LUCX: LUCID ENRICHED WITH CONTEXT

KaiYu Wan

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy

Concordia University

Montréal, Québec, Canada

April 2006

Your file Votre référence
ISBN: 978-0-494-16286-6
Our file Notre référence
ISBN: 978-0-494-16286-6

Abstract

Lucx: Lucid enriched with Context

KaiYu Wan, Ph.D.

Concordia University, 2006

Intensional logic is the mathematical foundation for Intensional Programming Languages (IPL). Lucid, initially founded on the dataflow paradigm, embraced intensional logic, and became a multi-dimensional intensional programming language. In all these developments *context* was the core concept. In its becoming an IPL, Lucid implicitly absorbed the notion of context, allowing expressions to be evaluated at different contexts. However, context cannot be explicitly named and manipulated in the current versions of Lucid. This restricts the ability of Lucid to be an effective programming language for programming diverse applications.

This thesis discusses the extension of Lucid with contexts as a first class object. That is, contexts can be defined, assigned values, used in expressions, and passed as function parameters. The language thus extended, is called *Lucx* (Lucid extended with *contexts*)(the x is used as the x in Latex). A context theory is developed to provide a semantic basis for context manipulation in Lucx. That is, contexts, context operators, and a context calculus are formally defined, and the formal syntax and semantics of Lucx are also given.

The benefits achieved by such an extension are illustrated by applying the extended language to program different applications including Timed Systems, Agent Communication, Constraint Programming, and in the formal development of context-aware systems.

Acknowledgments

I would like to express my deepest gratitude to Professor V. S. Alagar. His shrewdness and patience have been fundamental to the success of this work. Dr. Alagar has the ingenuity to bring together the scratches of thoughts and come out with a creative idea. His sound and extensive academic background has enriched my knowledge greatly. In addition to research coordination, his spiritual and literal background, together with utmost kindness, gave me a pleasant learning environment. For guiding me through this phase of my life and leading me towards the career life which I want, I am forever indebted to him.

I gratefully acknowledge Dr. Paquet for providing both technical and financial support for me. He is the person who initially gave me the opportunity of studying abroad, and led me to investigate the interesting language : Lucid. His confidence in me and open-minded discussions with me have encouraged me to perform at my best throughout the whole journey.

I would like to thank Professor Bill Wadge and members of my examining committee for their insightful comments which enabled me to revise the thesis to the current form.

My special thanks also go to the TROMLAB and GIPSY group members. The regular meetings inspired me a lot.

The unconditional love of my parents has always been a source of strength, pushing me to work hard and achieve my goals.

Finally, and above all, my gratitude goes to my husband XueJiang Pan who gives me endless love and incredible support. Without his sincere encouragement, none of this would have happened.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Intensional logic is the mathematical foundation for Intensional Programming Languages (IPL). Lucid, initially founded on the dataflow paradigm [67], embraced intensional logic, and became a multi-dimensional intensional programming language [6]. In all these developments *context* was the core concept. In its becoming an IPL, Lucid implicitly absorbed the notion of context, allowing expressions to be evaluated at different contexts. However, context cannot be explicitly named in the current version of Lucid. This thesis discusses an extension of Lucid with contexts as a first class object. The benefits achieved by such an extension are illustrated by applying the extended language to program different applications. The goal in this chapter is to provide the background material and a motivation for this work. We first review the role of context in Intensional Logic and the Intensional Programming Paradigm. Next we trace the evolution of Lucid language from a dataflow language to a multidimensional intensional programming language. Next, we discuss the motivation of this thesis. Finally, we enumerate the significant contributions of the thesis.

1.1 Intensional Logic

Intensional Logic [22, 58] , a family of mathematical formal systems that permits expressions whose value depends on *hidden context*, came into being from research in natural language understanding. According to Carnap [16], the real meaning of a natural language expression whose truth-value depends on the context in which it is uttered is its *intension*. The *extension* of that expression is its actual truth-value in the different possible contexts of utterance, where this expression can be evaluated. Basically, intensional logics add *dimensions* to logical expressions, and non-intensional logics can be viewed as *constant* in all possible dimensions, i.e. their valuation does not vary according to their context of utterance [46]. *Intensional operators* are defined to *navigate* in the context space. In order to navigate, some dimension *tags* (or indexes) are required to provide placeholders along dimensions. These dimension tags, along with the dimension names they belong to, are used to define the context for evaluating intensional expressions. For example, we can have an expression:

E: Beijing is now the capital of China.

This expression is intensional because the truth value of this expression depends on the context in which it is evaluated. The intensional natural language operators in this expression is *now*, which refers to the time dimension. Today it is certainly true, but there existed time points in the past when China had a different capital. For example, before 1949, the capital of China was NanJing. Those different values (i.e. True or False) along different time points are extensions of this expression. In other words, the evaluation of the above expression is time-dependent.

Naturally one can conclude that an expression may depend on more than one dimensions, such as time, space, audience, and so on. For example, the meaning of the expression

E': the average temperature this month here is greater than $0°C$.

can be obtained by considering its extension along the dimensions *month* and *location*. The table below gives partial extensions.

	Ja	Fe	Mr	Ap	Ma	Jn	Jl	Au	Se	Oc	No	De
Montreal	F	F	F	F	T	T	T	T	T	F	F	F
Ottawa	F	F	F	T	T	T	T	T	T	F	F	F
Toronto	F	F	T	T	T	T	T	T	T	T	F	F
Vancouver	F	T	T	T	T	T	T	T	T	T	T	T

1.2 Intensional Programming

The intensional programming paradigm has its foundations on intensional logic. It retains two aspects from intensional logic: first, at the syntactic level, are context-switching operators, called *intensional operators*; second, at the semantic level, is the use of *possible world semantics*.

