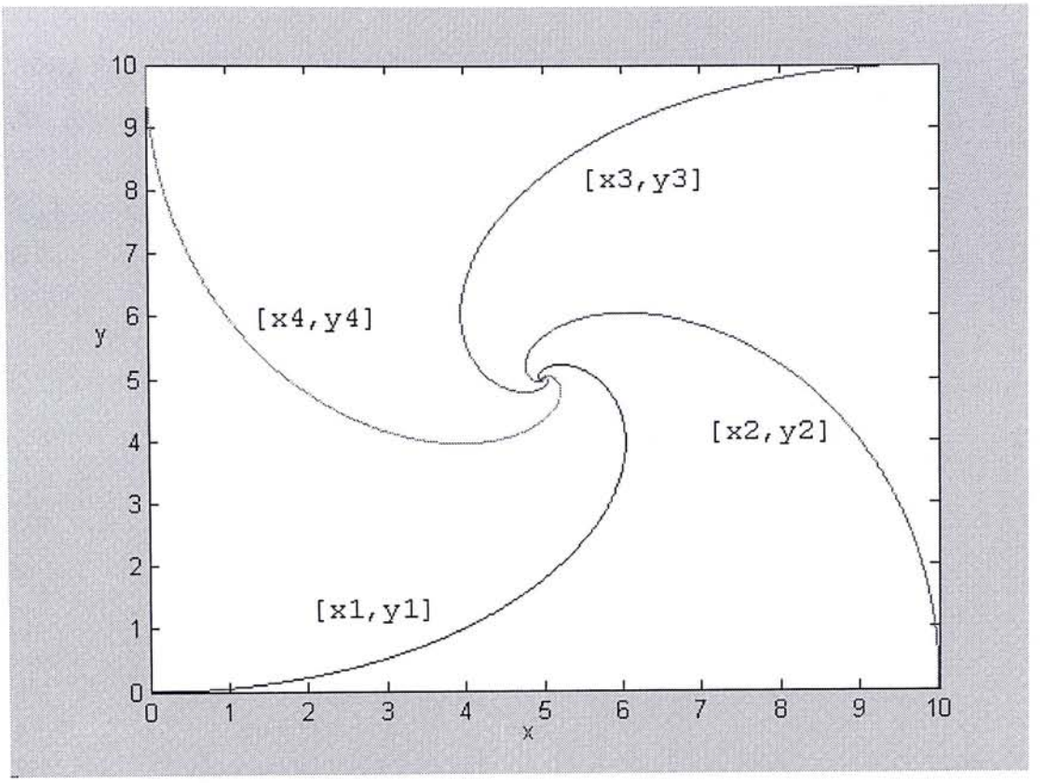


Figure 2.7: Solution of the Four Cockroaches Problem

and is initialized to {}. At the **foreach** block at Line 2, all datons of variables of interest with index t are assigned values and printed. When we mention "at stage i" in Dataflow Network afterwards, it means the stage when the loop index t of **foreach** block at Line 2 is i. At the stage, all variables of interest at index $i-1$ have been assigned values, but at least one of those at index i have not been assigned values.

We are unable to solve a Dataflow Network within finite time due to the infinite number of datons of datastreams. For example, we are unable to find values of the whole Hamming sequence within finite time by modelling the Hamming problem in Dataflow Network, as the sequence is a datastream with infinite number of datons.

If the pLucid interpreter prints the solution only when the problem is completely solved, nothing will be printed and the program will run forever as the solution will never be found. Instead, pLucid interpreter prints *partial*

```
/*
P : The set of variables of interest
t : An integer counting stage
*/
LUCIDinterpreter (P)
begin
    for ∀t ∈ 𝒵 ≥ 0 do
1       ζ := {};
2       foreach v ∈ P do
            print demand (v[t], t, ζ);
        end
    end
end

/*
cvof : A constant, variable, operator or function
u : The value of daton to be driven after demand
*/
demand (cvof, t, ζ)
begin
    if cvof is a variable then
        /* if cvof[t] is already assigned a value*/
        if ∃value s.t. (cvof[t] ↦ value) ∈ ζ then
            u := value;
        else
            driving := the constant / variable / operator / function driving
            cvof[t];
            u := demand (driving, t, ζ);
            ζ := ζ ∪ {cvof[t] ↦ u};
        end
    else if cvof is an operator or function then
        /* This is the (t + 1)^th firing of cvof*/
        the operands of cvof are r_1, ..., r_n;
        foreach operand r_i do
            t' := the index of r_i;
            /* For example, next has one operand with t' = t − 1 */
            value_i := demand (r_i, t', ζ);
        end
        u := evaluation of cvof with operand values {value_1, ..., value_n};
    else if cvof is a constant then
        u := value of cvof[0];

    return u;
end
```

Figure 2.8: Pseudo-code of `LUCIDinterpreter` and `demand`

solution, which is a subset of the solution. At the **foreach** block at Line 2 of Figure 2.8, for each $v \in P$, $v[t]$ is printed immediately when it is assigned a value. We do not need to suspend the printing until $v[t]$ is assigned a value for all $t \in \mathcal{Z} \geq 0$, as the valuation obtained must be a partial solution.

`LUCIDinterpreter` assigns a value to a daton by a *demand-driven* schema. With the schema, when a constant of a Dataflow Network is *demanded*, it *drives* the daton waiting at the constant. When a node is *demanded*, the incoming arcs of the node are demanded. When an arc is *demanded*, it *drives* the daton waiting at the arc. If no daton is waiting, the node at the tail of the arc is demanded.

Figure 2.8 shows the pseud-codes of `demand`, the function handling the demand-driven schema. One parameter of `demand` is *cvof*, which can be a constant, a variable, an operator, or a function to be demanded. The parameter t records the index. The parameter ζ is the valuation consisting of the assignments we have got. The function `demand` stores the value of daton to be driven at u and returns it. If *cvof* is a constant datastream, its daton is driven. In the example dataflow network shown in Figure 2.9 (a), if *cvof* is **7**, "7" is driven. If *cvof* is an operator or a function, the operand datastreams of *cvof* are demanded and the evaluation result is driven. In the example dataflow network shown in Figure 2.9 (b), if *cvof* is `next`, "2" and "3" are driven at the 1^{st} and 2^{nd} demands respectively. If *cvof* is a variable, the value of $cvof[t]$ is driven if $cvof[t]$ has been assigned a value; otherwise the value is obtained by demanding the constant, variable, operator, or function driving the daton to $cvof[t]$. In the example dataflow network shown in Figure 2.9 (c), if *cvof* is x, $x[t]$ is driven if $x[t]$ has been assigned a value, or `fby` is demanded otherwise.

Example 2.1 Taking the dataflow network of Figure 2.5 as an example, we suppose x is the variable of interest.

At stage 0, x demands `fby` to get the value of the first daton. For simplicity,

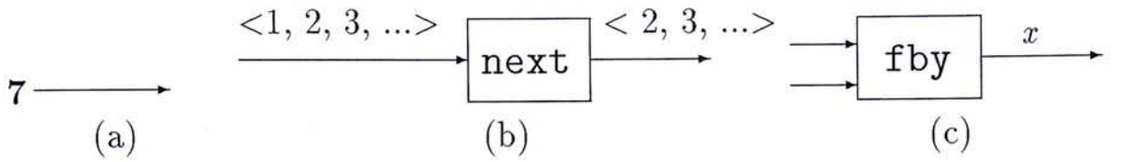


Figure 2.9: Examples of Demands

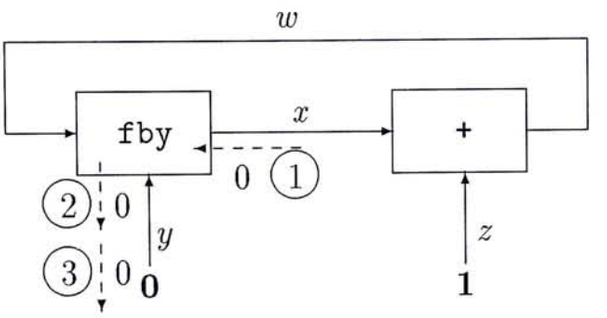


Figure 2.10: Demands at stage 0

we express it as "the daton $x[0]$ demands `fby`". The operator `fby` demands y, which demands **0**. The constant datastream **0** drives "0" to y, which drives "0" to `fby`. The operator `fby` drives "0" to $x[0]$. Thus we obtain the valuation $\{x[0] \mapsto 0\}$. The process can be represented by Figure 2.10, where the dash arrows represent that the nodes or arcs at the arrow tails demand datons of the nodes or arcs at the arrow heads. The circled numbers labelling the dash arrows are the order of demands. The uncircled numbers labelling the dash arrows are the datons to be driven as the result of the demand.

At stage 1, $x[1]$ demands `fby`, which demands $w[0]$. The daton $w[0]$ demands `+`, which demands $z[0]$ and $x[0]$. The daton $z[0]$ demands **1**, which drives "1" to $z[0]$. The variable z drives "1" to `+`. As $x[0]$ is already assigned 0, "0" is driven to `+`. The operator `+` evaluates $1 + 0$ and drives the result, "1", to $w[0]$. The variable w drives "1" to `fby`, which drives "1" to $x[1]$. Thus we obtain the valuation $\{x[0] \mapsto 0,\ x[1] \mapsto 1\}$. The process is as represented in Figure 2.11, where the repeated order numbers indicate that the demands

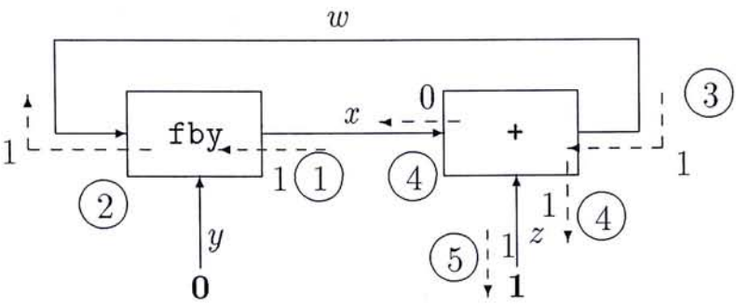


Figure 2.11: Demands at stage 1

are invoked in parallel. The process continues and we can obtain the values of other datons of x. □

Chapter 3

Extended Dataflow Network

This chapter presents Extended Dataflow Network, which is an extension of Dataflow Network with two additional components, namely "assertion arcs" and "selection operators". Their usages are introduced with examples. We also define *E-Lucid*, which is a language for specifying Extended Dataflow Network.

3.1 Assertion Arcs

An *Extended Dataflow Network* is a Dataflow Network associated with a special set of boolean arcs V_A, where the arcs are called *assertion arcs*. An assertion arc is *satisfied* if it is assigned the constant datastream **true**; it is *unsatisfied* otherwise. A *solution* of an Extended Dataflow Network is a valuation over all datons of all labelled arcs, so that the values of all the datons are the same as the result of the transformation of the operators, and all assertion arcs in V_A are satisfied. In graphical representation of Extended Dataflow Networks, an assertion arc is an arrow with a hollow arrow head, whereas an ordinary arc is represented by an arrow with a solid head. In Figures 3.1 and 3.2, the arcs labelled z are assertion arcs.

In the example of Figure 3.1, $x[t]$ is t, $y[t]$ is $t+1$, and $y[t]-x[t]$ is 1 for $t \in \mathcal{Z} \geq 0$. We can see that $z[t] \mapsto \texttt{true}$ for $t \in \mathcal{Z} \geq 0$, which means that z is

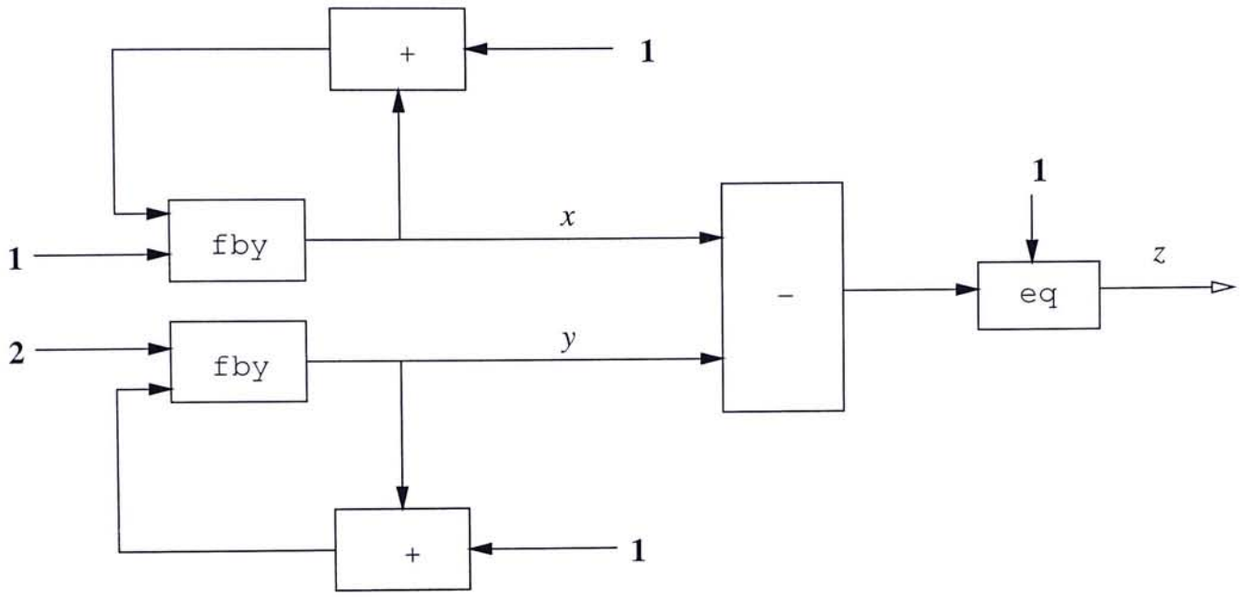


Figure 3.1: An Extended Dataflow Network with an assertion variable

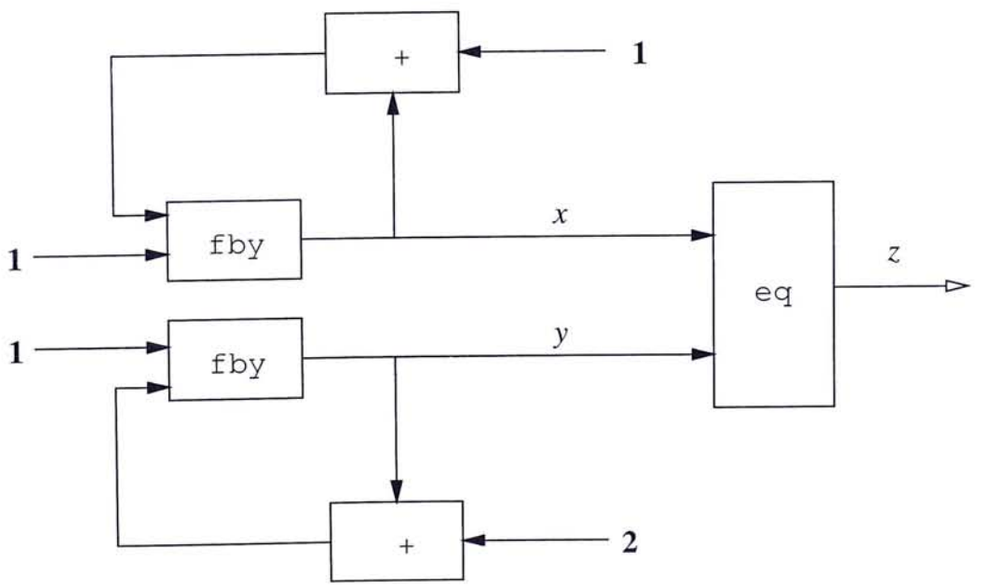


Figure 3.2: An Extended Dataflow Network with an assertion variable but without solution

satisfied. Thus the only solution is $\bigcup_{t \in \mathcal{Z} \geq 0} \{x[t] \mapsto t, y[t] \mapsto t+1, z[t] \mapsto \texttt{true}\}$.

In the example of Figure 3.2, $x[t] \mapsto t + 1$, $y[t] \mapsto 2t + 1$, $y[0] = x[0]$, and $y[t + 1] \neq x[t + 1]$ for $t \in \mathcal{Z} \geq 0$. We can see that $z[0] \mapsto \texttt{true}$ and $z[t + 1] \mapsto \texttt{false}$ for $t \in \mathcal{Z} \geq 0$. As $z \neq \textbf{true}$, z is unsatisfied. Thus there is no solution.

3.2 Selection Operators

There are four selection operators, namely the "discrete choice operator", the "discrete committed choice operator", the "range choice operator", and the "range committed choice operator". They allow selection of datons from a collection of datastreams to fire. Thus they are collectively named as *selection operators.* Graphically, this class of operators are denoted by a diamond-shaped node. In this section, we give their syntactic construction rules, and explain their meanings and usages.

3.2.1 The Discrete Choice Operator

The *discrete choice operator* is an n-ary commutative and non-pointwise operator, denoted by "[[]]". The output datons are chosen from one of its operand datons. Suppose the operand datastreams of the discrete choice operator are $x_1, \ldots, x_n$, and the output datastream is y. The output daton $y[t]$ at the $(t+1)^{th}$ firing of the operator must be equal to one of $x_1[t], \ldots, x_n[t]$. In other words, $\bigvee_{t \in \mathcal{Z} \geq 0} \{y[t] = x_i[t]\}$ is true.

We use the Extended Dataflow Network in Figure 3.3 (a) to illustrate the idea. From the shaded part of Figure 3.3 (b), we can see that $x[t] \mapsto t + 1$ for $t \in \mathcal{Z} \geq 0$. Similarly, $y[t] \mapsto t+2$ for $t \in \mathcal{Z} \geq 0$ in the shaded part of Figure 3.3 (c). In the shaded part of Figure 3.3 (d), $a[t]$ can take either the value of $x[t]$ or $y[t]$ for $t \in \mathcal{Z} \geq 0$. Thus the set of solutions is $\{\bigcup_{t \in \mathcal{Z} \geq 0} \{a[t] \mapsto k_t, x[t] \mapsto t+1, y[t] \mapsto t + 2\} \mid k_i = i + 1 \vee k_i = i + 2, i \in \mathcal{Z} \geq 0\}$.