By differentiating between intensions and extensions, IPL provides two different levels for programming. On the higher level, it allows us to represent/express problems in a declarative manner; On the lower level, it solves problems without loss of accuracy. The following is a list of some significant features of IPL:

- As IPL is based on solid mathematical foundations, i.e. intensional logic , it promotes a purer and more declarative way of programming than traditional imperative

languages.

- It deals with *infinite entities* of ordinary data values. Such entities could be a stream of numbers, a two-dimensional table of characters, a tree of strings etc. These streams are first class objects in intensional languages and functions can be applied to these streams.

- Because of the infinite nature of IPL, it is specially appropriate for describing the behavior of systems that change with time or physical phenomena that depend on more than one parameters (such as time, space, temperature, etc).

- As we know, the output of IPL programs may be infinite entities. Hence, the traditional approach can not be applied to those entities because it requires an infinite amount of time. However, the problem can be solved by using a computational model known as *eduction* [53]. That is, an implementation can start by computing the first element of the entity, then the second, and so on. This way, eduction deals with the problem arising due to the infinite nature of IPL without loss of accuracy.

1.3 Lucid

In this section we review several variants of Lucid family of languages.

Lucid as a dataflow language Lucid was originally invented as a *Program Verification Language* by Ashcroft and Wadge [4]. And later it evolved into a dataflow language [67]. The basic intensional operators are *first*, *next*, and *fby*. The four operators derived from the

basic ones are *wvr*, *asa*, *upon*, and *prev*, where wvr stands for *whenever*, asa stands for *as soon as*, upon stands for *advances upon*, and prev stands for *previous*. Lucid is a *stream* (i.e. infinite entity) manipulation language. All the above operators are applied to streams to produce new streams. The definitions of these operators [46] are shown as follows

Definition 1 *If* $X = (x_0, x_1, \ldots, x_i, \ldots)$ *and* $Y = (y_0, y_1, \ldots, y_i, \ldots)$, *then*

(1) $\underline{\texttt{first}}\ X \stackrel{\text{def}}{=} (x_0, x_0, \ldots, x_0, \ldots)$

(2) $\underline{\texttt{next}}\ X \stackrel{\text{def}}{=} (x_1, x_2, \ldots, x_{i+1}, \ldots)$

(3) $X\ \underline{\texttt{fby}}\ Y \stackrel{\text{def}}{=} (x_0, y_0, y_1, \ldots, y_{i-1}, \ldots)$

(4) $X\ \underline{\texttt{wvr}}\ Y \stackrel{\text{def}}{=} \texttt{if}\ \underline{\texttt{first}}\ Y\ \texttt{then}\ X\ \underline{\texttt{fby}}\ (\underline{\texttt{next}}\ X\ \underline{\texttt{wvr}}\ \underline{\texttt{next}}\ Y)$
$\texttt{else}\quad (\underline{\texttt{next}}\ X\ \underline{\texttt{wvr}}\ \underline{\texttt{next}}\ Y)$

(5) $X\ \underline{\texttt{asa}}\ Y \stackrel{\text{def}}{=} \underline{\texttt{first}}\ (X\ \underline{\texttt{wvr}}\ Y)$

(6) $X\ \underline{\texttt{upon}}\ Y \stackrel{\text{def}}{=} X\ \underline{\texttt{fby}}\ (\texttt{if}\ \underline{\texttt{first}}\ Y\ \texttt{then}\ (\underline{\texttt{next}}\ X\ \underline{\texttt{upon}}\ \underline{\texttt{next}}\ Y)$
$\texttt{else}\ (\quad X\ \underline{\texttt{upon}}\ \underline{\texttt{next}}\ Y))$

(7) $\underline{\texttt{prev}}\ X \stackrel{\text{def}}{=} X\ @\ (\# - 1)$ ∎

Example 1 illustrates the definitions on a stream A whose elements are integers, and a stream B whose elements are boolean. In a boolean stream the symbols 1 and 0 indicate true and false respectively. The symbol *nil* indicates an undefined value.

Example 1 *:*

A	$=$	1	2	3	4	5	...	
B	$=$	0	0	1	0	1	...	
first A	$=$	1	1	1	1	1	...	
next A	$=$	2	3	4	5	...		
prev A	$=$	*nil*	1	2	3	4	5	...
A *fby* B	$=$	1	0	0	1	0	1	...
A *wvr* B	$=$	3	5	...				
A *asa* B	$=$	3	3	3	...			
A *upon* B	$=$	1	1	1	3	3	5	...

Lucid as a Multidimensional Intensional Programming Language With the operators defined above, Lucid only allows sequential access into streams. That is, the $(i+1)th$ element in a stream is only computed once the *ith* element has been computed. To enable subcomputations to take place in arbitrary dimensions and all indexical operators to be parameterized by one or several dimensions, two basic intensional operators are added. One is *intensional navigation* (`@`$.d$), which allows the values of a stream to vary along the dimension d. Another is *intensional query* (`#`$.d$), which refers to the current position (i.e. tag value) along the dimension d. This way, it is possible to access streams randomly.

Example 2 illustrates the definitions of these two operators [34] on two streams A and B along the *time* dimension.

Example 2

A	$=$	1	2	4	8	16	32	64	128	...
B	$=$	1	2	3	0	6	7	4	5	...
A *@.time* B	$=$	2	4	8	1	64	128	16	32	...
#.time	$=$	0	1	2	3	4	5	6	7	...