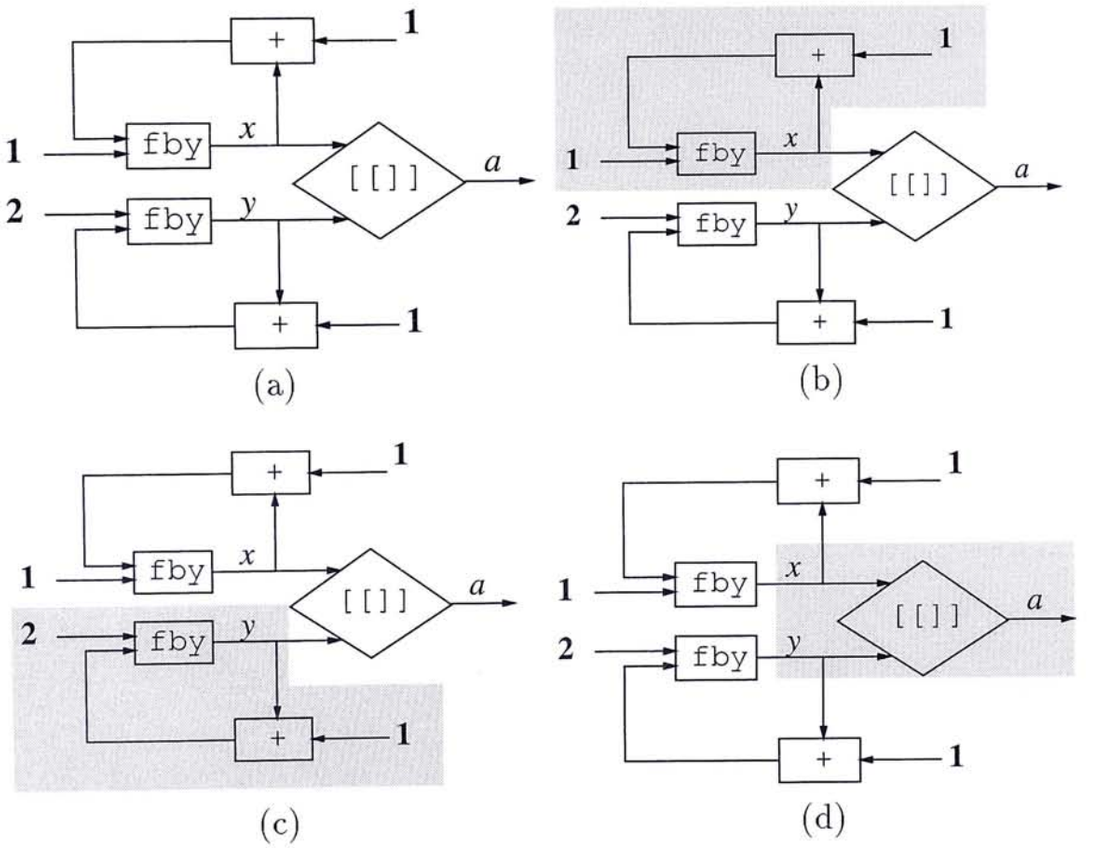


Figure 3.3: An Extended Dataflow Network with a discrete choice operator

3.2.2 The Discrete Committed Choice Operator

The *discrete committed choice operator* is an n-ary commutative and pointwise operator, denoted by "[]". The output datastream is chosen from one of its operands. Note that different from the discrete choice operator, the discrete committed choice operator chooses an entire datastream rather than individual datons. It means that the operator always takes datons from the same arc once the arc is chosen. Suppose the operand datastreams of the discrete committed choice operator are $x_1, \ldots, x_n$. The output datastream y is equal to one of them.

We use the example network in Figure 3.4 (a) to illustrate the idea. From the shaded part of Figure 3.4 (b), we can see that $x[t] \mapsto t+1$ for $t \in \mathcal{Z} \geq 0$. Similarly, $y[t] \mapsto t+2$ for $t \in \mathcal{Z} \geq 0$ in the shaded part of Figure 3.4 (c). In the shaded part of Figure 3.4 (d), a can take either the datastream x or y. It means that either all the datons of a are taken from x, or all of them are taken from y. Thus the two possible solutions are $\bigcup_{t \in \mathcal{Z} \geq 0}\{x[t] \mapsto t+1,\ y[t] \mapsto t+2,\ a[t] \mapsto t+1\}$ and $\bigcup_{t \in \mathcal{Z} \geq 0}\{x[t] \mapsto t+1,\ y[t] \mapsto t+2,\ a[t] \mapsto t+2\}$.

3.2.3 The Range Choice Operators

The family of *range choice operators* is a set of binary non-pointwise and commutative operators, denoted by "[[:]]", "{{:]]", "[[:}}", and "{{:}}". Each operator of the family has two operands, which are constant datastreams either both in the integer or both in the real domain, where the first operand has a smaller value than the second one. At each firing of the operator, an output daton with value lying between the values of the two operand datons is fired. It means that the values of the output datons of "[[:]]" at the $(t+1)^{th}$ firing of the operator must (1) be greater than or equal to the value of the first operand daton at index t, (2) be less than or equal to the value of the second operand daton at index t, and (3) have the same daton domain as the operand datons.

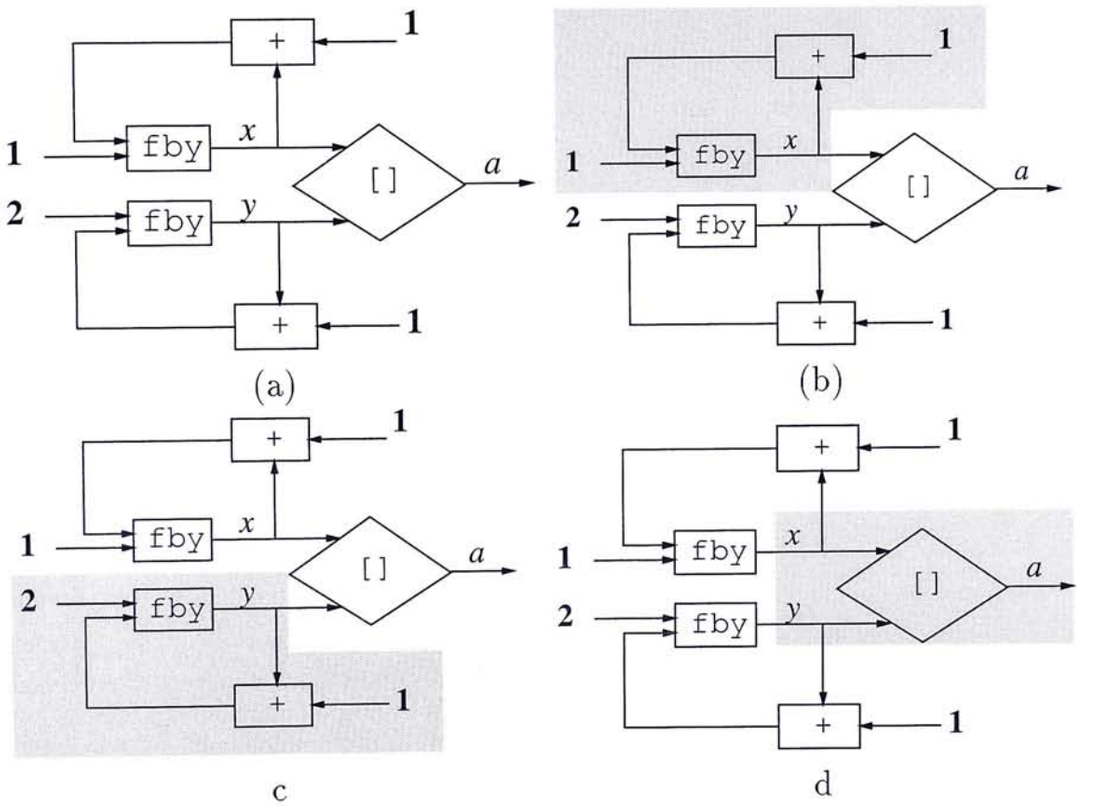


Figure 3.4: An Extended Dataflow Network with an discrete committed choice operator

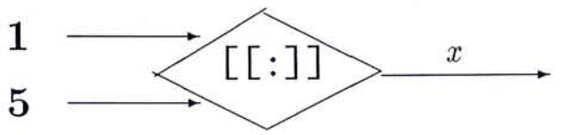


Figure 3.5: An Extended Dataflow Network with a range choice operator, of which operands have integer daton domains

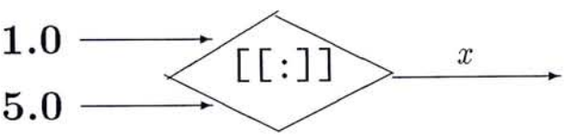


Figure 3.6: An Extended Dataflow Network with a range choice operator, of which operands have real daton domains

The other three operators are similar, except that the output datons at index t cannot be equal to the value of the first operand daton for the operators with open brace "{{", and cannot be equal to the value of the second operand daton for the operators with close brace "}}". Suppose the two operand datastreams of "[[:]]" are x_1 and x_2 with the same daton domain U, the output datastream is y, and $t \in \mathcal{Z} \geq 0$. If $y[t]$ is assigned the value k, $x_1[t] \leq k \leq x_2[t] \wedge k \in U$ is true. If the operator "[[:]]" is replaced by "{{:]]", $x_1[t] < k \leq x_2[t] \wedge k \in U$ is true. If the operator "[[:]]" is replaced by "[[:}}", $x_1[t] \leq k < x_2[t] \wedge k \in U$ is true. If the operator "[[:]]" is replaced by "{{:}}", $x_1[t] < k < x_2[t] \wedge k \in U$ is true.

In the example of Figure 3.5, x has integer daton domain. Thus $x[t]$ can take values 1, 2, 3, 4, or 5 for $t \in \mathcal{Z} \geq 0$. In the example of Figure 3.6, x has real daton domain. Thus $x[t]$ can take any real value between 1.0 and 5.0 for $t \in \mathcal{Z} \geq 0$.

3.2.4 The Range Committed Choice Operators

The family of *range committed choice operators* is a set of binary pointwise operators, denoted by "[:]", "{:]", "[:}" and "{:}". Each operator of the family has two operands, which are constant datastreams either both in the integer or both in the real domain, where the first operand has a smaller value than the second one. A constant output datastream with daton values lying between the values of the two operands is fired. It means that the values of the output datons of "[:]" must (1) be greater than or equal to the value of the first operand datons, (2) be less than or equal to the value of the second operand datons, (3) have the same daton domain as the operand daton, and (4) be equal to the output daton at index 0. The other three operators have the same properties, except that the output datons must not be equal to the value of the first operand datons for the operators with open brace "{", and must not be equal to the value of the second operand datons for the operators with close brace "}". Suppose the two operand datastreams of "[:]" are x_1 and x_2 with the same daton domain U, the output datastream is y, and $t \in \mathcal{Z} \geq 0$. If $y[t]$ is assigned the value k, $x_1[t'] \leq k \leq x_2[t'] \wedge k \in U \wedge t' \in \mathcal{Z} \geq 0$ is true. If the operator "[:]" is replaced by "{:]", $x_1[t'] < k \leq x_2[t'] \wedge k \in U \wedge t' \in \mathcal{Z} \geq 0$ is true. If the operator "[:]" is replaced by "[:}", $x_1[t'] \leq k < x_2[t'] \wedge k \in U \wedge t' \in \mathcal{Z} \geq 0$ is true. If the operator "[:]" is replaced by "{:}", $x_1[t'] < k < x_2[t'] \wedge k \in U \wedge t' \in \mathcal{Z} \geq 0$ is true.

In the example of Figure 3.7, x has integer daton domain. Thus x is **1**, **2**, **3**, **4**, or **5**. In the example of Figure 3.8, x has real daton domain. Thus x can be any constant datastream with daton values ranging from 1.0 to 5.0. It means that x is a constant datastream with real daton value k, where $1.0 \leq k \leq 5.0$. Different from the examples in Figure 3.5 and 3.6, $x[t]$ must be equal to $x[0]$ for any positive integer t in Figure 3.7 and 3.8.

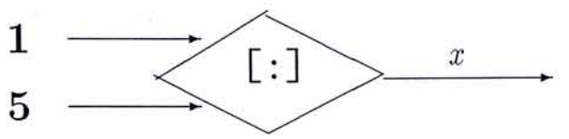


Figure 3.7: An Extended Dataflow Network with a range committed choice operator, of which operands have integer daton domains

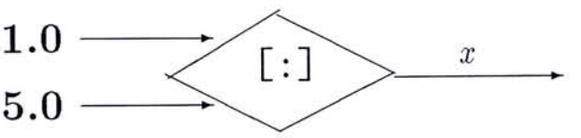


Figure 3.8: An Extended Dataflow Network with a range committed choice operator, of which operands have real daton domains

3.3 Examples

We present examples of Extended Dataflow Networks with both assertion arcs and selection operators to illustrate their interactions. In this section, we use the term "at stage t" to represent the stage after the valuation of $z[t-1]$ and before that of $z[t]$. If t is 0, "at stage t" represents the stage before the valuation of $z[0]$.

Example 3.1 Consider the example network in Figure 3.9.

At stage 0, $x[0]$ and $y[0]$ take the datons "1" and "3" respectively. It means that $x[0] \mapsto 1$ and $y[0] \mapsto 3$ as shown in Figure 3.10. Thus $y[0] - x[0]$ is 2 and $z[0] \mapsto \texttt{true}$.

At stage 1 as shown in Figure 3.11, as $z[1]$ can take `true` only if we have the valuation $\{u[0] \mapsto 1, v[0] \mapsto 1\}$ or $\{u[0] \mapsto 2, v[0] \mapsto 2\}$. One of the valuations is chosen. For instance, the former is chosen. Thus we have $x[1] \mapsto 2$, $y[1] \mapsto 4$, and $z[1] \mapsto \texttt{true}$.

At stage 2 as shown in Figure 3.12, as the discrete committed choice operator "[]" had chosen the daton from **1** at stage 1, it takes "1" from the same arc. The discrete choice operator "[[]]" chooses "1" so that $z[2]$ can take "`true`".

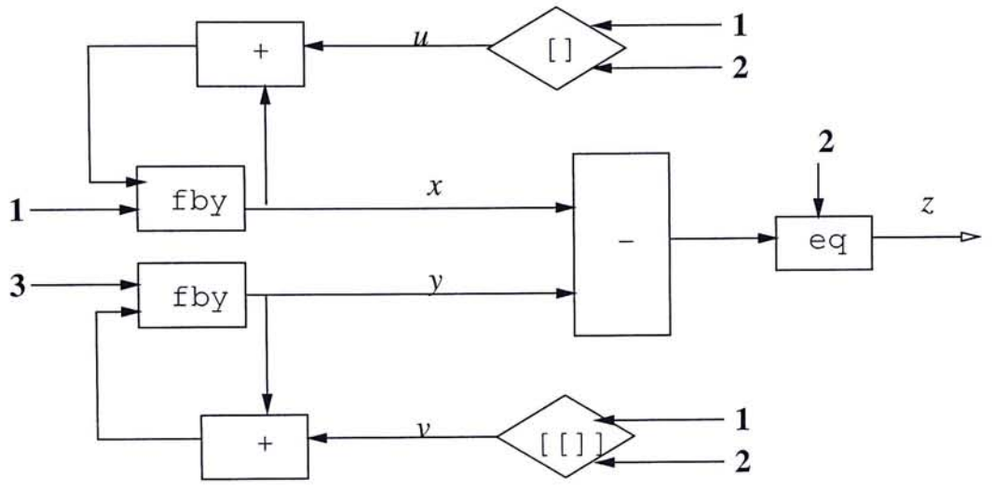


Figure 3.9: An Extended Dataflow Network with discrete choice operator, committed operator, and assertion variable

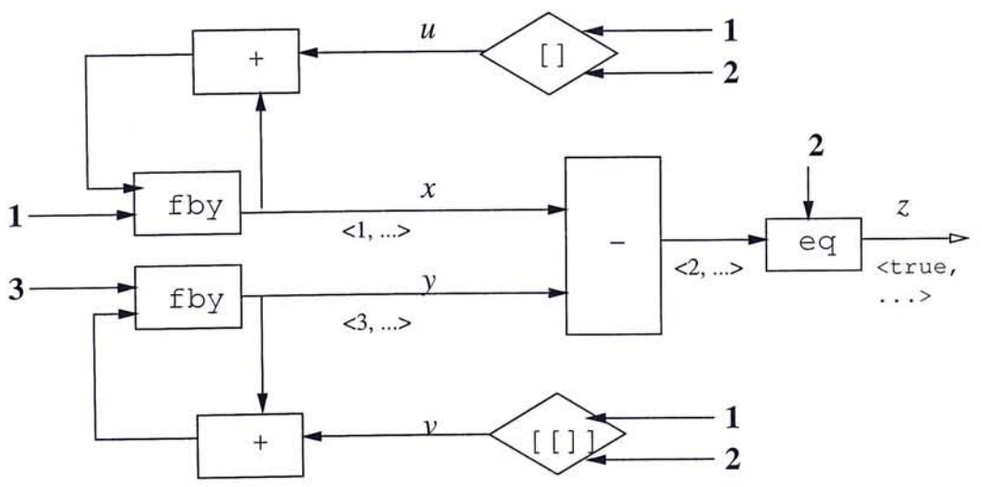


Figure 3.10: Valuation of the network in Figure 3.9 at stage 0

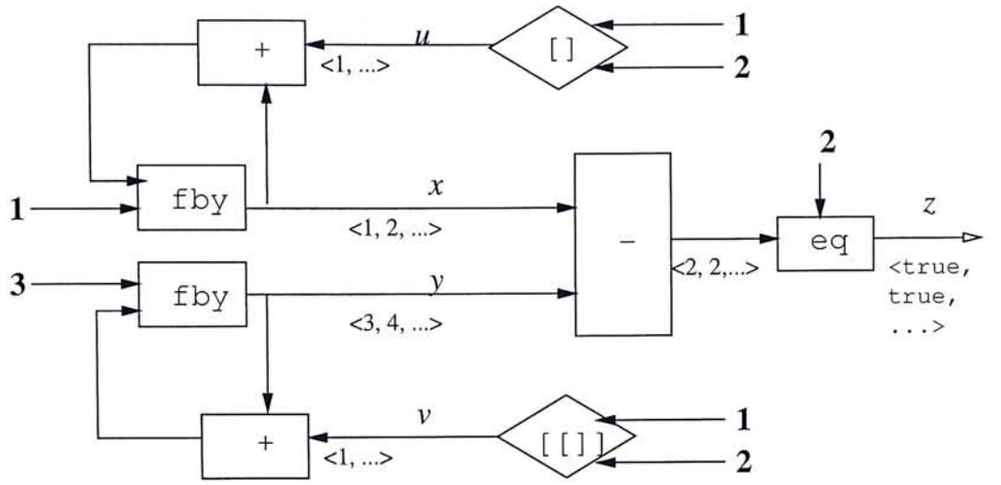


Figure 3.11: Valuation of the network in Figure 3.9 at stage 1, where $u[0] \mapsto 1$ and $v[0] \mapsto 1$.

We then get $u[1] \mapsto 1$, $v[1] \mapsto 1$, $x[2] \mapsto 3$, $y[2] \mapsto 5$, and $z[2] \mapsto$ `true`. The process continues, and we can get the solution $\bigcup_{t \in Z_{\geq 0}} \{u[t] \mapsto 1,\ v[t] \mapsto 1,\ x[t] \mapsto t+1,\ y[t] \mapsto t+3\}$.

If $u[0] \mapsto 2$ and $v[0] \mapsto 2$ instead, implying that $x[1] \mapsto 3$, $y[1] \mapsto 5$, and $z[1] \mapsto$ `true` as shown in Figure 3.13, another solution will be generated. At stage 2 as shown in Figure 3.14, "2" is chosen by the discrete committed choice operator "[]" from the same arc chosen last time. The daton "2" is chosen by the discrete choice operator "[[]]" so that $z[2]$ is possible to take "`true`". We then get $x[2] \mapsto 5$, $y[2] \mapsto 7$, and $z[2] \mapsto$ `true`. The process continues, and we can get the solution $\bigcup_{t \in Z_{\geq 0}} \{u[t] \mapsto 2,\ v[t] \mapsto 2,\ x[t] \mapsto 2t+1,\ y[t] \mapsto 2t+3\}$.

□

Example 3.2 In Figure 3.15, it is given that x and y have integer daton domains. At stage 0, as $z[0]$ can take "`true`" only if $x[0] < y[0]$, the possible valuations are $\{x[0] \mapsto 1,\ y[0] \mapsto 2\}$, $\{x[0] \mapsto 1,\ y[0] \mapsto 3\}$, $\{x[0] \mapsto 1,\ y[0] \mapsto 4\}$, $\{x[0] \mapsto 2,\ y[0] \mapsto 3\}$, $\{x[0] \mapsto 2,\ y[0] \mapsto 4\}$, and $\{x[0] \mapsto 3,\ y[0] \mapsto 4\}$. One of the valuations is chosen, for instance, $\{x[0] \mapsto 1,\ y[0] \mapsto 2\}$ as shown in Figure 3.16. Thus we have $x[0] \mapsto 1$, $y[0] \mapsto 2$, and $z[0] \mapsto$ `true` .

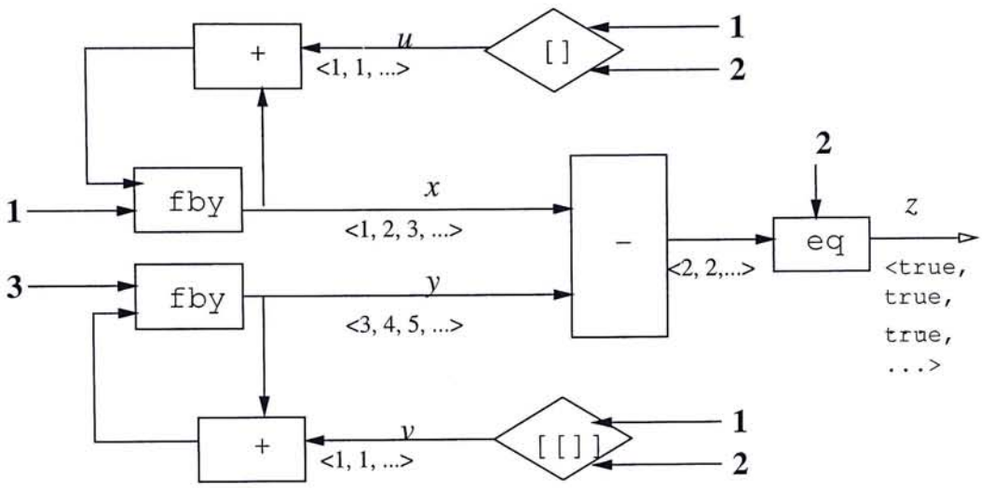


Figure 3.12: Valuation of the network in Figure 3.9 at stage 2, where $u[0] \mapsto 1$ and $v[0] \mapsto 1$.

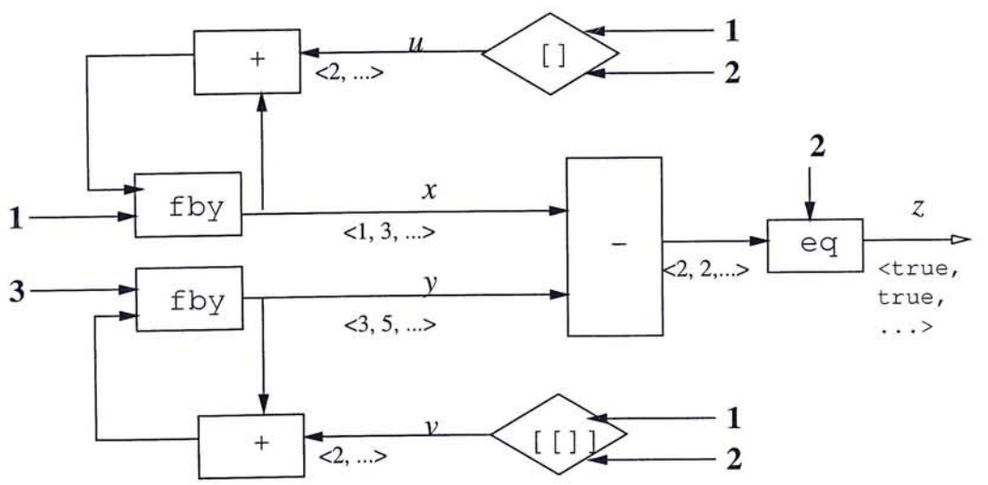


Figure 3.13: Valuation of the network in Figure 3.9 at stage 1, where $u[0] \mapsto 2$ and $v[0] \mapsto 2$.

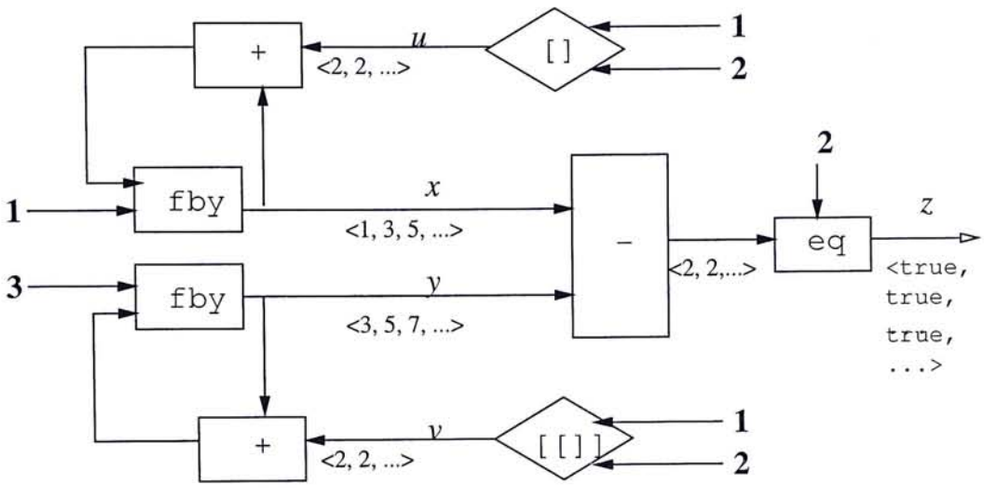


Figure 3.14: Valuation of the network in Figure 3.9 at stage 2, where $u[0] \mapsto 2$ and $v[0] \mapsto 2$.

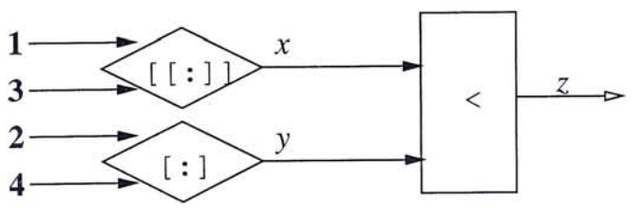


Figure 3.15: An Extended Dataflow Network with range choice operator, range committed operator, and assertion variable

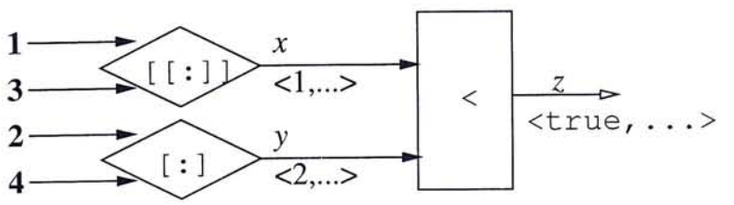


Figure 3.16: The valuation of the network in Figure 3.15 at stage 0, where $x[0] \mapsto 1$ and $y[0] \mapsto 2$.

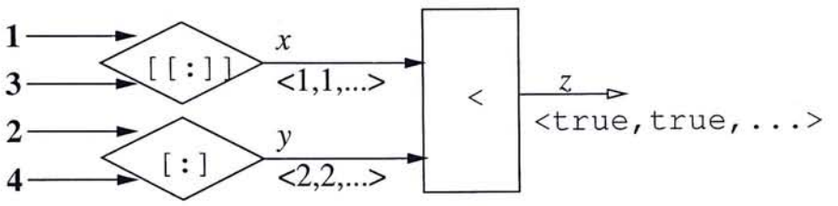


Figure 3.17: The valuation of the network in Figure 3.15 at stage 1, where $x[0] \mapsto 1$ and $y[0] \mapsto 2$.

At stage 1, as the range committed choice operator "[:]" had chosen "2" at stage 0, it has to take "2" again. The range choice operator "[[:]]" picks "1" so that $z[1]$ can take "`true`". We then get $x[1] \mapsto 1$, $y[1] \mapsto 2$, and $z[1] \mapsto$ `true` as shown in Figure 3.17. The process continues, and we have the solution $\bigcup_{t \in \mathcal{Z} \geq 0} \{x[t] \mapsto 1, y[t] \mapsto 2, z[t] \mapsto \texttt{true}\}$.

If we have $\{x[0] \mapsto 1, y[0] \mapsto 3, z[0] \mapsto \texttt{true}\}$ or $\{x[0] \mapsto 2, y[0] \mapsto 3, z[0] \mapsto \texttt{true}\}$ instead, other possible solutions will be generated. At stage 1, "3" is chosen by the range committed choice operator "[:]" from the same arc chosen last stage. The daton "1" or "2" can be chosen by the range choice operator "[[:]]" so that $z[1]$ can be "`true`" as shown in Figure 3.18. We then get a set of solutions $\{\bigcup_{t \in \mathcal{Z} \geq 0} \{x[t] \mapsto k_t, y[t] \mapsto 3, z[t] \mapsto \texttt{true}\} \mid k_i = 1 \vee k_i = 2, i \in \mathcal{Z} \geq 0\}$.

Moreover, if we have $\{x[0] \mapsto 1, y[0] \mapsto 4\}$, $\{x[0] \mapsto 2, y[0] \mapsto 4\}$, or $\{x[0] \mapsto 3, y[0] \mapsto 4\}$, other possible solutions will be generated. At stage 1, "4" is chosen by the range committed choice operator "[:]" from the same arc chosen last stage. The daton "1", "2", or "3" is chosen by the range choice operator "[[:]]" so that $z[1]$ is possible to take "`true`" as shown in Figure 3.19. We then get a set of solutions $\{\bigcup_{t \in \mathcal{Z} \geq 0} \{x[t] \mapsto k_t, y[t] \mapsto 4, z[t] \mapsto \texttt{true}\} \mid k_i = 1 \vee k_i = 2 \vee k_i = 3, i \in \mathcal{Z} \geq 0\}$.

□

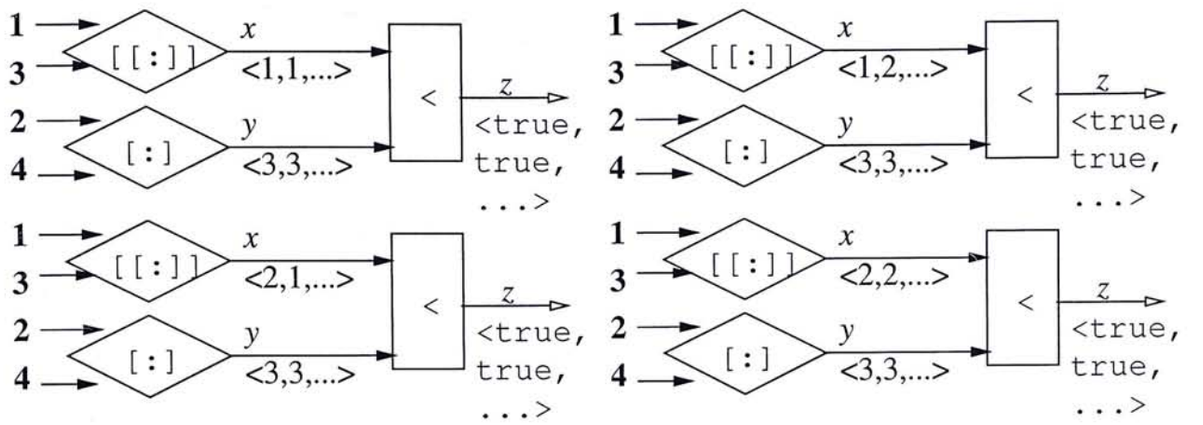


Figure 3.18: The valuations of the network in Figure 3.15 at stage 1, where $y[0] \mapsto 3$.

A classical Dataflow Network must have exactly one solution, but an Extended Dataflow Network can have more than one solution or no solutions. The assertion arcs of an Extended Dataflow Network restricts the number of choices in firing. On the other hand, the non-deterministic operators allow selection of datons and datastreams, resulting in possibly greater number of solutions.

3.4 E-Lucid

This section presents *E-Lucid*, which is a language for specifying and executing Extended Dataflow Networks. E-Lucid augments Lucid with the addition of the selection operators and assertion variables.

The selection operators of E-Lucid correspond exactly to those of Extended Dataflow Networks. The operand datastreams of the discrete choice operator and the discrete committed choice operator are separated by "," and embraced by "[[]]" and "[]" respectively. The operand datastreams of the range choice operators are separated by ":" and embraced by "[[]]", "{[]]", "[[}}", or "{{}}". The operand datastreams of the range committed choice operators are separated by ":" and embraced by "[]", "{]", "[}", or "{}".

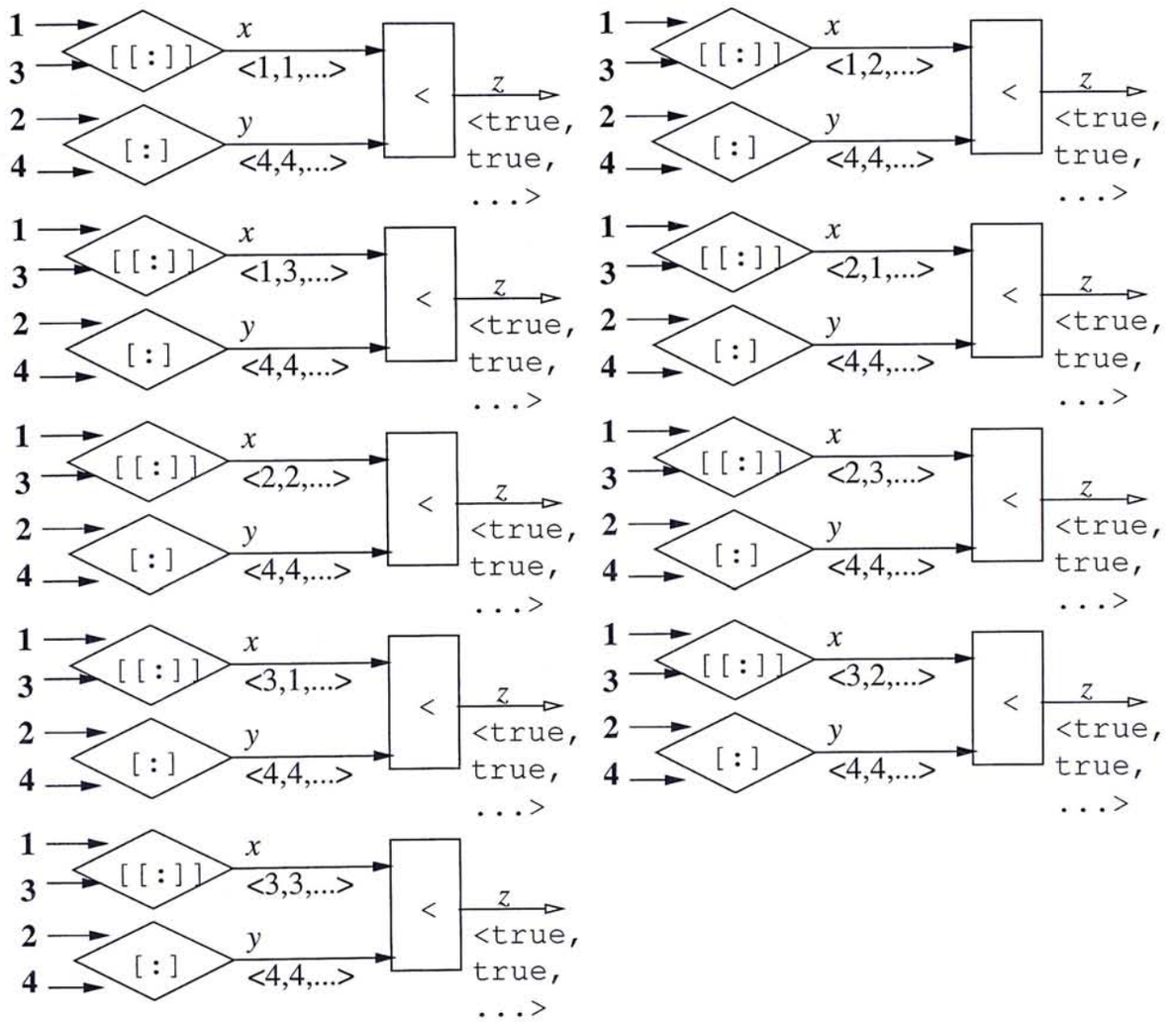


Figure 3.19: The valuations of the network in Figure 3.15 at stage 1, where $y[0] \mapsto 4$.

The statement on the left hand side of "⇒" can be rewritten into the statements on its right hand side to eliminate the committed choice operators with the meaning preserved. The terms in italic fonts represent the following: v is a variable, e_i is an expression, k_i is a constant, and *first_v* is a newly created variable.

v = [$e_1, ..., e_n$] ; ⇒ *first_v* = [[$e_1, ..., e_n$]] ; (3.1)
v = *first_v* fby
if (first *v* eq first e_1) then next e_1 else
⋮
if (first *v* eq first e_n) then next e_n else
false fi...fi;

v = [$k_1 : k_2$] ; ⇒ *first_v* = [[$k_1 : k_2$]] ; (3.2)
v = first *first_v*;

v = [$k_1 : k_2$}; ⇒ *first_v* = [[$k_1 : k_2$}}; (3.3)
v = first *first_v*;

v = {$k_1 : k_2$] ; ⇒ *first_v* = {{$k_1 : k_2$]]; (3.4)
v = first *first_v*;

v = {$k_1 : k_2$} ; ⇒ *first_v* = {{$k_1 : k_2$}}; (3.5)
v = first *first_v*;

Figure 3.20: Elimination of committed choice operators

The choice operators are built-in operators. The committed choice operators can be defined in terms of the classical Lucid operators and the choice operators as in Rules 3.1 to 3.5.

Assertion variables of E-Lucid implement assertion arcs of Extended Dataflow Networks. To declare a variable v to be an assertion variable, a keyword `assert` is added at the beginning of the defining statement of v. An assertion variable is *satisfied* if and only if it takes the value **true**. We code the networks of Figures 3.9 and 3.15 in E-Lucid as in Examples 3.3 and 3.4 respectively to illustrate the idea.

Example 3.3

```
x = 1 fby [1,2] + x;
y = 3 fby [[1,2]] + y;
assert z = (y - x eq 2);
```

□

Example 3.4

```
x = [[1 : 3]];
y = [4 : 6];
assert z = (x + y eq 7);
```

□

We give a formal semantics to E-Lucid programs. An E-Lucid program $\mathcal{E}$ is a tuple $\langle V, V_A, E \rangle$, in which V is a set of variables; E is a set containing, for every $v \in V$, an associated expression E_v; The set V_A is a set of assertion variables, where $V_A \subseteq V$. Given a valuation θ over V and an expression e, $\theta(e)$ is obtained by replacing variables in e by their corresponding values. θ is a *solution* of $\mathcal{E}$ if $\theta(v) = \theta(E_v)$ holds for all $v \in V$ and $\{v[t] \mapsto \texttt{true} \mid v \in V_A, t \in \mathcal{Z} \geq 0\} \subseteq \theta$.

Some examples will be used to demonstrate the possible problems that can be modelled and solved by E-Lucid.

3.4.1 Modified Four Cockroaches Problem

We modify the Four Cockroaches Problem in Section 2.3.7, so that we know only the coordinates of cockroaches after one time unit from the initial time. We show how this can be specified in E-Lucid and the corresponding Extended Dataflow Network.

There are four cockroaches at the four corners of a square, one facing another in a cyclic manner in the anti-clockwise direction, and their start x and y coordinates lie between 0 to 10. Their coordinates after one time unit are [0.01, 0.0], [10.0, 0.01], [9.99, 10.0], [0.0, 9.99] respectively. Each cockroach walks after the next cockroach in anti-clockwise direction, and they can move distance unit d in each time unit. We would like to find out the traces of the four cockroaches.