Lucid has evolved into a *multidimensional intensional programming language* [6]. That is, programs are written while thinking of large, multidimensional streams (intensions), and yet the implementation deals only with small fragments (extensions). One of the characteristics of the current Lucid is that multidimensional streams are manipulated in the language with tuples of integers as tags and dimensions as first class values.

Although Lucid has gone through several stages in purpose and generality, the implementation technique of evaluation for the different Lucid versions is the interpreted mode called *eduction*. Eduction can be described as *tagged-token demand-driven dataflow*, in which data elements (tokens) are computed on demand following a dataflow network defined in Lucid. Data elements flow in the normal flow direction (from producer to consumer) and *demands* flow in the reverse order, both being *tagged* with their current context of evaluation. Below, two examples are given to illustrate programming problems in Lucid. The examples can be clearly understood from the syntax and semantics of Lucid, whose in-depth descriptions are shown in Chapter 3.

Example 3 : The Natural Numbers

The following program extracts a value from the stream representing the natural numbers,

beginning from the ubiquitous number 42. We arbitrarily pick the third value of the stream, which is assigned tag number two (indexes starting at 0). We also set the stream's variance in the *d* dimension.

```
N @.d 2
  where
  dimension d;
  N = 42 fby.d (N + 1);
  end;
```

The program can also be represented as a dataflow graph shown in Figure 1. Intuitively, we can expect the program to return the value 44. To see how the program is evaluated, we use the translation rules presented in [46] and rewrite the program in terms of the basic intensional operators @ and # as shown in Figure 2.

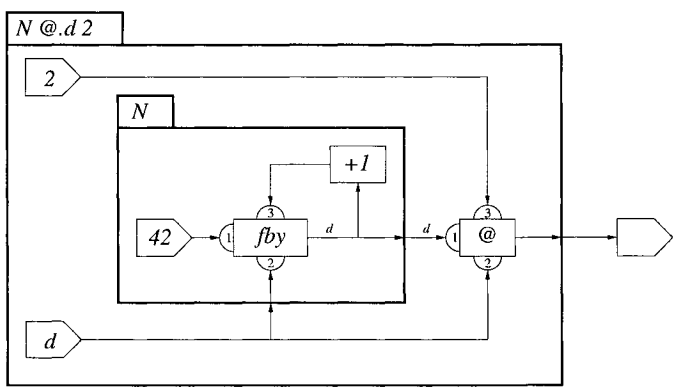

Figure 1: Dataflow graph for the natural numbers problem

In Figure 2, the evaluation takes place by generating successive demands for the appropriate values of N in different contexts, until the final computation can be affected. The demand

```
N @.d 2
  where
  dimension d;
  N = if (#.d ≤ 0) then 42 else (N + 1) @.d (#.d − 1);
  end;
```

Figure 2: Translated program for the natural numbers problem

for `N @.d 2` generates a demand for `N @.d 1` which in turn generates a demand for `N @.d 0`. The definition of the program explicitly states that the value of `N @.d 0` is 42. Once this is found, the successive addition operations are made on the demand results, as required by the equation `N = 42 fby.d N+1`, giving a final result of 44.

The most interesting feature of Lucid is its ability to naturally define and manipulate multidimensional streams. We illustrate this feature with a problem modeling heat transfer in a solid.

Example 4 *:* **Lucid Program for Heat Transfer Problem**

In this model the left end of a metal rod is touching a heat source with temperature 100. Initially, the temperature of the rod is 0. As the heat is transferred, the temperature at the various points of the rod changes. That is, the temperature depends on the time point and the spatial position on the rod, measured from the left end. The following equations compute the temperature of the rod as a function of time and space (where k is a small constant related to the physical properties of the rod)*:*

$$\mathtt{Temp}_{t+1,s+1} = \mathtt{k} \times \mathtt{Temp}_{t,s} - (1 - 2 \times \mathtt{k}) \times \mathtt{Temp}_{t,s+1} + \mathtt{k} \times \mathtt{Temp}_{t,s+2}$$

$$\mathtt{Temp}_{t,0} = 100$$

$$\mathtt{Temp}_{0,s+1} = 0$$

The Lucid program that models the above equations and queries the temperature at the

space 10 and time 10 is the following:

```
temp @.time 10 @.space 10
where
  dimension time, space;
  result = temp;
  temp = 100.0 fby.space (0.0 fby.time (k × temp−
          (1.0 − 2.0 × k) × (next.space temp)
          + k × (next.space next.space temp)));
  k = 0.3
end
```

From the above example, we can conclude the problem is expressed very concisely and naturally in Lucid language. The solution of such a problem in a traditional imperative language would most probably require the use of a bounded two-dimensional array together with the use of *for* loops in order to fill the entries of the array.

Different Variants of Lucid Lucid has been extended in several ways. Its variants have been used to specify 3D spreadsheets [23], attribute grammars [57], and database systems [53]. *Lustre*, as variant of Lucid, is a real-time reactive language which has been applied in developing commercial real-time systems, notably in aerospace [17]. *GLU* (Granular Lucid) is an industrial Lucid-C hybrid system which illustrates how the multidimensional structure of a problem expressed in Lucid can be harnessed to produce efficient parallel implementations of problems [35]. Currently, we are in the process of implementing

the GIPSY (General Intensional Programming System), which is an investigation platform (compiler, run-time environment, etc) for all members of the Lucid family of intensional programming languages [44].

The intensional version control approach described in [51] has recently found applications in the evolving area of Internet diversity development. One example in this domain is the development of the language IHTML(Intensional HTML), an extension of HTML which allows a single piece of source to specify a whole family of pages [69]. However IHTML is not a programming language, which restricts the flexibility of developing multi-version pages. Hence, ISE(Intensional Sequential Evaluator), a Perl-like scripting language that incorporates a runtime parametric version system, was developed by Paul Swoboda [55]. Based on that, IML(Internet Markup Language) was implemented as a front end for ISE [68], which has been used to develop multi-version pages [69].

1.4 Motivation of the Thesis

We are primarily motivated by Guha's work [29] on context, which according to him is a *rich concept* and *hard to define*. The meaning of "context" was only tacitly understood and used by researchers in several disciplines. In natural language processing [18], contexts arise as *situations* for interpreting natural language constructs. In programming languages, context is a meta-level concept to statically introduce constants, definitions, and constraints, and dynamically describe the executable information for evaluating expressions. Context

plays an important role in specification and modeling. In [19], the signature part, *CONSTRAINTS*, *SETS*, and *CONSTANTS* of a B component, is regarded as the *static* context of the component [2]. Proof obligations for internal consistency as well as for compositions of B components use context information. In modeling human-computer interaction [36], the context includes the *physical place* of the user, the *time constraints*, and the system's assumption about users interests. In Ubiquitous computing [18], context is understood as both *situated* and *environmental.* In Artificial Intelligence (AI), McCarthy [39] defined the formalization of the the notion of *context* as one of the main problems in the field of artificial intelligence (AI) and worked to formalize context and to develop a theory of introducing context as formal objects [40]. Later Guha [29] used the notion of context as a means of expressing assumptions made by natural language expressions. In particular, Guha's work addressed the limitations of the vocabulary and assumptions made within the traditional model of AI and introduced contexts in the statement of the theory to indicate explicitly that there is something left out. That is, contexts clearly extend known knowledge representation formalisms.

Guha embedded a syntactic representation of context into first order logic, and based on it developed a model theory and proof theory for contexts. An interesting part is the discussion and presentation of a general framework for *lifting* contextual knowledge. Lifting enables using formulas in one context to deduce formulas in another context. The lifting procedure involved performing a relative de-contextualization of the formula, i.e., the differences between the original and target context had to be taken into account to obtain a formula with the same truth conditions in both contexts. A set of axioms that act as lifting

rules are introduced.

The major distinction between contexts in AI and in IPL is that in the former case they are *rich objects* that are not *completely expressible* and in the later case they are *implicitly* expressible. Hence it is possible to write an expression in Lucid whose evaluation is context-dependent. However, a context in the current version of Lucid can not be explicitly manipulated. This restricts the ability of Lucid to be an effective programming language for programming diverse applications. So we have extended Lucid by adding the capability to explicitly manipulate contexts. This is achieved by introducing *context* as a first class object in the language. That is, contexts can be declared, assigned values, used in expressions, and passed as function parameters. The language thus extended, is called as *Lucx* (Lucid extended with *contexts*)(the x is used as the x in TeX). Thus, the rationale for introducing context in Lucid is quite analogous to the introduction of context to enrich knowledge base in AI. However, our notion of context differs significantly from McCarthy's. In our study context is both *finite* and *concrete*. It is finite in the sense that only a finite number of dimensions are allowed in defining a context. However it does not impose any limitation on handling infinite streams, because with every dimension an infinite *tag set* is introduced in the language. Thus, not all contexts studied by Guha can be dealt within our language. However, every context that we can define in *Lucx* is indeed a context in Guha's sense, but restricted to well-formed *Lucx* expressions. We define lifting as part of context theory. It plays an important role in programming solutions to problems in Lucx.

1.5 Major Contributions and Thesis Outline

The thesis investigates the merits of Lucx for programming diverse application domains by introducing context as first-class objects. In order to do that, a context theory is developed. This theory provides a semantic basis for context manipulation in Lucx. The major contributions are as follows (the references are for the published papers):

- Introducing context as first class objects in the language [10].
- Providing the formal syntax and semantics of Lucx [10, 63].
- Defining context operators and context calculus [10, 59], thus enabling dynamic manipulation of contexts in Lucx.
- Providing context theory [63].
- Demonstrating the use of the extended language for writing programs in different application domains including Programming Timed Systems [59], Agent Communication [10, 14], Constraint Programming [61], and in formal development of context-aware systems [9, 62].
- An architecture for integrating Lucx programming environment into GIPSY [60].

The thesis is organized as follows: in Chapter 2, *context, context operators, context expressions, a set of contexts, Boxes, operators for a set of contexts and context set expressions* are formally defined. *Formal syntax and semantics of Lucx* are given in Chapter 3. Chapter 4 discusses *rules of evaluation* for Lucx expression. The integration of Lucx into

GIPSY is shown in Chapter 5. Chapter 6 demonstrates the programming of timed systems in Lucx. The suitability of Lucx as an agent communication language is shown in Chapter 7. Chapter 8 illustrates how the constraint satisfaction problem can be represented and solved in Lucx. Chapter 9 concludes the thesis work and discusses the future research directions.

Chapter 2

A Formal Definition of Context

In this chapter, we give a formal definition of context and provide a context calculus so that contexts can be dynamically constructed and manipulated in Lucx.