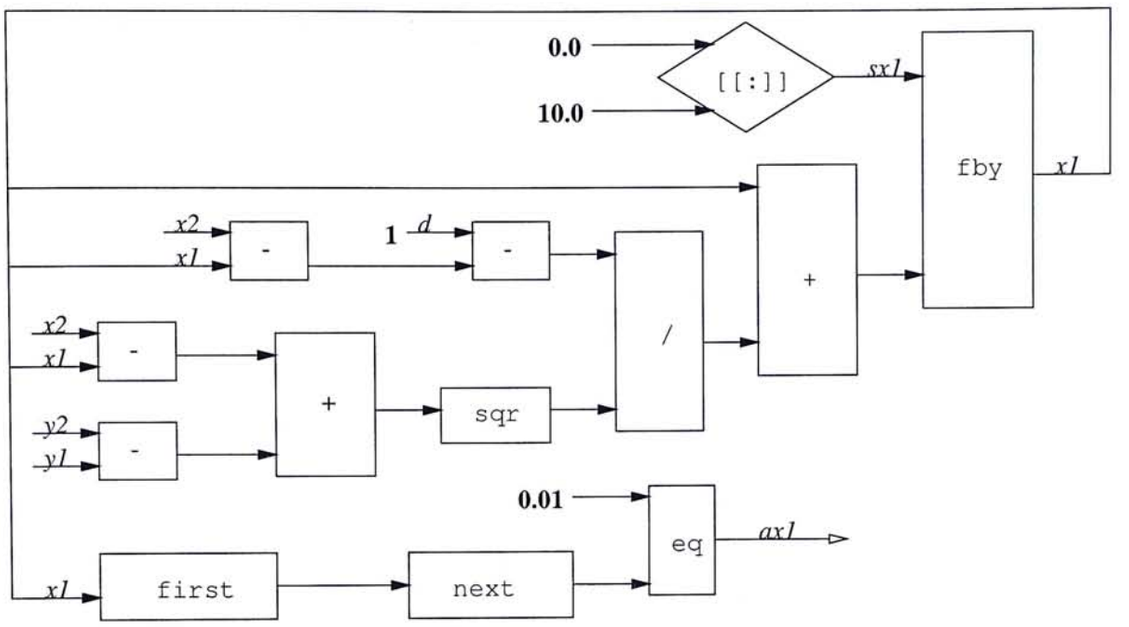


Figure 3.21: Part of the Extended Dataflow Network of the Modified Four Cockroaches Problem

We model the problem with the following E-Lucid program, where the datastreams xi and yi with real domains represent the x and y coordinates of the cockroach i, sxi and syi represent the start x and y coordinates of the cockroach i. The statements marked with "*"s at the end are represented in Figure 3.21 graphically, which form part of the Extended Dataflow Network of the problem.

Step distance d is defined as:

```
d = 0.01;*
```

The coordinates are defined as:

```
x1 = sx1 fby x1 + d*(x2 - x1) / sqr(sq(x2 - x1) + sq(y2 - y1));*
y1 = sy1 fby y1 + d*(y2 - y1) / sqr(sq(x2 - x1) + sq(y2 - y1));
x2 = sx2 fby x2 + d*(x3 - x2) / sqr(sq(x3 - x2) + sq(y3 - y2));
y2 = sy2 fby y2 + d*(y3 - y2) / sqr(sq(x3 - x2) + sq(y3 - y2));
x3 = sx3 fby x3 + d*(x4 - x3) / sqr(sq(x4 - x3) + sq(y4 - y3));
```

```
y3 = sy3 fby y3 + d*(y4 - y3) / sqr(sq(x4 - x3) + sq(y4 - y3));
x4 = sx4 fby x4 + d*(x1 - x4) / sqr(sq(x1 - x4) + sq(y1 - y4));
y4 = sy4 fby y4 + d*(y1 - y4) / sqr(sq(x1 - x4) + sq(y1 - y4));
```

The start coordinates are defined as follows. As only the first datons are considered, either the range choice operator or the range committed choice operator can be used. The range choice operator is chosen here:

```
sx1 = [[0.0 :  10.0]];*          sx3 = [[0.0 :  10.0]];
sy1 = [[0.0 :  10.0]];           sy3 = [[0.0 :  10.0]];
sx2 = [[0.0 :  10.0]];           sx4 = [[0.0 :  10.0]];
sy2 = [[0.0 :  10.0]];           sy4 = [[0.0 :  10.0]];
```

The coordinates after one time unit are defined as:

```
assert ax1 = (first next x1 eq 0.01);*
assert ay1 = (first next y1 eq 0.0);
assert ax2 = (first next x1 eq 10.0);
assert ay2 = (first next y1 eq 0.01);
assert ax3 = (first next x1 eq 9.99);
assert ay3 = (first next y1 eq 10.0);
assert ax4 = (first next x1 eq 0.0);
assert ay4 = (first next y1 eq 9.99);
```

The solution is the same as that of the original Four Cockroaches Problem in Section 2.3.7, with start coordinates $sx1[0] \mapsto 0.0$, $sy1[0] \mapsto 0.0$, $ax2[0] \mapsto 10.0$, $ay2[0] \mapsto 0.0$, $ax3[0] \mapsto 10.0$, $ay3[0] \mapsto 10.0$, $ax4[0] \mapsto 0.0$, and $ay4[0] \mapsto 10.0$.

In this problem, the coordinates of cockroaches after one time unit is given. Similarly, given the coordinates of cockroaches at any time unit t_0, we are able to find the coordinates at all indices t by modelling the problem in E-Lucid, even if $t < t_0$. Lucid is only able to find the coordinates with indices $t \geq t_0$.

We call this ability to find the values of datons by given the values of datons of larger indices as *backward valuation*, which is an advantage of E-Lucid over Lucid.

3.4.2 Traffic Light Problem

The Traffic Light Problem [14] is defined as follows. In a 4-way traffic junction (in Germany) with 8 traffic lights, where 4 lights ($V1$, $V2$, $V3$, $V4$) for vehicles and 4 lights ($P1$, $P2$, $P3$, $P4$) for pedestrians as shown in Figure 3.22. The vehicle lights show colors in order of red (1), red-yellow (2), green (3) and yellow (4), and the pedestrian lights show colors in order of red (1) and green (3). Given integers i and j, where $1 \leq i \leq 4$ and $j = (1 + i)$ `mod` 4, the only possible signal combinations of Vi, Pi, Vj, and Pj are (1,1,3,3), (2,1,4,1), (3,3,1,1), and (4,1,2,1) as shown in Figure 3.23, so that the vehicles do not collide head-on, and the pedestrians and the vehicles do not cross the same road simultaneously.

To model the problem, the variable $ColorV$ is used to defined the order of light signals:

```
ColorV = 1 fby 2 fby 3 fby 4 fby ColorV;
```

Then the 8 traffic lights are defined using the discrete choice operator and the discrete committed choice operator as:

```
V1 = [ColorV, next ColorV, next next ColorV, next next next ColorV];
P1 = [[1, 3]];
V2 = [ColorV, next ColorV, next next ColorV, next next next ColorV];
P2 = [[1, 3]];
V3 = [ColorV, next ColorV, next next ColorV, next next next ColorV];
P3 = [[1, 3]];
V4 = [ColorV, next ColorV, next next ColorV, next next next ColorV];
P4 = [[1, 3]];
```

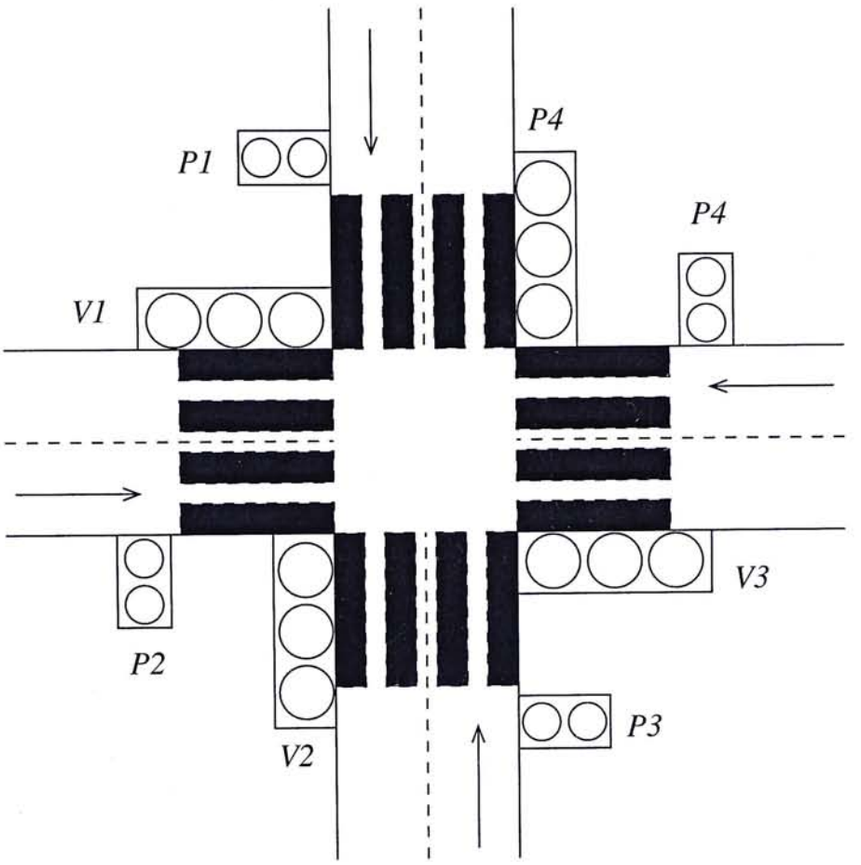


Figure 3.22: A 4-way traffic junction in Germany

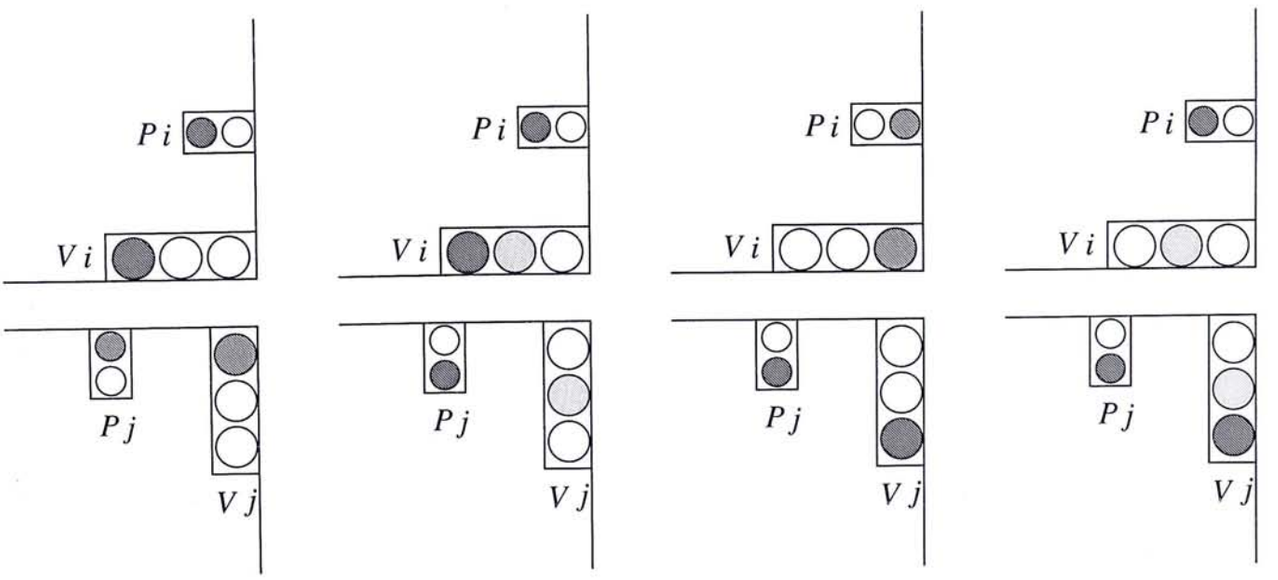


Figure 3.23: The four possible signal combinations of adjacent traffic lights

The color combinations of Vi, Pi, Vj, and Pj can be expressed by using assertion variables to enforce safety constraints as follows:

```
assert L12 = (V1 eq 1 and P1 eq 1 and V2 eq 3 and P2 eq 3)
          or (V1 eq 2 and P1 eq 1 and V2 eq 4 and P2 eq 1)
          or (V1 eq 3 and P1 eq 3 and V2 eq 1 and P2 eq 1)
          or (V1 eq 4 and P1 eq 1 and V2 eq 2 and P2 eq 1);
assert L23 = (V2 eq 1 and P2 eq 1 and V3 eq 3 and P3 eq 3)
          or (V2 eq 2 and P2 eq 1 and V3 eq 4 and P3 eq 1)
          or (V2 eq 3 and P2 eq 3 and V3 eq 1 and P3 eq 1)
          or (V2 eq 4 and P2 eq 1 and V3 eq 2 and P3 eq 1);
assert L34 = (V3 eq 1 and P3 eq 1 and V4 eq 3 and P4 eq 3)
          or (V3 eq 2 and P3 eq 1 and V4 eq 4 and P4 eq 1)
          or (V3 eq 3 and P3 eq 3 and V4 eq 1 and P4 eq 1)
          or (V3 eq 4 and P3 eq 1 and V4 eq 2 and P4 eq 1);
assert L41 = (V4 eq 1 and P4 eq 1 and V1 eq 3 and P1 eq 3)
          or (V4 eq 2 and P4 eq 1 and V1 eq 4 and P1 eq 1)
          or (V4 eq 3 and P4 eq 3 and V1 eq 1 and P1 eq 1)
          or (V4 eq 4 and P4 eq 1 and V1 eq 2 and P1 eq 1);
```

There are four possible solutions with different initial color combinations as shown in Figure 3.24. The set of the solutions is:

$\{\bigcup_{t\in Z\geq 0}\{ V1[4t+i \bmod 4] \mapsto 2, P1[4t+i \bmod 4] \mapsto 1,$
$V2[4t+i \bmod 4] \mapsto 4, V2[4t+i \bmod 4] \mapsto 1,$
$V3[4t+i \bmod 4] \mapsto 2, P3[4t+i \bmod 4] \mapsto 1,$
$V4[4t+i \bmod 4] \mapsto 4, V4[4t+i \bmod 4] \mapsto 1,$
$V1[4t+(i+1) \bmod 4] \mapsto 3, P1[4t+(i+1) \bmod 4] \mapsto 3,$
$V2[4t+(i+1) \bmod 4] \mapsto 1, V2[4t+(i+1) \bmod 4] \mapsto 1,$
$V3[4t+(i+1) \bmod 4] \mapsto 3, P3[4t+(i+1) \bmod 4] \mapsto 3,$

$V4[4t + (i+1) \texttt{ mod } 4] \mapsto 1, V4[4t + (i+1) \texttt{ mod } 4] \mapsto 1,$
$V1[4t + (i+2) \texttt{ mod } 4] \mapsto 4, P1[4t + (i+2) \texttt{ mod } 4] \mapsto 1,$
$V2[4t + (i+2) \texttt{ mod } 4] \mapsto 2, V2[4t + (i+2) \texttt{ mod } 4] \mapsto 1,$
$V3[4t + (i+2) \texttt{ mod } 4] \mapsto 4, P3[4t + (i+2) \texttt{ mod } 4] \mapsto 1,$
$V4[4t + (i+2) \texttt{ mod } 4] \mapsto 2, V4[4t + (i+2) \texttt{ mod } 4] \mapsto 1,$
$V1[4t + (i+3) \texttt{ mod } 4] \mapsto 1, P1[4t + (i+3) \texttt{ mod } 4] \mapsto 1,$
$V2[4t + (i+3) \texttt{ mod } 4] \mapsto 3, V2[4t + (i+3) \texttt{ mod } 4] \mapsto 3,$
$V3[4t + (i+3) \texttt{ mod } 4] \mapsto 1, P3[4t + (i+3) \texttt{ mod } 4] \mapsto 1,$
$V4[4t + (i+3) \texttt{ mod } 4] \mapsto 3, V4[4t + (i+3) \texttt{ mod } 4] \mapsto 3,$
$\}|i \in [0,1,2,3]\}$

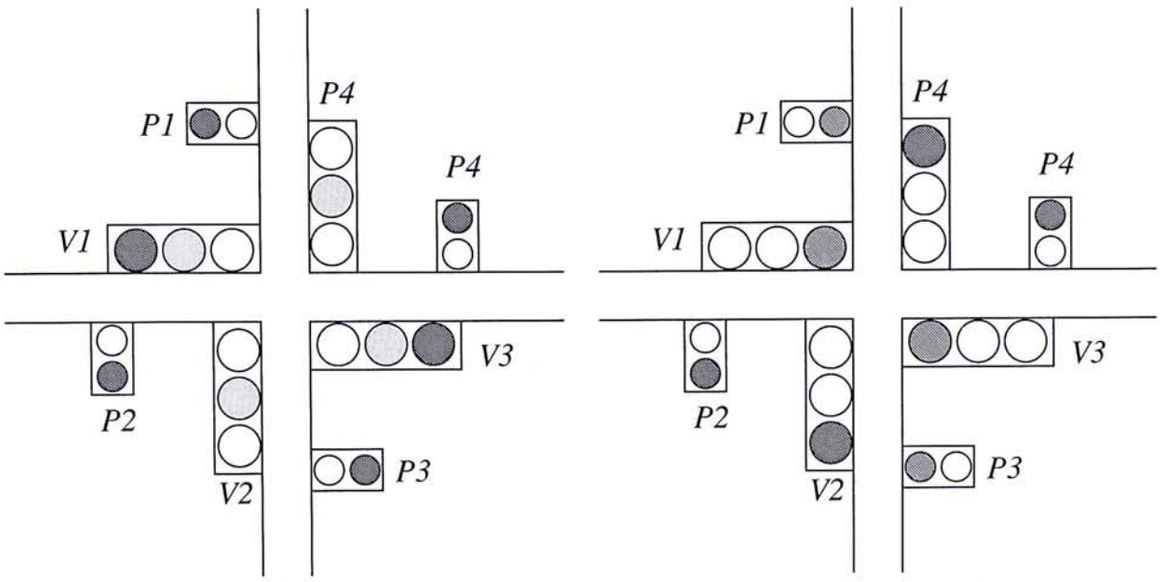
Classical Lucid operators is only able to express the order of the traffic light colors, while the interrelationship among the colors of the 8 traffic lights at different time units mush be expressed with the selection operators and the assertion variables. Moreover, E-Lucid is able to find all the four solutions. This example illustrates that E-Lucid is able to model problems with interrelationship among variables and find more than one solution.

3.4.3 Old Maid Problem

Old Maid [32], or "cim wu gwai" in Cantonese, is a card game which can be played by two or more players. From a standard 52 cards pack, remove one queen leaving 51 cards. Deal and play in clockwise fashion.

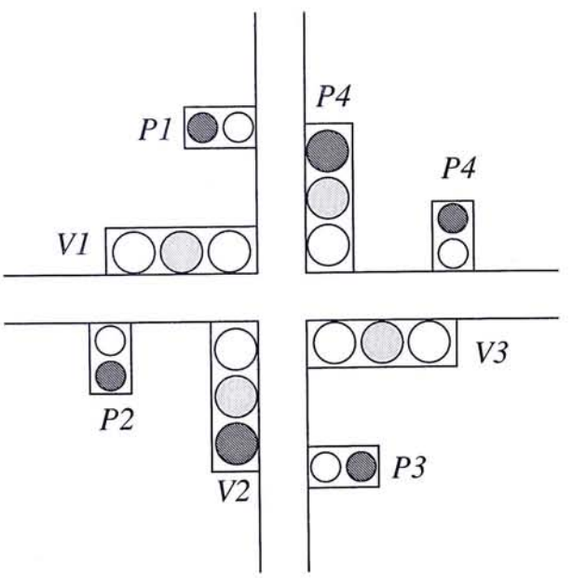
The dealer deals out all the cards to the players (generally some will have one more card than others - this does not matter). The players all look at their cards and discard all pairs they have (a pair is two cards of equal rank, such as two sevens or two kings).

The dealer begins. At your turn you must offer your cards spread face down to the player to your left. That player selects a card from your hand

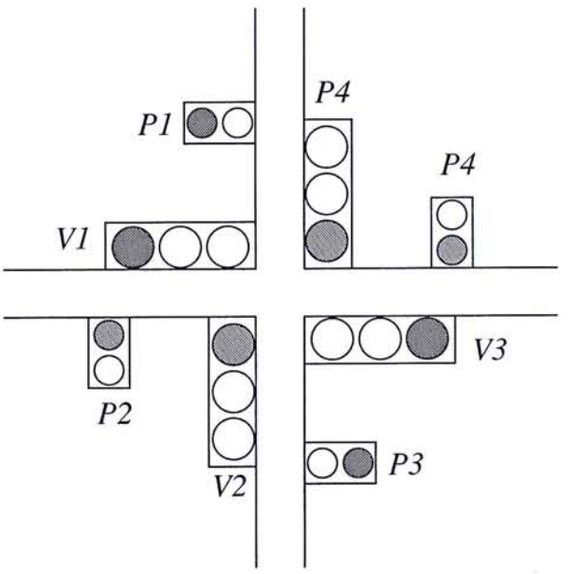


At stage $4t + i$ `mod` 4

At stage $4t + (i + 1)$ `mod` 4



At stage $4t + (i + 2)$ `mod` 4



At stage $4t + (i + 3)$ `mod` 4

Figure 3.24: The possible solutions of the traffic light problem, where $t \in \mathcal{Z} \geq 0$, $i \in \{0, 1, 2, 3\}$

without seeing it, and adds it to her hand. If it makes a pair in her hand she discards the pair. The player who just took a card then offers her hand to the next player to her left, and so on.

If you get rid of all your cards you are safe - the turn passes to the next player and you take no further part. Eventually all the cards will have been discarded except one queen (the old maid) and the holder of this queen loses.

We can model the game with an E-Lucid program, solutions of which are all possible plays. For simplicity, players do not discard any card at the first round. The assumption can be relaxed by adding more statements with `if-then-else-fi`.

We use constants 1, 2, 3, 4 to represent the 4 players. The variable *seq* represents the order of play. For example, if $seq[t]$ is equal to 3, the t^{th} turn is Player 3's turn. The variable is defined as:

```
seq = 1 fby 2 fby 3 fby 4 fby seq;
```

Variables *dA*, *d2*, ..., *dK*, *cA*, *c2*, ..., *cK*, *hA*, *h2*, ..., *hK*, *sA*, *s2*, ..., *sK* represent the holders of the cards. For example, if *d2* is equal to ⟨ 3, 3, 4, 1, 1, `eod`, ... ⟩, the card diamond 2 is held by player 3 for two turns and passed to player 4 and player 1 in the next term. The later keeps it for one turn and then discards it. Except the Queens, the rules can be represented by:

```
dA = [seq, next seq, next next seq, next next next seq] upon pass_dA;
⋮
sK = [seq, next seq, next next seq, next next next seq] upon pass_sK;
```

Similarly, the Queens are defined in similar way, except that they may take `eod` at the first turn:

```
dQ = [seq, next seq, next next seq, next next next seq, eod]
     upon pass_dQ;
⋮
```

```
sQ = [eod, seq, next seq, next next seq, next next next seq]
     upon pass_sQ;
```

One Queen is discarded at the beginning, therefore one of `dQ`, `cQ`, `hQ`, and `sQ` is equal to **eod**:

```
assert discardQ = (dQeod + cQeod + hQeod + sQeod eq 1);
dQeod = if iseod(dQ) then 1 else 0 fi;
⋮
sQeod = if iseod(sQ) then 1 else 0 fi;
```

Variables *pass_dA*, *pass_d2*, ..., *pass_sK* represent the status of the corresponding cards. Value `true` means passing, value `false` means keeping, and value `eod` means discarding. For example, if *pass_d2* is equal to ⟨ `false`, `true`, `true`, `false`, `eod`,...⟩, the card diamond 2 is kept for one turn, which is passed to next two players in the subsequent two turns. It is kept for one turn and discarded in the next turn.

One card is discarded if it makes pairs with another card of the players, otherwise it may be kept or passed. For example, if *dA* is equal to one of *cA*, *hA*, or *sA*, diamond A makes a pair with one of them and is discarded. Otherwise *dA* is equal to `true` or `false`. The rule is represented by:

```
pass_dA = if (dA eq cA or dA eq hA or dA eq sA) then
          eod else [[true, false]];
⋮
pass_sK = if (sK eq dA or sK eq cA or sK eq hA) then
          eod else [[true, false]];
```

Exactly one card is passed to next player at each time, therefore exactly one of *pass_dA*, ..., *pass_sK* is `true`:

```
assert onepass = (numpass_dA + ...  + numpass_sK eq 1);
```

```
numpass_dA = if pass_dA then 1 else 0 fi;
⋮
numpass_sK = if pass_sK then 1 else 0 fi;
```

A player can pass his cards only in his turns, therefore *pass_dA* is equal to `true` only if *dA* is equal to *seq*:

```
assert holderpass_dA = (not pass_dA or (dA eq seq));
⋮
assert holderpass_sK = (not pass_dA or (dA eq seq));
```

Taking diamond A as an example, the following values of the variables on the right hand side indicates that the diamond A is passed from player 1 to player 2 at the first turn:

$$\begin{aligned} dA &= \langle\ 1, & 2, & 2, & 2, & 2, & \ldots\rangle \\ pass\_dA &= \langle\ \texttt{true}, & \texttt{false}, & \texttt{false}, & \texttt{false}, & \ldots\rangle & \\ seq &= \langle\ 1, & 2, & 3, & 4, & 1, & \ldots\rangle \end{aligned}$$

While the following values of variables are impossible, as player 1 cannot pass his diamond A at the turn of player 2:

$$\begin{aligned} dA &= \langle\ 1, & 2, & 2, & 2, & 2, & \ldots\rangle \\ pass\_dA &= \langle\ \texttt{false}, & \texttt{true}, & \texttt{false}, & \texttt{false}, & \ldots\rangle & \\ seq &= \langle\ 1, & 2, & 3, & 4, & 1, & \ldots\rangle \end{aligned}$$

When the game ends, only one Queen is kept by one player. All the other cards are discarded and other players have no more cards. Therefore, datons of $dA, \ldots, sK$ will take `eod` finally, except the variable representing the "Old Maid" will be equal to the ID of the loser. In each solution, the values of $dA, \ldots, sK$ represent a possible play. Therefore this problem has huge numbers of solutions.

The player sequence of the cards can be expressed with the classical Lucid

operators, while the rules such as "only one card can be passed each time" and "a pair should be discarded" mush be expressed with the selection operators and the assertion variables. This example has illustrated that E-Lucid is able to express complicated rules.

Chapter 4

Implementation of E-Lucid

In this chapter, we introduce our prototype E-Lucid interpreter by showing and discussing the pseudo-code of the functions of the interpreter with examples. Section 4.1 gives an overview of the interpreter. Section 4.2 defines new terms used in this chapter. Sections 4.3 to 4.8 introduce the functions used in the implementation.

4.1 Overview

The prototype E-Lucid interpreter is an extension of the pLucid interpreter. Different from the pLucid interpreter, the valuation obtained may not be a partial solution.

Figure 4.1 is an example of Extended Dataflow Network. The value of daton $x[0]$ is 1, which is less than 3. Thus, $z[0]$ takes the value `true` and we obtain the valuation $\{x[0] \mapsto 1, z[0] \mapsto \texttt{true}\}$. The value of daton $x[1]$ is 2, which is less than 3. Thus $z[1]$ takes the value `true` and we obtain the valuation $\{x[0] \mapsto 1, z[0] \mapsto \texttt{true}, x[1] \mapsto 2, z[1] \mapsto \texttt{true}\}$. The value of daton $x[2]$ is 3. Thus $z[2]$ takes the value `false`. As the assertion variable is not satisfied, there is no solution.

The interpreter does not realize that the problem has no solution until $z[2]$ is assigned a value. If we modify the problem by replacing the datastream **3** with

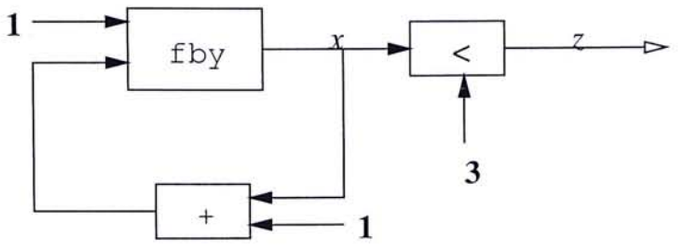


Figure 4.1: An Extended Dataflow Network

10000, the interpreter will not realize that the problem has no solution until $z[9999]$ is assigned a value. Generally speaking, given an E-Lucid program with at least one assertion variable, the interpreter cannot ensure that the program has solution or not, even if the first n datons of assertion variables are `true`, given that n is a finite number.

In this and all other examples of Extended Dataflow Network, we use the phrase "at stage t" ($t > 0$) to represent the stage after all variables of interest and assertion variables at index $t-1$ are assigned values and some of those at index t are not assigned values yet. "At stage 0" represents the stage before all variables of interest and assertion variables at index 0 are assigned values.

At stage t, if all daton with index $< t$ of assertion variables pick the value `true`, the valuation ζ obtained is called *the solution up to stage* t, which *may be* a partial solution. In the example, $\{x[0] \mapsto 1,\ z[0] \mapsto \texttt{true}\}$ is the solution up to stage 0, while $\{x[0] \mapsto 1,\ z[0] \mapsto \texttt{true},\ x[1] \mapsto 2,\ z[1] \mapsto \texttt{true}\}$ is the solution up to stage 1. However, they are not partial solutions as there is no solution.

`ELUCIDinterpreter` is the main function of the prototype E-Lucid interpreter, which is an extension of `LUCIDinterpreter` in Figure 2.8. `ELUCIDinterpreter` invokes functions `transformD`, `Edemand`, `constructCSP`, and `backtrack`. Function `transformD` transforms statements to *daton statements*, which are statements expressed in terms of datons. Function `Edemand` is the modification

of `demand` in Figure 2.8 with additional codes for handling selection operators. It invokes `constructCSP` to construct CSPs, which is done by invoking `findC` and `transformD` to transform daton statements to constraints. Function `backtrack` performs backtracking to instruct selection operators to fire alternate datons and explore other possible partial solutions.

4.2 Definition of Terms

We define new terms for use in subsequent sections. The variables expressed in terms of datons are called *daton variables.* Expressions expressed in terms of daton variables are called *daton expressions.* In a daton statement s: $u = e$, s and e are the *defining daton statement* and *defining daton expression* of u respectively, while u is their *defined daton variable.* In an E-Lucid program, each daton variable has a unique defining daton statement and a unique defining daton expression. For example, in the daton statement s: $x[3] = 1 + y[2]$, s and $1 + y[2]$ are the defining daton statement and defining daton expression of $x[3]$ respectively, while $x[3]$ is their defined daton variable. We define $\texttt{sta}(u)$ as the defining daton statement of daton variable u. In the example, $\texttt{sta}(x[3])$ is s. We abuse terminology by saying that $\texttt{sta}(U)$ is the set of defining daton statements of daton variables in U.

We define the variables labelling output datastreams of selection operators to be *selection variables.* For example, in statement `x = [[3, y]]`, x is a selection variable. We define the daton variables labelling output datons of selection operators be *selection daton variables.* For example, in daton statement $x[1] = [[3, y[1]]]$, $x[1]$ is a selection daton variable. We define the daton variables of assertion variables be *assertion daton variables.*

Recall that $\mathcal{E}$ is defined as a tuple $\langle V, V_A, E \rangle$ previously. With the daton statements, we can redefine an E-Lucid program $\mathcal{E}$ as a tuple $\langle U, U_A, F \rangle$, in which $U = \{v[t] \mid v[t] \in \texttt{var}(V), t \in \mathcal{Z} \geq 0\}$ is an infinite set of daton

variables. F is a set containing, for every $v \in V$, an associated daton expression $F_{v[t]} = E_v[t]$. The set $U_A = \{v[t] \mid v \in V_A,\ t \in \mathcal{Z} \geq 0\}$ is an infinite set of assertion daton variables, where $U_A \subseteq U$. Given a valuation θ over U, it is a *solution* of $\mathcal{E}$ if $\theta(u) = F_u$ holds for all $u \in U$ and $\{u \mapsto \texttt{true} \mid u \in U_A\} \subseteq \theta$.

A valuation ζ is a *solution up to stage* t_0 if $\zeta(v[t]) = \zeta(F_{v[t]})$ holds for all $v[t] \in U$ where $t \leq t_0$ and $\{u[t] \mapsto \texttt{true} \mid u[t] \in U_A, t \leq t_0\} \subseteq \zeta$.

4.3 Function ELUCIDinterpreter

This section introduces the main function `ELUCIDinterpreter` of the prototype E-Lucid interpreter with pseudo-code shown in Figure 4.2. Firstly, it transforms statements to daton statements. The variables of interest and assertion variables are demanded by the function. If selection operators are demanded, some daton statements are transformed to constraints to construct CSPs, solutions of which determine the datons fired by the operators. If necessary, the system backtracks to find alternate partial solutions.

Compared to `LUCIDinterpreter` in Figure 2.8, `ELUCIDinterpreter` contains extra parameters and variables. V_A is the set of assertion variables. E is the set of statements. *Sol* is an array of sequences of solutions, where $Sol[t]$ denotes the sequence of solutions at stage t. As $Sol[t]$ may consist of more than one solution, *solIndex* is defined as an array of indices, where the $solIndex[t]^{th}$ solution, $Sol[t][solIndex[t]]$, is picked at stage t. We use a boolean flag *back* to indicate whether backtracking is required. For example, given that the solutions at stage 0 are $\{a \mapsto 0\}$ and $\{a \mapsto 1\}$ and at stage 1 are $\{b \mapsto 0\}$ and $\{b \mapsto 1\}$. If $solIndex[0]$ is 1, $Sol[0][solIndex[0]]$ is $\{a \mapsto 1\}$. If $solIndex[1]$ is 0, $Sol[1][solIndex[1]]$ is $\{b \mapsto 0\}$.

The numbered lines in Figure 4.2 show the main differences between the two interpreters. At Line 1, `transformD` is invoked to transform the set of

```
ELUCIDinterpreter (P, V_A, E);
begin
    t := 0;
    Sol[t] := {};
    U_A := {v[t] | v ∈ V_A};
1   S := transformD (E);
    repeat
        foreach v ∈ P ∪ V_A do
2           value := Edemand (v[t], t, S, U_A, Sol, solIndex, ζ, back);
            if back then exit;
            else print value;
        end
3       if the user requests backtracking or back == true then
4           t := backtrack (t, solIndex, Sol, ζ);
        else
            t := t + 1;
            Sol[t] := {};
        end
    until t ≤ 0;
end
```

Figure 4.2: Pseudo-code of `ELUCIDinterpreter`

statements to the set of daton statements S. At Line 2, variables of interest and assertion variables are demanded by the function. `Edemand` instead of `demand` is invoked to handle the demand-driven schema with selection operators and assertion variables. At Lines 3 and 4, `backtrack` is invoked to perform backtracking if the user requests it or *back* is set to `true` by `Edemand`.

4.4 Function Edemand

This section introduces function `Edemand` with pseudo-code shown in Figure 4.3. The function extends `demand` with additional abilities to handle selection operators and assertion variables. Compared to `demand`, `Edemand` has additional lines which are numbered and contains extra parameters which are

defined in the same way as in `ELUCIDinterpreter`.

Line 1 sets *back* to `true` to perform backtracking if it finds an unsatisfied assertion variable. The other numbered lines handle firings of selection operators. When a selection operator is demanded at stage t, Line 3 checks whether there was CSP constructed at stage t. If there was not, a CSP *aCSP* is constructed by invoking `constructCSP` at Line 4. The CSP is solved by ILOG Solver 4.4, which is a CSP solver. At Line 5, the solutions are projected to the selection daton variables and stored in $Sol[t]$. At Line 6, the solution counter $solIndex[t]$ is initialized to 0. At Line 7, if there is no solution, *back* is set to `true` to perform backtracking. At Line 9, the solution is applied to the output daton of *cvof* if *back* is `false`.

4.5 Function `transformD`

Function `transformD` transforms the input set of statements to a set of daton statements. In the pseudo-code of `transformD` shown in Figure 4.4, there are five transformation steps, which will be detailed in each of the following subsections.

4.5.1 Labelling Datastreams of Selection Operators

Each varaible labels a datastream. However, there are some *unlabelled datastreams* which are not labelled with any variable. For example, in `a = first(b + c)`, `(b + c)` is an unlabelled datastream. We can label it by assigning it a new variable n, so that we have the statements `a = first n` and `n = b + c`. Labelling unlabelled datastreams is a prepare step to transform statements to daton statements, we will use the technique in the subsequent subsections. As it is useless to label constant datastreams, they are not labelled in our implmentation. When we refer to "unlabelled datastream", the constant datastreams are excluded.

```
Edemand (cvof, t, S, U_A, Sol, solIndex, ζ, back)
begin
   if NOT back then
      if cvof is a daton variable then
         /* if cvof is already assigned a value*/
         if ∃value s.t. (cvof ↦ value) ∈ ζ then u := value;
         else
            driving := the constant / operator / variable driving cvof at
            stage t;
            u := Edemand (driving, t, S, U_A, Sol, solIndex, ζ, back);
            /* cvof is an assertion daton variable and it is assigned false
            */
1           if cvof ∈ U_A and u ==false then back := true;
         end
         ζ := ζ ∪ {cvof ↦ u};
      else if cvof is an operator with n operands then
         foreach operand r_i do
            value_i :=Edemand (r_i, t, S, U_A, Sol, solIndex, ζ, back);
            if back then
               return eod;
            end
         end
2        if cvof is a selection operator with selection daton variable d
         then
3           if Sol[t][solIndex[t]] == {} then
4              aCSP := constructCSP (t, S, U_A, ζ);
5              Sol[t] := the solutions of aCSP projecting on selection
               daton variables;
6              solIndex[t] := 0;
7              if |Sol[t]| == 0 then back := true;
8           /* apply the valuation of d in Sol[t][solIndex[t]] to u*/
9           if NOT back then u := Sol[t][solIndex[t]](d);
         else u := evaluation of cvof(value_1, ..., value_n);
      else if cvof is a constant then u := cvof;
      return u;
   end
end
```

Figure 4.3: Pseudo-code of Edemand

```
transformD (E)
begin
    S := E;
    foreach s ∈ S do
        labelling datastreams of selection operators;
    end
    foreach s ∈ S do
        removing committed choice operators;
    end
    foreach s ∈ S do
        removing asa, wvr, and upon;
    end
    foreach s ∈ S do
        labelling datastreams of if-then-else-fi;
    end
    foreach s ∈ S do
        transforming statements to daton statements;
    end
    foreach s ∈ S do
        transforming daton expressions recrusively;
    end
end
```

Figure 4.4: Pseudo-code of `transformD`

For each $s \in S$, if s consists of unlabelled output datastreams of selection operators, we label them with new selection variables. It is done by creating a variable for each of the unlabelled datastreams with its defining statement s_1, the expression of which is the selection operator with its corresponding operand datastreams. We create one more statement s_2, which is the same as s except the selection operator is replaced by the new variable. The statement s in S is replaced by s_1 and s_2. For example, the statement:

s: `x = 1 + [7, 8];`

is replaced by the following statements with new variable n:

s_1: `n = [7, 8];` s_2: `x = 1 + n;`

4.5.2 Removing Committed Choice Operators

For each $s \in S$, if s consists of committed choice operators, s is replaced by two statements by Rules 3.1 to 3.5 to eliminate the operators. The two statements together are equivalent to s. For example, the statement:

```
n = [x, y, z];
```

is replaced by the following statements by Rule 3.1:

```
first_n = [[x, y, z]];
n = first_n fby if (first n eq first x) then next x else
                if (first n eq first y) then next y else
                if (first n eq first z) then next z else
                eod fi fi fi;
```

4.5.3 Removing `asa`, `wvr`, and `upon`

For each $s \in S$, if s consists of operators `asa`, `wvr`, or `upon`, s is replaced by equivalent statements by Rules 2.1, 2.2, and 2.3 to eliminate the operators.