In intensional programming, context is a reference to the representation of the "possible worlds" relevant to the current discussion. In Lucid, context cannot be defined; however, dimensions are defined, and the tag set associated with each dimension is implicitly $\mathbb{N}$, the set of natural numbers. Lucx extends these concepts of Lucid by associating a universal tag set to dimensions in a context defined in the language, and provides several context operators. We formalize context as *a relation*, set of ordered pairs of (d,x) where d is a dimension and x is a tag value.

Definition 2 *Let DIM denote the set of all possible dimensions, and U denote the set of all possible tags. A context c is a finite subset of the relation* $\{(d,x) \mid d \in DIM \ \wedge \ x \in U\}$. *The* degree *of the context c is* $\mid dom\ c \mid$. *The empty relation is a* Null *context. The degree of a Null context is 0.* ∎

Let G denote the set of all contexts that are defined according to Definition 2. A context having only one (dimension,tag) pair is called a *micro* context (or m_context). The set of *micro* contexts is $M = \{c \mid c \in G, \mid c \mid = 1\}$. The set of *simple* contexts (or s_context) is $S = \{c \mid c \in G, c \textit{ is a function}\}$. Clearly, a simple context c of degree 1 is a micro context. A context which is not simple is a *non-simple* context.

The basic functions *dim* and *tag* are to extract the set of dimensions and their associated tag values from a set of contexts.

Definition 3 $dim : G \rightarrow \mathbb{P}\, DIM \quad tag : G \rightarrow \mathbb{P}\, U$, *such that for* $c \in G$, $dim(c) = dom\ c$, *and* $tag(c) = ran\ c$. ∎

For the tuple $m = (d, x)$ we use the functions dim_m and tag_m to extract the tuple components: $dim_m(m) = d$ and $tag_m(m) = x$.

2.1 Context Calculus

In this section, context operators are discussed. A context being a relation we borrow the notation and meaning of those relational operators that are available in mathematics. Rest of them we define, using set theory notation. Using these context operators contexts can be managed dynamically and flexibly. The syntax of context expressions are also formally defined. In order to evaluate context expression correctly, precedence rules for context operators are provided as well.

2.1.1 Context Operators

Context operators are:*override* $\oplus$, *difference* $\ominus$, *choice* $|$, *conjunction* $\cap$, *disjunction* $\cup$, *undirected range* $\rightleftharpoons$, *directed range* $\rightharpoonup$, *projection* $\downarrow$, *hiding* $\uparrow$, *substitution* / , *comparison* $=$, $\supseteq$, $\subseteq$. The *difference* $\ominus$, *conjunction* $\cap$, *disjunction* $\cup$, and *comparison* $=$, $\subseteq$, $\supseteq$, operators are set operators. The rest of the operators are explained and formally defined below.

Definition 4 Override $\oplus$ *This operator takes two contexts* $c_1 \in G$, *and* $c_2 \in S$ *and returns a context* $c \in G$, *which is the result of the conflict-free union of* c_1 *and* c_2, *as defined below:*

$$\_ \oplus \_ : G \times S \rightharpoonup G,$$

$$c = c_1 \oplus c_2 = \{ m \mid (m \in c_1 \wedge dim_m(m) \notin dim(c_2)) \vee m \in c_2\} \quad \blacksquare$$

Definition 5 Choice $|$ *This operator accepts a finite number of* $c_1, \ldots, c_k$ *of contexts and nondeterministically returns one of the* c_i*s. The definition* $c = c_1 \mid c_2 \mid \ldots, \mid c_k$ *implies that* c *is one of the* c_i, *where* $1 \leq i \leq k$:

$$\_ \mid \_ : G \times G \times \ldots \times G \rightharpoonup G, \quad \blacksquare$$

Definition 6 *Projection. This operator takes a context* $c \in G$ *and a set of dimensions* $D \subseteq DIM$ *as arguments and filters only those ordered pairs in* c *that have their dimensions in set* D.

$$\_ \downarrow \_ : G \times \mathbb{P}\, DIM \rightharpoonup G,$$

$$c \downarrow D = \{m \mid m \in c \wedge dim_m(m) \in D\}. \quad \blacksquare$$

Example 5 *:*

Let $c_1 = \{(d,1),(e,4),(f,3)\}$, $D = \{\ d,\ e\ \}$

then $c_1 \downarrow D\ =\ \{(d,1),(e,4)\}$

Definition 7 *Hiding. This operator enables a set of dimensions* $D \subseteq DIM$ *to be applied on a context* $c \in G$ *to remove all the ordered pairs in c whose dimensions are in D:*

$\_ \uparrow \_ : G \times \mathbb{P}\,DIM \rightarrow G,$

$c \uparrow D = \{m \mid m \in c \wedge dim_m(m) \notin D\}.$ ∎

Using the precedence rules given in Figure 3 we can verify the properties for *Projection* and *Hiding* operators:

- $c \uparrow D \cap c \downarrow D = \varnothing$
- $c \uparrow D \cup c \downarrow D = c$
- $c \uparrow D = c \ominus (c \downarrow D)$

Example 6 *:*

Let $c_1\ =\ \{(d,1),(e,4),(f,3)\}$, $D = \{\ d,\ e\ \}$

then $c_1 \uparrow D\ =\ \{(f,3)\}$

Definition 8 *Substitution. This operator produces a context for a given context, a dimension and a tag value belonging to that dimension:*

$\_/\_ : G \times (DIM \times U) \rightarrow G,$

$c/\langle d', t'\rangle = \{m \mid m \in c\ \wedge\ dim_m(m) \neq d'\}\ \cup\ \{(d', t') \mid d' \in dim(c)\}.$ ∎

Example 7 :