4.5.4 Labelling Output Datastreams of `if-then-else-fi`

For each $s \in S$, if s consists of unlabelled output datastreams of `if-then-else-fi`, we label them with new variables. It is done by creating a variable for each of the unlabelled datastreams with its defining statement s_1, the expression of which is `if-then-else-fi` with its corresponding operand datastreams. We create one more statement s_2, which is the same as s except `if-then-else-fi` is replaced by the new variable. The statement s in S is replaced by s_1 and s_2. For example, the statement:

s: `x = if y then 1 else if z then a else 7 fi fi;`

is replaced by the following statements with new variable n:

s_1: `n = if z then a else 7 fi;` s_2: `x = if y then 1 else n fi;`

4.5.5 Transforming Statements to Daton Statements

For each $s \in S$, its defined variable and expression are transformed to daton variables and daton expressions by Rule 4.1. In the following example, Statement 4.9 is transformed into Daton Statements 4.10 in Example 4.1.

Example 4.1 Given variables a, b, c, and d. Statement 4.9 is transformed to Daton Statement 4.11.

$$\texttt{a} = \texttt{next (b fby (first (c + d)))} \tag{4.9}$$

$$\Rightarrow a[t] = (\texttt{next}(b\ \texttt{fby}(\texttt{first}(c+d))))[t]\ \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.1} \tag{4.10}$$

$$\Rightarrow a[t] = (b\ \texttt{fby}(\texttt{first}\ (c+d)))[t+1]\ \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.4}$$

$$\Rightarrow a[t] = (\texttt{first}\ (c+d))[t]\ \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.5}$$

$$\Rightarrow a[t] = (c+d)[0]\ \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.3}$$

$$\Rightarrow a[t] = c[0] + d[0]\ \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.2} \tag{4.11}$$

□

Transformation Rules

The Rule 4.1 is derived from the definition of daton statement. The Rules 4.2 to 4.7 are directly derived from the definition of the operators. The (daton) expressions on the left hand side of the arrows can be transformed into the daton expressions on the right hand side, where e_i represents an expression, and PT represents an n-ary pointwise operator (*e.g.*, +, -, and, eq).

$$e_1 \Rightarrow e_1[t]\ \forall t \in \mathcal{Z} \geq 0 \tag{4.1}$$

$$(\mathtt{PT}(e_1, ..., e_n))[t] \Rightarrow \mathtt{PT}(e_1[t], ..., e_n[t])\forall t \in \mathcal{Z} \geq 0 \tag{4.2}$$

$$(\mathtt{first}\ e_1)[t] \Rightarrow e_1[0]\ \forall t \in \mathcal{Z} \geq 0 \tag{4.3}$$

$$(\mathtt{next}\ e_1)[t] \Rightarrow e_1[t+1]\ \forall t \in \mathcal{Z} \geq 0 \tag{4.4}$$

$$(e_1\ \mathtt{fby}\ e_2)[t] \Rightarrow \begin{cases} e_1[0] & \mathtt{if}\ t = 0 \\ e_2[t-1] & \mathtt{if}\ t > 0 \end{cases} \tag{4.5}$$

$$(e_1\ \mathtt{attime}\ e_2)[t] \Rightarrow e_1[e_2[t]]\ \forall t \in \mathcal{Z} \geq 0 \tag{4.6}$$

$$(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3\ \mathtt{fi})[t] \Rightarrow \mathtt{if}\ e_1[t]\ \mathtt{then}\ e_2[t]\ \mathtt{else}\ e_3[t]\ \mathtt{fi} \forall t \in \mathcal{Z} \geq 0 \tag{4.7}$$

$$e_1[t] = [[e_2, e_3]][t] \Rightarrow ((e_1[t] = e_2[t])\ \mathtt{or}\ (e_1[t] = e_3[t]))\ \forall t \in \mathcal{Z} \geq 0 \tag{4.8}$$

4.5.6 Transforming Daton Expressions Recursively

For each $s \in S$, s is in the form of $u = e$. Depends on the operator of e, e is transformed into another daton expression by one of the Rules 4.2 to 4.7. The arguments of e are transformed by the rules recursively until all the arguments are daton variable or constants. For example, Statement 4.10 is transformed into the set of Daton Statements 4.11 in Example 4.1.

4.5.7 An Example

Given the following statement, we transform it to daton statements by the algorithm of `transformD`.

```
a = [b, c] asa d;
```

Step 1: Labelling datastreams of selection operators

```
n1 = [b, c];
a = n1 asa d;
```

Step 2: Removing committed choice operators

```
first_n1 = [[b, c]];
n1 = first_n1 fby if (first n1 eq first a) then next a else
                 if (first n1 eq first b) then next b else
                 eod fi fi;
a = n1 asa d;
```

Step 3: Removing `asa`, `wvr`, and `upon`

```
first_n1 = [[b, c]];
n1 = first_n1 fby if (first n1 eq first b) then next b else
                 if (first n1 eq first c) then next c else
                 eod fi fi;
```

```
a = first(if d then n1 else next a fi);
```

Step 4: Removing `if-then-else-fi`

```
first_n1 = [[b, c]];
n1 = first_n1 fby n2;
n2 = if n3 then n4 else n5 fi;
n3 = first n1 eq first b;
n4 = next b;
n5 = if n6 then n7 else eod fi;
n6 = first n1 eq first c;
n7 = next c;
a = first n8;
n8 = if d then n1 else n9 fi;
n9 = next a;
```

Step 5: Transforming statements into daton statements by Rule 4.1:

$first\_n1[t] = ([[b, c]])[t]\ \forall t \in \mathcal{Z} \geq 0$

$n1[t] = (first\_n1$ `fby` $n2)[t]\ \forall t \in \mathcal{Z} \geq 0$

$n2[t] =$ (`if` $n3$ `then` $n4$ `else` $n5$ `fi`)$[t]\ \forall t \in \mathcal{Z} \geq 0$

$n3[t] =$ (`first` $n1$ `eq` $firstb)[t]\ \forall t \in \mathcal{Z} \geq 0$

$n4[t] =$ (`next` $b)[t]\ \forall t \in \mathcal{Z} \geq 0$

$n5[t] =$ (`if` $n6$ `then` $n7$ `else eod fi`)$[t]\ \forall t \in \mathcal{Z} \geq 0$

$n6[t] =$ (`first` $n1$ `eq first` $c)[t]\ \forall t \in \mathcal{Z} \geq 0$

$n7[t] =$ (`next` $c)[t]\ \forall t \in \mathcal{Z} \geq 0$

$a[t] =$ (`first` $n8)[t]\ \forall t \in \mathcal{Z} \geq 0$

$n8[t] =$ (`if` d `then` $n1$ `else` $n9$ `fi`)$[t]\ \forall t \in \mathcal{Z} \geq 0$

$n9[t] =$ (`next` $a)[t]\ \forall t \in \mathcal{Z} \geq 0$

Step 6: Transforming daton expressions recursively:

$$first\_n1[t] = ([[b, c]])[t] \; \forall t \in \mathcal{Z} \geq 0$$

$$\Rightarrow \; ((first\_n1[t] = b[t]) \; \texttt{or} \; (first\_n1[t] = c[t])) \; \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.8}$$

$$n1[t] = (first\_n1 \; \texttt{fby} \; n2)[t] \; \forall t \in \mathcal{Z} \geq 0$$

$$\Rightarrow \begin{cases} n1[0] = first\_n1[0] \\ n1[t+1] = n2[t] \; \forall t \in \mathcal{Z} \geq 0 \end{cases} \text{ by Rule 4.5}$$

$$n2[t] = (\texttt{if} \; n3 \; \texttt{then} \; n4 \; \texttt{else} \; n5 \; \texttt{fi})[t] \; \forall t \in \mathcal{Z} \geq 0$$

$$\Rightarrow \; n2[t] = (\texttt{if} \; n3[t] \; \texttt{then} \; n4[t] \; \texttt{else} \; n5[t] \; \texttt{fi})[t] \; \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.7}$$

$$n3[t] = (\texttt{first} \; n1 \; \texttt{eq} \; firstb)[t] \; \forall t \in \mathcal{Z} \geq 0$$

$$\Rightarrow \; n3[t] = (\texttt{first} \; n1)[t] \; \texttt{eq} \; (firstb)[t] \; \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.2}$$

$$\Rightarrow \; n3[t] = n1[0] \; \texttt{eq} \; b[0] \; \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.3}$$

$$n4[t] = (\texttt{next} \; b)[t] \; \forall t \in \mathcal{Z} \geq 0$$

$$\Rightarrow \; n4[t] = b[t+1] \; \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.4}$$

$$n5[t] = (\texttt{if} \; n6 \; \texttt{then} \; n7 \; \texttt{else} \; \textbf{eod} \; \texttt{fi})[t] \; \forall t \in \mathcal{Z} \geq 0$$

$$\Rightarrow \; n5[t] = (\texttt{if} \; n6[t] \; \texttt{then} \; n7[t] \; \texttt{else} \; \textbf{eod}[t] \; \texttt{fi})[t] \; \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.7}$$

$$\Rightarrow \; n5[t] = (\texttt{if} \; n6[t] \; \texttt{then} \; n7[t] \; \texttt{else eod fi}) \; \forall t \in \mathcal{Z} \geq 0$$

$$n6[t] = (\texttt{first} \; n1 \; \texttt{eq first} \; c)[t] \; \forall t \in \mathcal{Z} \geq 0$$

$$\Rightarrow \; n6[t] = (\texttt{first} \; n1)[t] \; \texttt{eq} \; (\texttt{first} \; c)[t] \; \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.2}$$

$$\Rightarrow \; n6[t] = n1[0] \; \texttt{eq} \; c[0] \; \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.3}$$

$$n7[t] = (\texttt{next} \; c)[t] \; \forall t \in \mathcal{Z} \geq 0$$

$$\Rightarrow \; n7[t] = c[t+1] \; \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.4}$$

$$a[t] = (\texttt{first} \; n8)[t] \; \forall t \in \mathcal{Z} \geq 0$$

$$\Rightarrow \; a[t] = n8[0] \; \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.3}$$

$$n8[t] = (\texttt{if} \; d \; \texttt{then} \; n1 \; \texttt{else} \; n9 \; \texttt{fi})[t] \; \forall t \in \mathcal{Z} \geq 0$$

$$\Rightarrow \; n8[t] = (\texttt{if} \; d[t] \; \texttt{then} \; n1[t] \; \texttt{else} \; n9[t] \; \texttt{fi}) \; \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.7}$$

$$n9[t] = (\texttt{next}\ a)[t]\ \forall t \in \mathcal{Z} \geq 0$$
$$\Rightarrow\ n9[t] = a[t+1]\ \forall t \in \mathcal{Z} \geq 0 \text{ by Rule 4.4}$$

4.6 Functions `constructCSP`, `findC`, and `transformC`

In this section, we introduce the algorithm of the Functions `constructCSP`, `findC`, and `transformC` with Example 4.2.

The function `constructCSP` constructs CSPs to ensure that the assertion daton variables at time t are satisified by assigning values to some selection daton variables. We give an example with the following parameters:

$$\begin{aligned} t &= 3 \\ S_{all} &= \bigcup_{t \in \mathcal{Z} \geq 0} \{a[0] = 2, a[t+1] = 2b[t],\ b[t] = [[-1, 1]], \\ &\qquad z[0] = \texttt{true},\ z[t+1] = (a[t+1] > b[t])\} \\ U_A &= \bigcup_{t \in \mathcal{Z} \geq 0} \{z[t]\} \end{aligned}$$

where

t is the construction stage of CSP

S_{all} is the set of all daton statements

U_A is the set of assertion daton variables

First we find the set of daton statements $S_{expand} \subset S_{all}$ with which we can assigning values to the $z[3]$. S_{expand} shall be

$\{a[3] = 2b[2],$

$b[2] = [[-1, 1]],$

$z[3] = (a[3] > b[2])\}$

We convert them to the set of constraints C_{expand}:

$\{a[3] = 2b[2],$

$(b[2] = -1)\ \texttt{or}\ (b[2] = 1),$

$z[3] = (a[3] > b[2])\}$

```
constructCSP (t, S_all, U_A, ζ)
begin
    /* assign defining daton statements of unassigned assertion daton variables
    to C*/
1   U := {u[t] | u[t] ∈ U_A};
2   S_expand := findC (U, ζ, S_all);
3   C_expand := transformC (S_expand);
4   C_A := {v[t] = true | v[t] ∈ V ∧ t ≤ t_0};
5   C_sol := C_expand ∪ C_A;
6   X_sol := var(C_sol);
7   D := a set containing, for every x ∈ X_sol, an associated domain D_x
    which is the daton domain of x;
8   return (⟨X_sol, D, C_sol⟩);
end
```

Figure 4.5: Pseudo-code of `constructCSP`

To ensure that the assertion daton variable is asserted, we add one more constraint $z[3] = \texttt{true}$. Finally we construct a CSP $\langle X_{sol}, D, C_{sol}\rangle$, where $X_{sol} = \{a[3], b[2], z[3]\}$, $D_{a[3]} = D_{b[2]} = \mathcal{Z}$, $D_{z[3]} = \{\texttt{true}, \texttt{false}\}$, C_{sol} is

$\{a[3] = 2b[2],$

$(b[2] = -1)$ `or` $(b[2] = 1),$

$z[3] = (a[3] > b[2]),$

$z[3] = \texttt{true}\}$

The pseudo-code of `constructCSP` is shown in Figure 4.5. The parameters of `constructCSP` are the construction stage t of CSP, the set of all daton statements S_{all}, the set of assertion daton variables U_A, and the valuation we have obtained ζ. At Line 1, the set of unassigned assertion daton variables with index t is assigned to the set U. With U, we can identify which daton statements in S_{all} can help us to solve the problem by invoking `findC` at Line 2, where S_{expand} is the set of helpful daton statements. Daton statements in S_{expand} are transformed to a set of constraints C_{expand} at Line 3. C_A at Line 4 is a set of constraints to ensure that assertion daton variables are equal

to `true`. The set of constraints we need is C_{sol} at Line 5 which is the union of C_{expand} and C_A. The set of variables X_{sol} at Line 6 is equal to $\texttt{var}(C_{sol})$. The set of domain D at Line 7 is a set containing, for every $x \in X_{sol}$, an associated domain D_x which is the daton domain of x. The CSP $\langle X_{sol}, D, C_{sol}\rangle$ is returned at Line 8.

Given set of daton variables U, `findC` is responsible for finding a set of daton statements S with which can determine the possible valuations over $\texttt{var}(S) \supset U$. We give an example with the following parameters:

$$\begin{aligned} U &= \{z[3]\} \\ \zeta &= \{a[0] \mapsto 1,\ z[0] \mapsto \texttt{true}\} \\ S_{all} &= \bigcup_{t \in \mathcal{Z}_{\geq 0}} \{a[0] = 2, a[t+1] = 2b[t],\ b[t] = [[-1, 1]], \\ &\qquad z[0] = \texttt{true},\ z[t+1] = (a[t+1] > b[t])\} \end{aligned}$$

where

U is the set of daton variables to be assigned

ζ is the valuation we have obtained before the function is invoked

S_{all} is the set of all daton statements

To determine the possible valuations over U, we need to find $S = \zeta(\texttt{sta}(U)) = \{\zeta(\texttt{sta}(z[3]))\}$. If $\texttt{sta}(z[3])$ were "$z[3] = z[0]$", we would know that $S = \{z[3] = \texttt{true}\}$ from ζ and $\{z[3] \mapsto \texttt{true}\}$ is the possible valuation, so that S would be returned by the function. However, as $\texttt{sta}(z[3])$ is "$z[3] = a[3] > b[2]$", where $a[3]$ and $b[2]$ are unassigned any value, S is not the set to be returned. This can be checked by finding whether $\texttt{var}(S) \subseteq U$, or simply $\texttt{var}(S) = U$ as $\texttt{var}(S)$ must be a subset of U. If it is true, S is returned; Otherwise another set of daton statements S' should be found, with which the valuation over the set of unassigned daton varaibles can be determined. This is done by invoking `findC` recursively.

At the second recursive call of `findC`, we add a slash "$'$" to each variable to denote that it is a variable of the second recursive call. Two slashes are added

at the third call and so on. The notation is used in the subsequent sections.

At the second recursive call, we add $a[3]$ and $b[2]$ to U and get $U' = \{z[3],\ a[3],\ b[2]\}$, while $\zeta' = \zeta$ and $S'_{all} = S_{all}$. S' is $\zeta(\{\texttt{sta}(z[3]),\ \texttt{sta}(a[3]),\ \texttt{sta}(b[2])\})$, it is:

$$\{z[3] = a[3] > b[2],\quad a[3] = 2b[2] + 1,\quad b[2] = [[-1, 1]]\}$$

As $\texttt{var}(S') = \{z[3],\ a[3],\ b[2]\} = U$, S' is the result. We can determine that the possible valuations are $\{b[2] \mapsto -1,\ a[3] \mapsto -1,\ z[3] \mapsto \texttt{false}\}$ and $\{b[2] \mapsto 1,\ a[3] \mapsto 3,\ z[3] \mapsto \texttt{true}\}$.

S is *expanded* to S' at the recursive call. We called that S' is the *expansion* of S. Our algorithm is based on expanding $S = \texttt{sta}(U)$ by recursive calls until we obtain an extension $S^{n\prime}$, where $\texttt{var}(S^{n\prime}) = U^{n\prime}$ at the $(n+1)^{th}$ recursive call. If there are some variables which depend on their "future values", there will be an infinite recursion. For example, if the program has the statement `a = next a`, $\bigcup_{t \in \mathcal{Z}_{\geq 0}} \{a[t] = a[t+1]\}$ will be a subset of S_{all}. When U consists of $a[0]$, S will be expanded infinitely to $S^{\infty\prime} \supset \bigcup_{t \in \mathcal{Z}_{\geq 0}} \{\texttt{sta}(a[t])\}$. The interpreter will check and report the error.

The pseudo-code of `findC` is shown in Figure 4.6. The parameters of `findC` are the set of daton variables U, the valuation ζ, and the set of all daton statements S_{all}. The set of defining statements of unassigned assertion daton variables with index t is assigned to the set S at Line 1. At Line 2, the function checks whether $\texttt{var}(S)$ is equal to U. If it is true, S is the required set of daton statements and returned at Line 3. Otherwise, `findC` is invoked recursively to expand S at Line 4. The recursion terminates when $\texttt{var}(S) = U$ at Line 2.

Function `transformC` transforms a set of daton statements to a set of constraints. Most operators of daton statements can be directly mapped to operators of constraints with a few exceptions: discrete choice operators, range

```
findC (U, ζ, S_all)
begin
1    S := {ζ(sta(u))|∀u ∈ U ∧ sta(u) ∈ S_all ∧ u ∉ var(ζ)};
     /* if S consists of all defining daton statements of var(S)*/
2    if var(S) == U then
3        return (S);
     else
         /* expand S otherwise */
4        return (findC (var(S), ζ, S_all));
     end
end
```

Figure 4.6: Pseudo-code of `findC`

choice operators, `if-then-else-fi`, and "=". The statements with the exceptions are transformed to constraints by logical operators. The pseudo-code of `transformC` is shown in Figure 4.7. The parameter of `transformC` is the set of daton statements S.

We use Example 4.2 to demonstrate the collaboration of `constructCSP`, `findC`, and `transformC`.