Let $c_1 = \{(d,1),(e,4)\};\ c_2 = \{(e,4),(f,3)\}$,

then $c_1 \,/\, \langle d,\ 2\rangle = \{(d,2),(e,4)\};\ c_2 \,/\, \langle d,\ 2\rangle = c_2$

Definition 9 *Undirected range. This operator takes two contexts* $c_1,\ c_2 \in G$ *as arguments and returns a set of simple contexts. The fixed total ordering* $\leq$*, as defined for naturals, is assumed to be defined for the tag set U. We give a constructive definition here. In Chapter 6 the Lucid program implementing this operation is declarative.*

$\_ \rightleftharpoons \_ : G \times G \rightarrow \mathbb{P}\,S,$

Steps for constructing the final result are shown as follows:

1. *Let S' be the set of simple contexts, which is the result of* $(\ c_1 \rightleftharpoons c_2)$.

2. *For each pair of* $m_1 \in c_1$, $m_2 \in c_2$, *and* $dim_m(m_1) = dim_m(m_2)$, *do the following:*

   (a) *Define* $a = min\{tag_m(m_1), tag_m(m_2)\}$ *and* $b = max\{tag_m(m_1), tag_m(m_2)\}$

   (b) *Define the subrange* $t_a^b = a..b$.

   (c) *Construct the set* Y_1*:*

   $$Y_1 = \{(d_a^b, x) \mid d_a^b = dim_m(m_1) = dim_m(m_2), x \in t_a^b\}$$

3. $Y = \{Y_1, Y_2, \ldots, Y_p\}$, *where* $Y_i (i = 1, \ldots, p)$, *are the sets of micro context s constructed in Step 2. Define for* $Y_i \in Y$, $first(Y_i) = \{dim_m(m) \mid m \in Y_i\}$, *and* $second(Y_i) = \{tag_m(m) \mid m \in Y_i\}$. *If there exists* $Y_i, Y_j \in Y$ *such that* $first(Y_i) = first(Y_j)$, *for* $i \neq j$, *we replace the sets* Y_i *and* Y_j *by their union* $Y_i \cup Y_j$, *and repeat this process until the* $first(Y_i)$*s for* $Y_i \in Y$ *are distinct.*

4. *For $Y_i \in Y$, construct the set Z of contexts:* $Z = \{\{(first(Y_1), x_1), (first(Y_2), x_2), \ldots, (first(Y_p), x_p)\} \mid (x_1, x_2, \ldots, x_p) \in \Pi_{i=1}^{p} second(Y_i))\}$.

5. *Define:* $X_{c_1} = c_1 \uparrow \bigcup_{Y_i \in Y} first(Y_i)$.

6. *Define:* $X_{c_2} = c_2 \uparrow \bigcup_{Y_i \in Y} first(Y_i)$.

7. *Construct S':* $S' = \{\{z\} \cup X_{c1} \cup X_{c2} \mid z \in Z\}$. ∎

Basically, the result consists of three parts:

1. For each pair $m_1 \in c_1, m_2 \in c_2$ which share the same dimension, we construct a set Y_i. From the set of Y_is, constructed in step 2 and step 3, the set Z is computed.

2. All the other tuples of c_1 which have different dimensions are in X_{c1}.

3. Similarly, all the other tuples of c_2 which have different dimensions are in X_{c2}.

Example 8 *:*

Let $c_1 = \{(e,3),(d,1)\}$, $c_2 = \{(e,1),(d,3)\}$, $c_3 = \{(e,3)\}$, $c_4 = \{(f,4)\}$, $c_5 = \{(e,1),(f,4)\}$

then $c_1 \rightleftharpoons c_2 = \{ \{(e,1),(d,1)\}, \{(e,1),(d,2)\}, \{(e,1),(d,3)\}, \{(e,2),(d,1)\}, \{(e,2),(d,2)\}, \{(e,2),(d,3)\}, \{(e,3),(d,1)\}, \{(e,3),(d,2)\}, \{(e,3),(d,3)\} \}$

$c_3 \rightleftharpoons c_4 = \{\{(e,3),(f,4)\}\}$

$c_3 \rightleftharpoons c_5 = \{ \{(e,1),(f,4)\}, \{(e,2),(f,4)\}, \{(e,3),(f,4)\} \}$

Definition 10 *Directed Range. This operator takes two contexts $c_1, \in G$ and $c_2 \in S$ and returns a set of contexts:*

$\_ \rightharpoonup \_ : G \times S \rightarrow \mathbb{P}\, G,$

We change only Step 2 of the method described for the undirected range(Page 20) to obtain the result:

(a) Define $a = tag_m(m_1), b = tag_m(m_2)$, *if* $tag_m(m_1) < tag_m(m_2)$, *else we ignore the tag of this pair in further calculation and denote it by ? (don't care) and let* $Y_1 = (d, ?)$. *Hence in step 4 (of Definition 9) we ignore the pair (first(Y), second(Y)) if second(Y) = ?.*

(b) Define the subrange $t_a^b = a..b$. ∎

Example 9 *:*

Let $c_1 = \{(d,1)\}, c_2 = \{(d,3),(f,4)\},$

then $c_1 \rightharpoonup c_2 = c_1 \rightleftharpoons c_2 = \{\, \{(d,1),(f,4)\}, \{(d,2),(f,4)\}, \{(d,3),(f,4)\} \,\},$

$c_2 \rightharpoonup c_1 = \{(f,4)\}.$

2.1.2 Context Expression

Informally, a context expression is an expression involving context variables and context operators. Let c ranges over contexts, D over dimension sets, and C over context expressions. A formal syntax for context expression C is shown in Figure 3 (left column). A context expression that satisfies those syntactic rules is a well-formed context expression.