Example 4.2 Function `constructCSP` is invoked with the following parameters:

$$
\begin{aligned}
t &= 1 \\
S_{all} &= \bigcup_{t \in \mathcal{Z} \geq 0} \{a[t+1] = c[t+1] > d[t],\ b[t+1] = d[t+1]\ \texttt{eq}\ e[t], \\
&\qquad c[0] = [[5,6]],\ c[t+1] = c[t] + 1, \\
&\qquad d[t] = \texttt{if}\ e[t] > 0\ \texttt{then}\ e[t]\ \texttt{else}\ -e[t]\ \texttt{fi}, \\
&\qquad e[t] = 3,\ f[t] = d[t] + c[t]\} \\
U_A &= \bigcup_{t \in \mathcal{Z} \geq 0} \{a[t],\ b[t]\} \\
\zeta &= \{a[0] \mapsto \texttt{true},\ b[0] \mapsto \texttt{true},\ c[0] \mapsto 6,\ d[0] \mapsto 3,\ e[0] \mapsto 3,\ f[0] \mapsto 9\}
\end{aligned}
$$

```
transformC (S)
begin
   /* if S consists of all defining daton statements of var(S)*/
   foreach s ∈ S do
      if s is u = [[r1, ..., rn]] then
         replace s by u = r1 or ... or u = rn;

      else if s is u = [[r1 : r2]] then
         replace s by u ≥ r1 and u ≤ r2;

      else if s is u = {[r1 : r2]] then
         replace s by u > r1 and u ≤ r2;

      else if s is u = [[r1 : r2}} then
         replace s by u ≥ r1 and u < r2;

      else if s is u = {{r1 : r2}} then
         replace s by u > r1 and u < r2;

      else if s is u = if e1 then e2 else e3 fi then
         replace s by (not e1 or u eq e2) and (e1 or u eq e3);

      Change all assignment operator "=" of s to eq;
   end
   return (S);
end
```

Figure 4.7: Pseudo-code of `transformC`

$a[1] = c[1] > 3,$	$b[1] = d[t]$ `eq` 3
$c[1] = 6 + 1,$	$d[1] =$ `if` $e[1] > 0$ `then` $e[1]$ `else` $-e[1]$ `fi`
$e[1] = 3$	

Table 4.1: Expansion of S

U at Line 1 of Figure 4.5 consists of two unassigned assertion daton variables, $a[1]$ and $b[1]$, with indices equal to t. U is passed to `findC` at Line 2.

At Line 1 of Figure 4.6, as $\zeta(\texttt{sta}(a[1]))$ is "$a[1] = c[1] > 3$" and $\zeta(\texttt{sta}(b[1]))$ is "$b[1] = d[t]$ `eq` 3", S is $\{a[1] = c[1] > 3,\ b[1] = d[t]$ `eq` $3\}$ and $\texttt{var}(S) = \{a[1], b[1], c[1], d[1]\}$. At Line 4, S is passed to `findC` recursively.

At the second recursive call of `findC`, S is expanded to $S' = \{a[1] = c[1] > 3,\ b[1] = d[t]$ `eq` $3,\ c[1] = 6 + 1,\ d[1] =$ `if` $e[1] > 0$ `then` $e[1]$ `else` $-e[1]$ `fi`$\}$. Similarly, as $\texttt{var}(S') = \{a[1], b[1], c[1], d[1], e[1]\}$, S' is further expanded to $S'' = \{a[1] = c[1] > 3,\ b[1] = d[t]$ `eq` $3,\ c[1] = 6 + 1,\ d[1] =$ `if` $e[1] > 0$ `then` $e[1]$ `else` $-e[1]$ `fi`$,\ e[1] = 3\}$ at the third recursive call. The expansion is accomplished as $\texttt{var}(S'') = U''$. The set to be returned, S'', is S_{expand}.

The expansion can be simply represented by Table 4.1. The daton statements at the 1^{st} column represent S, while S' is represented by the daton statements at the 1^{st} and 2^{nd} columns collectively, and S'' is represented by the daton statements at the 1^{st}, 2^{nd}, and 3^{rd} columns collectively. In general, columns 1 to n represent the set of daton statements at the n expansions. The whole table represents the set of daton statements S_{expand}.

At Line 3 of Figure 4.5, S_{expand} is passed to `transformC`. We obtain the set of constraints $C_{expand} = \{a[1]$ `eq` $c[1] > 3,\ b[1]$ `eq` $d[t]$ `eq` $3,\ c[1]$ `eq` $6 + 1,$ `not` $e[1] > 0$ `or` $d[1]$ `eq` $e[1],\ e[1] > 0$ `or` $d[1]$ `eq` $-e[1],\ e[1]$ `eq` $3\}$.

At Line 4, we find that $C_A = \{a[1]$ `eq` `true`, $b[1]$ `eq` `true`$\}$. With C_{expand}, we find that $C_{sol} = C_{expand} \cup C_A$ at Line 5. At Line 6, we find that $X_{sol} = \{a[1], b[1], c[1], d[1], e[1]\}$. At Line 7, we find that $D = \{D_{a[1]}, D_{b[1]}, D_{c[1]}, D_{d[1]}, D_{e[1]}\}$, where $D_{a[1]} = D_{b[1]} = \{$`true`, `false`$\}$ and $D_{c[1]} = D_{d[1]} = D_{e[1]} = \mathcal{Z}$.

The CSP obtained at Line 8 is $\langle X_{sol}, D, C_{sol} \rangle$. □

4.7 An Example

To give a whole picture before discussing `backtrack` in the next section, we give an example run of `ELUCIDinterpreter` to show the collaboration of `ELUCIDinterpreter`, `Edemand`, `transformD`, `constructCSP`, `findC`, and `transformC` by Example 4.3.

Example 4.3 Function `ELUCIDinterpreter` is invoked with the set of variables of interest I and the set of statements E:

$I = \{$`x, y`$\}$

$E = \{$`x = 1 fby [1, 2] + x;`
`y = 3 fby [[1,2]] + y;`
`z = y - x eq 2;`$\}$

At Line 1 of pseudo-code of `ELUCIDinterpreter` at Figure 4.2, E is transformed to a set of daton statements S by `transformD`. S is:

$\bigcup_{t \in \mathcal{Z} \geq 0}$ $\{x[t+1] = a[t] + x[t],\ y[t+1] = b[t] + y[t],\ a[0] = [[1,\ 2]],$
$a[t+1] =$ `if` $a[0]$ `eq 1 then 1 else` $n[t]$ `fi`,
$n[t] =$ `if` $a[0]$ `eq 2 then 2 else eod fi`,
$b[t] = [[1,\ 2]],\ z[t] = (y[t] - x[t]$ `eq` $2)\}$

where $a[t]$ and $b[t]$ for $t \in \mathcal{Z} \geq 0$ are newly created selection daton variables.

At stage 0, $x[0]$, $y[0]$, and $z[0]$ are demanded at Line 2. Without constructing any CSP, we obtain the solution up to stage 0: $\zeta = \{x[0] \mapsto 1,\ y[0] \mapsto 3,$ $z[0] \mapsto$ `true`$\}$.

At stage 1, $x[1]$ is demanded first. It drives $a[0]$ and its selection operator, so that `constructCSP` is invoked at Line 4 of Figure 4.3. S_{expand} at Line 2 of Figure 4.5 is obtained by expansion shown in Table 4.2. At Line 3, C_{expand} is :

$z[1] = (y[1] - x[1]$ eq $2)$	
$x[1] = a[0] + 1$	$y[1] = b[0] + 3$
$a[0] = [[1,\ 2]]$	$b[0] = [[1,2]]$

Table 4.2: Expansion of S at stage 1

$\{z[1]$ eq $(y[1] - x[1]$ eq $2)$,
$x[1]$ eq $a[0] + 1$, $y[1]$ eq $b[0] + 3$,
$(a[0]$ eq $1)$ or $(a[0]$ eq $2)$, $(b[0]$ eq $1)$ or $(b[0]$ eq $2)\}$

C_{sol} at Line 5 is the union of C_{expand} and $C_A = \{z[1]$ eq true$\}$. D at Line 7 is simply obtained from the daton domains of daton variables. The CSP constructed is $\langle \mathtt{var}(C_{sol}), D, C_{sol} \rangle$, which is returned by constructCSP at Line 8. It is solved and the solutions are projected on the selection daton variables, $a[0]$ and $b[0]$ at Line 5 of Figure 4.3. The possible solutions of the CSP are $Sol[1][0] = \{a[0] \mapsto 1, b[0] \mapsto 1\}$ and $Sol[1][1] = \{a[0] \mapsto 2, b[0] \mapsto 2\}$.

As $solIndex[1]$ is 0, $Sol[1][0]$ is chosen. We obtain $\zeta = \{x[0] \mapsto 1, y[0] \mapsto 3, z[0] \mapsto$ true, $a[0] \mapsto 1, b[0] \mapsto 1\}$. As $a[0]$ and $x[0]$ are assigned values already, $x[1]$ is assigned $a[0] + x[0] = 1 + 1 = 2$. Then $y[1]$ and $b[1]$ are demanded. As $b[0]$ and $y[0]$ are assigned values already, $y[1]$ is assigned $b[0] + y[0] = 1 + 3 = 4$. Similarly, $z[1]$ is demanded and true is obtained. We obtain the solution up to stage 1: $\zeta = \{x[0] \mapsto 1, y[0] \mapsto 3, z[0] \mapsto$ true, $a[0] \mapsto 1, b[0] \mapsto 1, x[1] \mapsto 2, y[1] \mapsto 4, z[1] \mapsto$ true$\}$.

At stage 2, $x[2]$, $y[2]$, and $z[2]$ are demanded. As $a[1]$ and $x[1]$ are assigned values already, $x[2]$ is assigned $a[1] + x[1] = 1 + 2 = 3$. On the other hand, $b[1]$ and its selection operator are demanded and constructCSP is invoked. C_{expand} is obtained by expansion shown in Table 4.3. C_{expand} is :

$\{z[2]$ eq $(y[2] - 1$ eq $2)$,
$y[2]$ eq $b[1] + 4$,
$(b[1]$ eq $1)$ or $(b[1]$ eq $2)\}$

$z[2] = (y[2] - 1$ `eq` $2)$
$y[2] = b[1] + 4$
$b[1] = [[1, 2]]$

Table 4.3: Expansion of S at stage 2

C_{sol} is the union of C_{expand} and $C_A = \{z[2]$ `eq true`$\}$. The set of domain D is simply obtained from the daton domains of daton variables. The CSP constructed is $\langle \mathtt{var}(C_{sol}), D, C_{sol} \rangle$. It is solved and the solutions are projected on the selection daton variables, $a[1]$ and $b[1]$. The solution of the CSP is $Sol[2][0] = \{b[1] \mapsto 1\}$.

As $solIndex[2]$ is 0, $Sol[2][0]$ is chosen and added to ζ. As $b[1]$ and $y[1]$ are assigned values already, $y[2]$ is assigned $b[1] + y[1] = 1 + 4 = 5$. Similarly, $z[2]$ is demanded and `true` is obtained. Finally $\{a[1] \mapsto 1,\ b[1] \mapsto 1,\ x[2] \mapsto 3,\ y[2] \mapsto 5,\ z[2] \mapsto$ `true`$\}$ is added to ζ, which is the solution up to stage 2.

The process continues, and more assignments are added to ζ with increasing stage. □

4.8 Function `backtrack`

The function `backtrack` tries to find alternative solutions upon user request or when a CSP without solution is constructed. Considering this E-Lucid program:

```
x = [[1, -1]];
y = x fby 2y;
assert z = (y < 3);
```

The statements are transformed to the following daton statements:

$\bigcup_{t \in \mathcal{Z}_{\geq 0}} \{\ x[t] = 1$ `or` $x[t] = -1,\ y[0] = x[0],$
$y[t+1] = 2y[t],\ z[t] = (y[t] < 3)\}$

The corresponding selection operator of x fires "1"s at the 1^{st} and the 2^{nd}

firings, so that we have the solution up to stage 1 as $\{x[0] \mapsto 1,\ y[0] \mapsto 1,\ z[0] \mapsto \texttt{true},\ x[1] \mapsto 1,\ y[1] \mapsto 2,\ z[1] \mapsto \texttt{true}\}$. However, we are unable to assign value to $y[2]$ as $2y[1] = 4 \not< 3$.

But it is obvious that there is a solution if $x[0]$ is assigned -1. We can find the solution by the technique of *backtracking* as following. We undo the valuation at the previous stage and let the selection operator(s) fire(s) alternative datons. If we are still unable to find a solution, we undo more valuations until we find a solution. If it cannot be found after undoing all valuations, the program has no solution and the function prints "No further backtracking".

In the example, we undo the valuation done at stage 1 and the selection operator fire "-1" at the 2^{nd} firing, so that we have the solution up to stage 1 as $\{x[0] \mapsto 1,\ y[0] \mapsto 1,\ z[0] \mapsto \texttt{true},\ x[1] \mapsto -1,\ y[1] \mapsto 2,\ z[1] \mapsto \texttt{true}\}$. However, we are still unable to assign value to $y[2]$ as $2y[1] = 4 \not< 3$.

We undo the valuation done at stage 1, but the selection operator has no alternative daton to fire at the 2^{nd} firing. Therefore we undo the valuation done at stage 0 also and the selection operator fire "-1" at the 1^{st} firing, so that we have the solution up to stage 0 as $\{x[0] \mapsto -1,\ y[0] \mapsto -1,\ z[0] \mapsto \texttt{true}\}$, and we will get $y[1] \mapsto -2$ and $y[1] \mapsto -4$, with which both $z[1]$ and $z[2]$ will be assigned `true`. The process continues and no more backtracking is required.

Backtracking is invoked when there is no solution with the valuation we have. Moreover, a user can request backtracking at each stage. Continuing the example, the selection operator fire "1" at the 2^{nd} firing, so that we have the solution up to stage 1 as $\{x[0] \mapsto -1,\ y[0] \mapsto -1,\ z[0] \mapsto \texttt{true},\ x[1] \mapsto 1,\ y[1] \mapsto -2,\ z[1] \mapsto \texttt{true}\}$. A user can request backtracking at stage 1, so that the valuation done at stage 1 is undone. The selection operator fire "-1" at the 2^{nd} firing, so that we have the solution up to stage 1 as $\{x[0] \mapsto -1,\ y[0] \mapsto -1,\ z[0] \mapsto \texttt{true},\ x[1] \mapsto -1,\ y[1] \mapsto -2,\ z[1] \mapsto \texttt{true}\}$.

The pseudo-code of `backtracking` is shown in Figure 4.8. The parameters of the function are the invoking time *startt*, the array of solution counters

```
backtrack (startt, solIndex, Sol, ζ)
begin
    exit := false;
    t := startt;
    repeat
1       if t < 0 then
2           print "No further backtracking";
            exit := true;
3       else if solIndex[t] < |Sol[t]| − 1 then
            /*pick the next solution. As solIndex[t] can be 0 to indicate the
            1st solution is picked, Sol[t] is decreased by 1*/
4           solIndex[t] := solIndex[t] + 1;
            exit := true;
        else
            /* no more solution */
5           remove the valuations from ζ at stage t − 1;
6           Sol[t][solIndex[t]] := {};
7           t := t − 1;
        end
    until NOT exit;
    return (t);
end
```

Figure 4.8: Pseudo-code of `backtrack`

solIndex, the array of sequence of solutions *Sol*, and the valuation has been obtained ζ. If we find that $solIndex[t] < |Sol[t]| - 1$ at Line 3, the counter $solIndex[t]$ is increased by 1 to pick up the next solution at Line 4. Otherwise assignments of selection daton variables at stage t are removed from ζ at Line 5 and 6, and the stage number is decreased by 1 at Line 7. At Line 1, if t is decreased to -1, CSPs of all stages have no solution and "No further backtracking" is printed at Line 2. Example 4.4 shows the algorithm.

Example 4.4 We are given the set of variables of interest I and the set of statements E, with corresponding Extended Dataflow Network shown in Figure 4.9.

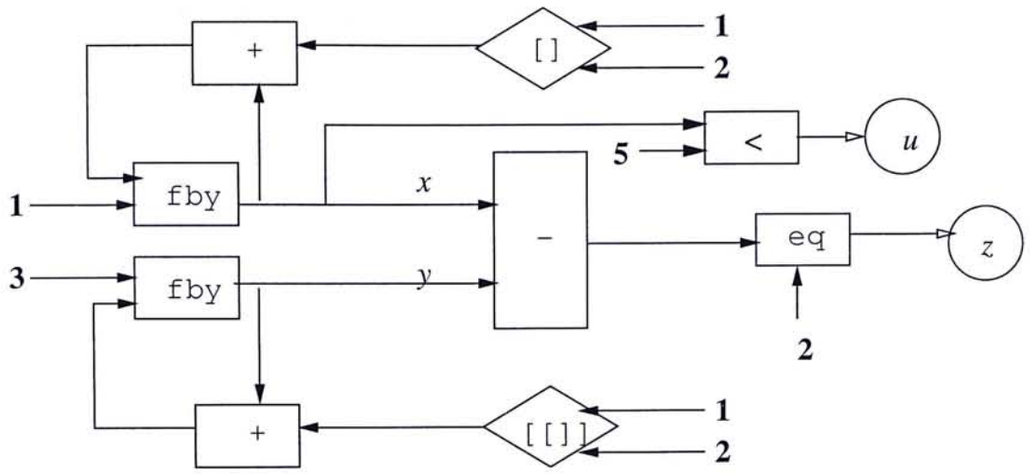


Figure 4.9: Backtracking example

$I =$ {x, y}

$E =$ {x = 1 fby (x + [1, 2]); y = 3 fby (y + [[1, 2]]);
assert z = (y - x eq 2); assert u = (x < 5)}

E is transformed to a set of daton statements S by `transformD`. S is:

$\bigcup_{t \in \mathcal{Z} \geq 0}$ $\{x[0] = 1,\ x[t+1] = x[t] + a[t],\ a[0] = [[1,\ 2]],$
$a[t+1] =$ `if` $a[0]$ `eq 1 then 1 else` $n[t]$ `fi`,
$n[t] =$ `if` $a[0]$ `eq 2 then 2 else eod fi`,
$y[0] = 3,\ y[t+1] = y[t] + b[t],\ b[t] = [[1,\ 2]],$
$z[t] = (y[t] - x[t]$ `eq` $2),\ u[t] = (x[t] < 5)\}$

where $a[t]$, $b[t]$ for $t \in \mathcal{Z} \geq 0$ are newly created selection daton variables.

At stage 0, $x[0]$, $y[0]$, $z[0]$, and $u[0]$ are demanded. We obtain the solution up to stage 0: $\zeta = \{x[0] \mapsto 1,\ y[0] \mapsto 3,\ z[0] \mapsto$ `true`, $u[0] \mapsto$ `true`$\}$.

At stage 1, $x[1]$ is demanded. Thus $a[0]$ and its selection operator are demanded and `constructCSP` is invoked. The CSP constructed is $\langle$`var`(C_{sol}), D, $C_{sol}\rangle$, where C_{sol} is:

$\{z[1]$ `eq true`, $u[1]$ `eq true`,
$z[1]$ `eq` $(y[1] - x[1]$ `eq` $2)$, $u[1]$ `eq` $(x[1] < 5)$, $x[1]$ `eq` $1 + a[0]$,
$y[1]$ `eq` $3 + b[0]$, $(a[0]$ `eq` $1)$ `or` $(a[0]$ `eq` $2)$, $(b[0]$ `eq` $1)$ `or` $(b[0]$ `eq` $2)$

There are two solutions: $Sol[1][0] = \{a[0] \mapsto 1,\ b[0] \mapsto 1\}$ and $Sol[1][1] = \{a[0] \mapsto 2,\ b[0] \mapsto 2\}$. We choose $Sol[1][0]$. Thus valuation gets additional elements: $x[1] \mapsto 2$, $y[1] \mapsto 4$, $s[0] \mapsto 1$, $d[0] \mapsto 1$, $z[1] \mapsto$ `true`, and $u[1] \mapsto$ `true`.