In order to provide a precise meaning for a context expression, we define the precedence rules for all the operators. Figure 3(right column) shows the operator precedence from the highest (top row) to the lowest (bottom row). Parentheses will be used to override this precedence when needed. Operators having the same precedence will be applied from left to right. The comparison operators $=, \subseteq, \supseteq$ have lowest precedence. Expressions

syntax				precedence
C	$::=$	c		
	$\mid$	$C \mid C$	$C/\langle d,t\rangle$	$\downarrow, \uparrow, /$
	$\mid$	$C \oplus C$	$C \ominus C$	$\mid$
	$\mid$	$C \cap C$	$C \cup C$	$\cap, \cup$
	$\mid$	$C \rightleftharpoons C$	$C \rightharpoonup C$	$\oplus, \ominus$
	$\mid$	$C \downarrow D$	$C \uparrow D$	$\rightleftharpoons, \rightharpoonup$

Figure 3: Formal Syntax of Context Expressions and Precedence Rules for Context Operators

consisting of comparison operators are boolean expressions instead of context expressions. Hence the syntax of these boolean expressions is not included in the above table.

Example 10 *Let c_1, c_2, c_3 be three contexts. The following equivalences hold according to the precedence defined above.*

1. $c_1 \downarrow \{d_1\} \mid c_2 = (c_1 \downarrow \{d_1\}) \mid c_2$
2. $c_1 \mid c_2 \downarrow \{d_1\} = c_1 \mid (c_2 \downarrow \{d_1\})$
3. $c_1 \mid c_2 \cap c_3 = (c_1 \cap c_3) \mid (c_2 \cap c_3)$
4. $c_1 \cap c_2 \mid c_3 = (c_1 \cap c_2) \mid (c_1 \cap c_3)$
5. $c_1 \oplus c_2 \cap c_3 = c_1 \oplus (c_2 \cap c_3)$
6. $c_1 \cap c_2 \oplus c_3 = (c_1 \cap c_2) \oplus c_3$
7. $c_1 \oplus c_2 \rightleftharpoons c_3 = (c_1 \oplus c_2) \rightleftharpoons c_3$
8. $c_1 \rightleftharpoons c_2 \oplus c_3 = c_1 \rightleftharpoons (c_2 \oplus c_3)$
9. $c_1 \mid c_2 \oplus c_3 = (c_1 \oplus c_3) \mid (c_2 \oplus c_3)$

10. $c_1 \oplus c_2 \mid c_3 = (c_1 \oplus c_2) \mid (c_1 \oplus c_3)$

11. $c_1 \mid c_2 \rightleftharpoons c_3 = (c_1 \rightleftharpoons c_3) \mid (c_2 \rightleftharpoons c_3)$

12. $c_1 \rightleftharpoons c_2 \mid c_3 = (c_1 \rightleftharpoons c_2) \mid (c_1 \rightleftharpoons c_3)$

13. $c_1 \cap c_2 \rightleftharpoons c_3 = (c_1 \cap c_2) \rightleftharpoons c_3$

14. $c_1 \rightleftharpoons c_2 \cap c_3 = c_1 \rightleftharpoons (c_2 \cap c_3)$

Example 11 *Given the context expression* $c_3 \uparrow D \oplus c_1 \mid c_2$, *where* $c_1 = \{(x,3), (y,4), (z,5)\}$, $c_2 = \{(y,5)\}$, *and* $c_3 = \{(x,5),(y,6),(w,5)\}$, $D = \{w\}$, *the evaluation steps are shown as follows:*

[Step1]. $c_3 \uparrow D = \{(x,5),(y,6)\}$ [Definition 7, Page 19]

[Step2]. $c_1 \mid c_2 = c_1$ *or* c_2 [Definition 5, Page 18]

[Step3]. Suppose in Step2, c_1 is chosen,

$c_3 \uparrow D \oplus c_1 = \{(x,3),(y,4),(z,5)\}$ [Definition 4, Page 18]

else if c_2 is chosen,

$c_3 \uparrow D \oplus c_2 = \{(x,5),(y,5)\}$ [Definition 4, Page 18]

2.2 Sets of Contexts

In this section we consider $\mathbb{P}\,S$, where S is the set of simple contexts defined in Section 2.1, and define operators on it. We introduce the special notation *Box* to denote a subset of $\mathbb{P}\,S$ such that all contexts in it have the same dimension set $\Delta \subseteq DIM$, and the tags at these dimensions satisfy a logical expression p. The *Box* notation may be viewed as

analogous to *schema* notation in Z. Our motivation to introduce *Box* comes from analyzing the requirements of a high level declarative language for programming timed systems and constraint solving problems. Using *Box* construct Lucx programs for such problems can be written in a precise and concise manner. This will become clear from the programming examples in Chapters 6 and 8.

In Lucx, *Boxes* and other sets of contexts are also first class objects. They can be assigned to variables. This will become clear when we discuss the syntax and semantics of Lucx in Chapter 3. A special interest in constraint solving problems is to consider only those operators that take *Box*es as arguments and produce a *Box* as the result. In view of this, we define three special operators on *Box*es.

Non-simple Contexts and Sets of Contexts Intuitively, a non-simple context can be viewed as a set of simple contexts. Definition 11 formalizes this point of view. In Lucx, an @ expression is evaluated at a non-simple context by evaluating the expression at the simple contexts in the set representation of the non-simple context. Hence, the result of evaluation is a set. We discuss this aspect in the Chapter 3.