At stage 2, $a[1]$ is assigned 1 and $x[2]$ is assigned 3. The selection operator of $b[1]$ is demanded. The CSP constructed is $\langle \mathtt{var}(C_{sol}),\ D,\ C_{sol} \rangle$, where C_{sol} is:

$\{z[2]$ `eq` $(y[2] - 3$ `eq` $2),\ u[2]$ `eq` $(3 < 5),$
$y[2]$ `eq` $3 + b[1],\ (b[1]$ `eq` $1)$ `or` $(b[1]$ `eq` $2),$
$z[2]$ `eq` `true`, $u[2]$ `eq` `true`$\}$

There is one solution $Sol[2][0] = \{b[1] \mapsto 1\}$. Valuation gets additional elements: $x[2] \mapsto 3$, $y[2] \mapsto 5$, $a[1] \mapsto 1$, $b[1] \mapsto 1$, $z[2] \mapsto$ `true`, and $u[2] \mapsto$ `true`.

At stage 3, $a[2]$ is assigned 1 and $x[3]$ is assigned 4. The selection operator of $b[2]$ is demanded. The CSP constructed is $\langle \mathtt{var}(C_{sol}),\ D,\ C_{sol} \rangle$, where C_{sol} is:

$\{z[3]$ `eq` $(y[3] - 4$ `eq` $2),\ u[3]$ `eq` $(4 < 5),$
$y[3]$ `eq` $5 + b[2],\ (b[2]$ `eq` $1)$ `or` $(b[2]$ `eq` $2),$
$z[3]$ `eq` `true`, $u[3]$ `eq` `true`$\}$

There is one solution $Sol[3][0] = \{b[2] \mapsto 1\}$. Valuation gets additional elements: $x[3] \mapsto 4$, $y[3] \mapsto 6$, $a[2] \mapsto 1$, $b[2] \mapsto 1$, $z[3] \mapsto$ `true`, and $u[3] \mapsto$ `true`.

At stage 4, $a[3]$ is assigned 1 and $x[4]$ is assigned 5. The selection operator of $b[3]$ is demanded. The CSP constructed is $\langle \mathtt{var}(C_{sol}),\ D,\ C_{sol} \rangle$, where C_{sol} is:

$\{z[4]$ `eq` $(y[4] - 5$ `eq` $2),\ u[4]$ `eq` $(5 < 5),$
$y[4]$ `eq` $6 + b[2],\ (b[3]$ `eq` $1)$ `or` $(b[3]$ `eq` $2),$
$z[4]$ `eq` `true`, $u[4]$ `eq` `true`$\}$

There is no solution. The function `backtrack` is invoked with *startt* as 4. At Line 3 of Figure 4.8, as $solIndex[4] = |Sol[4]| = 0$, t is decreased from 4 to 3 at Line 7. As $solIndex[3] = |Sol[3]| = 0$, t is decreased to 2. As $solIndex[2] = |Sol[2]| = 1$, t is decreased to 1. As $solIndex[1] = 1$ and $|Sol[1]| = 2$, $solIndex[1]$ is increased to 1 at Line 4. Then at stage 1, we pick $Sol[1][1] = \{a[0] \mapsto 2, b[0] \mapsto 2\}$. We obtain the solution up to stage 1: $\zeta = \{x[0] \mapsto 1, y[0] \mapsto 3, z[0] \mapsto \texttt{true}, u[0] \mapsto \texttt{true}, x[1] \mapsto 3, y[1] \mapsto 5, a[0] \mapsto 2, b[0] \mapsto 2, z[1] \mapsto \texttt{true}, u[1] \mapsto \texttt{true}\}$.

At stage 2, $a[1]$ is assigned 2 and $x[2]$ is assigned 4. The selection operator of $b[1]$ is demanded. The CSP constructed is $\langle \texttt{var}(C_{sol}), D, C_{sol} \rangle$, where C_{sol} is:

$\{z[2]$ `eq` $(y[2] - 4$ `eq` $2),$ $u[2]$ `eq` $(4 < 5),$
$y[2]$ `eq` $3 + b[1],$ $(b[1]$ `eq` $1)$ `or` $(b[1]$ `eq` $2),$
$z[2]$ `eq` `true`, $u[2]$ `eq` `true`$\}$

There is no solution. The function `backtrack` is invoked with *startt* as 2. As $|Sol[2]| = 0$, t is decreased from 2 to 1. As $solIndex[1] = |Sol[1]| = 2$, t is decreased to 0. As $|Sol[0]| = 0$, t is decreased to -1. At Line 2, the message "No further backtracking" is printed and the program ends. □

Chapter 5

Related Works

We can model dynamic problems by various kinds of systems. In this chapter, we survey some of them and compare them with Extended Dataflow Network and E-Lucid.

Difference equations and dynamic systems [30] define a variable, which is a sequence of values, by exactly one equation, so that it is impossible to define extra constraints for the variable. For example, if we want to define x to be y and **3**, E-Lucid can specify them with two statements `x = y` and `assert b = (x eq 3)`. However, a difference equation can only specify $x(t) = y(t)$ or $x(t) = 3$, but not both. Therefore they cannot model problems specified with extra constraints, such as Traffic Light Problem. In addition to equations defining the colors of the light, its E-Lucid program needs to use assertion variables to define extra safety constraints.

Dynamic programming [5] can specify extra constraints. However, a dynamic programming model can have only one control variable for decisions. On the other hand, E-Lucid programs support arbitrary number of selection operators for non-deterministic decisions. For example, the E-Lucid program of Traffic Lights Problem uses 8 selection operators to define the possible signals of the lights.

E-Lucid is extended from Lucid. As a dataflow language, Lucid is intuitive to express dynamic problems. With the operator `first` and `attime`, Lucid

can specify equations with *fixed time points*, which means that the indices of datastreams can be specified as constants in an equation. For example, a Lucid statement `x = y attime 8` represents that $x[t] = y[8]$ for all $t \in \mathcal{Z} \geq 0$. E-Lucid shares the same feature, which is important in modelling the Modified Four Cockroaches Problem to specify the indices of coordinates we would like to find. This is not true for the dynamic systems mentioned before, which can only express the indices in terms of t, the current time. For example, we may find $x(t)$, $x(t+1)$, or $x(t-2)$ but not $x(1)$, $x(8)$, or $x(5)$ in a difference equation. On the other hand, however, Lucid is not able to specify any extra constraints.

Theoretically, we can model a dynamic problem with a CSP. It is done by expressing a variable v in E-Lucid program by a set of variables $\{v[0], v[1], ...\}$, and a constraint c in E-Lucid program by a set of constraints $\{c[0], c[1], ...\}$. However, the sets are infinite.

Dynamic CSP [10] extends classical CSP so that a Dynamic CSP can be an infinite sequence of classical CSPs. By splitting a dynamic problem into infinite stages with finite set of variables and constraints, the problem can be modelled with Dynamic CSP. However, it is often tedious to express dynamic problems with constraints. For example, the infinite set of constraints $\bigcup_{t \in \mathcal{Z} \geq 0} \{x[t] = 1\}$ can be easily expressed by just one E-Lucid statement `x = 1`.

Using Constraint Logic Programming (CLP) [19], the dynamic problem can be modelled by expressing a variable v in E-Lucid program as a list of variables, and a constraint c in E-Lucid program as a recursive rule. For example, the Modified Four Cockroaches Problem can be modelled with CLP($\mathcal{R}$) as following:

```
cockroach([], [], [], [], [], [], [], [], _, 0).
cockroach([_], [_], [_], [_], [_], [_], [_], [_], _, 1).
cockroach([X1A,X1B|X1T], [Y1A,Y1B|Y1T], [X2A,X2B|X2T],
          [Y2A,Y2B|Y2T], [X3A,X3B|X3T], [Y3A,Y3B|Y3T],
```

```
            [X4A,X4B|X4T], [Y4A,Y4B|Y4T], D, L) :-
    cockroach([X1B|X1T], [Y1B|Y1T], [X2B|X2T],
            [Y2B|Y2T], [X3B|X3T], [Y3B|Y3T],
            [X4B|X4T], [Y4B|Y4T],  D, L-1),
        X1B = X1A + D *(X2A - X1A) /
            pow(pow(X2A - X1A, 2.0) + pow(Y2A - Y1A, 2.0), 0.5),
        Y1B = Y1A + D *(Y2A - Y1A) /
            pow(pow(X2A - X1A, 2.0) + pow(Y2A - Y1A, 2.0), 0.5),
        X2B = X2A + D *(X3A - X2A) /
            pow(pow(X3A - X2A, 2.0) + pow(Y3A - Y2A, 2.0), 0.5),
        Y2B = Y2A + D *(Y3A - Y2A) /
            pow(pow(X3A - X2A, 2.0) + pow(Y3A - Y2A, 2.0), 0.5),
        X3B = X3A + D *(X4A - X3A) /
            pow(pow(X4A - X3A, 2.0) + pow(Y4A - Y3A, 2.0), 0.5),
        Y3B = Y3A + D *(Y4A - Y3A) /
            pow(pow(X4A - X3A, 2.0) + pow(Y4A - Y3A, 2.0), 0.5),
        X4B = X4A + D *(X1A - X4A) /
            pow(pow(X1A - X4A, 2.0) + pow(Y1A - Y4A, 2.0), 0.5),
        Y4B = Y4A + D *(Y1A - Y4A) /
            pow(pow(X1A - X4A, 2.0) + pow(Y1A - Y4A, 2.0), 0.5).
```

The first eight arguments of `cockroach` are lists, which represent the xy-coordinates of the four cockroaches. The ninth argument `D` is the distance a cockroach can walk within one time unit. The tenth argument `L` is the time limit for the cockroaches to walk. The query is:

```
?- cockroach([X1S,0.01|X1], [Y1S, 0|Y1], [X2S,10|X2],
             [Y2S,0.01|Y2], [X3S,9.99|X3], [Y3S,10|Y3],
             [X4S,0|X4], [Y4S,9.99|Y4], 0.01, 7).
```

`X1S` is the x-coordinate of the first cockroach at time 0. `X1` is a list of the

x-coordinates of the first cockroaches from time unit 2 to L = 7. Similarly, other variables represent the xy-coordinates of the cockroaches.

E-Lucid represents a sequence by a datastream with infinite datons, while CLP represents it by a list with finite elements as CLP programs should not be intended for perpetual execution for semantic reason [29]. Therefore we need to specify the length of each list in a CLP model. In the example, the time limit L is equivalent to the lengths of the lists. However, if there are non-linear constraints, CLP($\mathcal{R}$) may not be able to solve it[1]. For example, in solving the Modified Four Cockroaches Problem, CLP($\mathcal{R}$) prints "Maybe" to indicate that it cannot determine the satisfiability [20] and the solution shown is just a rephrasing of the constraints.

[1]By using lazy mechanism, CLP($\mathcal{R}$) is able to solve some problems with non-linear constraints, but some are still not solvable.

Chapter 6

Conclusion

We conclude the thesis in this chapter by giving our contributions and possible directions of future works.

First, we have proposed an extension of Dataflow Network, Extended Dataflow Network, by introducing selection operators and assertion arcs. We can define the possible values with selection operators, that only fire datons with which the assertion arcs are satisfied. With the mechanism, Extended Dataflow Network can model dynamic problems with constraints. Second, we have proposed its corresponding language E-Lucid with a formal definition. Third, we have implemented a prototype E-Lucid interpreter by extending the pLucid interpreter [1, 12] with help of a CSP solver to solve E-Lucid programs. The pLucid interpreter can solve the dataflow network part while the CSP solver can implement the idea of selection variables and assertion variables. With backtracking, we are able to find all finite desirable solutions. Fourth, we have expressed several problems in E-Lucid to show the expressive power of the language. We can perform "backward valuation", express complicated interrelationship among variables, and find multiple solutions by E-Lucid programs.

Our work shows a possible way to make use of constraint solving techniques in dataflow network. There is plenty of scope for future work. First, we construct and compile ILOG programs on-the-fly in our implementation. The compilation accounts for large amount of execution time, which slows down

the whole process. It would be worth to construct a parser so that only a single compilation is required.

Second, our implementation does not support user-defined functions. Although the feature does not enhance the expressiveness, it is worth to develop it to facilitate programming. For example, we can define the function `light(Vi, Pi, Vj, Pj)` for Traffic Light Problem as follows.

```
light(Vi, Pi, Vj, Pj) =
      (Vi eq 1 and Pi eq 1 and Vj eq 3 and Pj eq 3)
   or (Vi eq 2 and Pi eq 1 and Vj eq 4 and Pj eq 1)
   or (Vi eq 3 and Pi eq 3 and Vj eq 1 and Pj eq 1)
   or (Vi eq 4 and Pi eq 1 and Vj eq 2 and Pj eq 1);
```

With the function, we can redefine *L12*, *L23*, *L34*, and *L41* in Section 3.4.2 as follows, which are simpler and clearer than the original ones.

```
assert L12 = light(V1, P1, V2, P2);
assert L23 = light(V2, P2, V3, P3);
assert L34 = light(V3, P3, V4, P4);
assert L41 = light(V4, P4, V1, P1);
```

Third, the E-Lucid language is based on the the original Lucid. It is worthwhile to extend the other variants of Lucid to develop other dataflow languages supporting Extended Dataflow Network, so that they can take the advantages of the variants. For example, we can introduce an extension of Indexical Lucid [25] to support Extended Dataflow Network, so that we can deal easily with multidimensional data structures such as arrays and trees [25].

Fourth, in Traffic Light Problem, it is obvious that the colors of the lights cycle with period of 4. It would be interesting to study new algorithms to identify such kind of dynamic problems and solve them completely, that is, to find the solution up to stage ∞. For example, by detecting the cycle of Traffic

Light Problem, we can completely solve the problem by finding the solution up to stage 8. The idea is explained in the following.

In Traffic Light Problem, there is a set of constraints $\bigcup_{t \in \mathcal{Z} \geq 0} \{ColorV[t+4] = ColorV[t]\}$. It has a pattern that all constraints of which are the same but different in the indices of daton variables. Given that "$ColorV[4] = ColorV[0]$" holds, it is obvious that "$ColorV[8] = ColorV[4]$" holds if $ColorV[8]$ and $ColorV[4]$ are assigned the same values as $ColorV[4]$ and $ColorV[0]$ respectively. The sufficient condition can be checked by finding the solution up to stage 8. In general, given that "$ColorV[n+4] = ColorV[n]$" holds, "$ColorV[n+4+p] = ColorV[n+p]$" holds if $ColorV[n+4+p]$ and $ColorV[n+p]$ are assigned the same values as $ColorV[n+4]$ and $ColorV[n]$ respectively. The sufficient condition can be checked by finding the solution up to stage $n+4+p$. We put $p = 4$, and we can find that $ColorV[t]$ is assigned the same value as $ColorV[t \texttt{ mod } p]$ with $0 \leq t \leq n$. We would like to prove that the valuation of $ColorV[t]$ cycles with period $p = 4$ by mathematical induction. Similarly, by observing the patterns of other sets of constraints in the problem, we would like to prove that the valuations of all daton variables are cyclic, it is, $v[t+p]$ is assigned the same value as $v[t]$ for all variables v and all positive integer t. With the proof, we can find a complete solution of the program.

We define that a solution θ of an E-Lucid program has a *period p since index t_0* if and only if $\theta(v[t+p]) = \theta(v[t])\ \forall v \in V, \forall t \in \mathcal{Z} \geq t_0$, where V is the set of variables of the program. If such period exists, θ is *cyclic*. For example, all the four solutions of Traffic Light Problem are cyclic with period 4 since index 0.

If a solution is cyclic with period p since index t_0, we can find the complete solution θ by finding the corresponding solution up to stage $(2p + t_0)$, ζ. The solution θ is equal to $\zeta \cup \bigcup_{v \in V, t \in \mathcal{Z} \geq p} \{v[t] \mapsto \zeta(v[((t - t_0) \texttt{ mod } p) + t_0])\}$.

Bibliography

[1] FAUSTINI A.A., MATHEWS S.G., and YAGHI A.G. The plucid programming manual. Technical report, Dept. of Computer Science, Univ. of Arizona, 1983.

[2] E.A. Ashcroft, A.A. Faustini, and R. Jagannathan. An intensional parallel processing language for applications programmin. Technical Report SRI-CSL-89-1, SRI International, Menlo Park, 1989.

[3] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. *Multi-dimensional Programming.* Oxford University Press, New York, 1995.

[4] S. A. Babiker, R. A. Fleming, and R. E. Milne. A tutorial for lts. Technical Report RR 225. 84. 1, Standard Telecommunication Laboratories, 1984.

[5] Dimitri P. Bertsekas. *Dyamic Programming and Optimal Control.* Athena Scientific, 1995.

[6] A.P.W. Bohm, editor. *Dataflow Computation*, volume 6 of *CWI Tract.* Centre for Mathematics and Computer Science, 1984.

[7] Manfred Broy. Functional specification of time-sensitive communicating systems. *ACM Transactions on Software Engineering and Methodology*, 2(1):1–46, 1993.

[8] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In ACM Press, editor,

14th ACM Conf. on Principles of Programming Languages, pages 178–188, 1987.

[9] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963.

[10] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proc. of AAAI-88*, pages 37–42, St. Paul, MN, 1988.

[11] Ashcroft E.A. and Wadge W.W. The syntax and semantics of lucid. Technical Report CSL-146, Computer Science Laboratory, SRI International, Menlo Park, 1984.

[12] A. A. Faustini and W. W. Wadge. An eductive interpreter for plucid. Technical Report TR-006-86, Department of Computer Science and Engineering, Arizona State University, 1986.

[13] E.A. Ashcroft. Ferds. massive parallelism in lucid. In IEEE, editor, *1985 Phoenix Computer and Communications Conference*, pages 16–2, March 1985.

[14] I.P. Gent and T. Walsh. Csplib: a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available from http://csplib.cs.strath.ac.uk/. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).

[15] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.

[16] J. R. Mc Graw. The val language: Description and analysis. *ACM TOPLAS*, 4(1), January 1982.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[18] ILOG. *ILOG Solver 4.4 Reference Manual*, 1999.

[19] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM Press, 1987.

[20] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The clp(r) language and system. *ACM Trans. Program. Lang. Syst.*, 14(3):339–395, 1992.

[21] Jagannathan. Dataflow models. In *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1996.

[22] R. Jagannathan. Adding eagerness to eduction.

[23] R. Jagannathan. Intensional and extensional graphical models for glu programming, 1996.

[24] R. Jagannathan and Chris Dodd. *GLU Programmer's Guide*. Computer Science Laboratory, SRI International 333 Ravenswood Avenue, Menlo Park, California 94025, U.S.A., 7 1994. Version 0.9.

[25] F. Jaggannathan. Multidimensional problem solving in lucid, 1993.

[26] Dennis J.B. Control flow and data flow: Concepts of distributed programming. In *Dataflow Computation*, volume 14 of *NATO ASI F: Computer and system sciences*, 1985.

[27] Mackworth A. K. Consistency in networks of relations. In *Artificial Intelligence*, pages 99–118, 1977.

[28] Al Kelley and Ira Pohl. *C by Dissection.* University of California, third edition, 1996.

[29] J. W. Lloyd. *Foundations of logic programming; (2nd extended ed.).* Springer-Verlag New York, Inc., 1987.

[30] David G. Luenberger. *Introduction to Dynamic Systems. Theory, Models, and Applications.* John Wiley and Sons, 1979.

[31] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* Massachusetts Institute of Technology, 1998.

[32] John McLeod. Rules of card games: Old maid. http://www.pagat.com/passing/oldmaid.html, 7 1996.

[33] John A. Plaice. Rlucid, a general real-time dataflow language. In Jan Vytopil, editor, *Formal Techniques in Real-time and Fault-tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, Nijmegan, the Netherlands, January 1992. Springer-Verlag.

[34] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.