Definition 11 *For a non-simple context c, we construct the set* $Y = \{y_d = c \downarrow \{d\} \mid d \in dim(c)\}$. *Denoting the elements of set Y as* $y_1, \ldots, y_p$, *we construct the set* $\mathcal{S}(c)$ *of simple contexts:*

$$\mathcal{S}(c) = \{m_1 \oplus m_2 \oplus \ldots \oplus m_p \mid m_1 \in y_1 \wedge m_2 \in y_2 \wedge \ldots \wedge m_p \in y_p\},$$

The non-simple context is viewed as the set $\mathcal{S}(c)$. *It is easy to see that*

$$\mathcal{S}(c) = \{s \in S \mid dim(s) = dim(c) \wedge s \subset c\} \quad \blacksquare$$

Example 12 *:*

Let $c = \{(d,1),(d,2),(d,5),(e,2),(e,6)\}$, $dim(c) = \{d,e\}$,

$y_d = c \downarrow \{d\} = \{(d,1),(d,2),(d,5)\}$,

$y_e = c \downarrow \{e\} = \{(e,2),(e,6)\}$.

then $\mathcal{S}(c) = \{\{(d,1),(e,2)\},\{(d,1),(e,6)\},\{(d,2),(e,2)\},\{(d,2),(e,6)\},\{(d,5),(e,2)\},$

$\{(d,5),(e,6)\}\}$.

Extending the above construction to a set of non-simple contexts we end up dealing with higher-order sets. In Lucx we avoid higher-order sets of contexts, and allow only sets of simple contexts. Hereafter, by "set of contexts" we refer only to "set of simple contexts".

2.3 Context Set Operators

There are two kinds of such operators: *lifting* operators, and *relational* operators.

2.3.1 Lifting Operators

Definition 12 *Projection. For $s \in \mathbb{P}\,S$, and $D \subseteq DIM$. the projection operator constructs a set of contexts $s' \in \mathbb{P}\,S$, where s' is obtained by projecting each context from s on to the dimension set D.*

$\_ \downarrow \_ : \mathbb{P}\,S \times \mathbb{P}\,DIM \rightarrow \mathbb{P}\,S$

$s' = s \downarrow D = \{c \downarrow D \mid c \in s\}$ ∎

Example 13 *:*

Let $s_1 = \{\{(x,1),(y,2)\},\{(x,1),(z,3)\},\{(y,4),(z,3)\},\{(y,4)\}\}$, $D = \{y,z\}$

Then $s_1 \downarrow D = \{\{(y,2)\},\{(z,3)\},\{(y,4),(z,3)\},\{(y,4)\}\}$

Definition 13 *Hiding. For $s \in \mathbb{P}\,S$, and $D \subseteq DIM$. The hiding operator constructs $s' \in \mathbb{S}_2$, where s' is obtained by hiding each context in s on the dimension set D.*

$$\_ \uparrow \_ : \mathbb{P}\,S \times \mathbb{P}\,DIM \rightarrow \mathbb{P}\,S$$

$$s' = s \uparrow D = \{c \uparrow D \mid c \in s\} \quad \blacksquare$$

Example 14 *:*

Let $s_1 = \{\{(x,1),(y,2)\},\{(x,2),(z,3)\},\{(y,4),(z,3)\},\{(y,4)\}\}$, $D = \{y,z\}$

Then $s_1 \uparrow D = \{\{(x,1)\},\{(x,2)\},NULL\}$

Definition 14 *Substitution. This operator produces a set of contexts s', $s' \in \mathbb{P}\,S$, for a given set of contexts s, $s \in \mathbb{P}\,S$, a dimension and a tag value belonging to that dimension:*

$$\_/\_ : \mathbb{P}\,S \times (DIM \times U) \rightarrow \mathbb{P}\,S$$

$$s' = s/\langle d', t' \rangle = \{c/\langle d', t' \rangle \mid c \in s\} \quad \blacksquare$$

Example 15 *:*

Let $s_1 = \{\{(x,1),(y,2)\},\{(x,2),(z,3)\},\{(y,4),(z,3)\},\{(y,4)\}\}$

Then $s_1/\langle x,2\rangle = \{\{(x,2),(y,2)\},\{(x,2),(z,3)\},\{(y,4),(z,3)\},\{(y,4)\}\}$

Definition 15 *Choice. This operator accepts two sets of contexts s_1, s_2 and nondeterministically returns one of them. The definition $s = s_1 \mid s_2$ implies that s is either s_1 or s_2.*

$$\_ \mid \_ : \mathbb{P}\,S \times \mathbb{P}\,S \rightarrow \mathbb{P}\,S \quad \blacksquare$$

Definition 16 *Override. For every pair of context sets s_1, s_2, $s_1, s_2 \in \mathbb{P}\,S$ this operator returns a set of contexts s, $s \in \mathbb{P}\,S$, where every context $c \in s$ is computed as $c_1 \oplus c_2$, $c_1 \in s_1, c_2 \in s_2$.*

$$\_ \oplus \_ : \mathbb{P}\,S \times \mathbb{P}\,S \rightarrow \mathbb{P}\,S$$

$$s = s_1 \oplus s_2 = \{c_1 \oplus c_2 \mid c_1 \in s_1 \wedge c_2 \in s_2\} \quad \blacksquare$$

Example 16 *:*

Let $s_1 = \{\{(x,1),(y,2)\},\{(x,2),(z,3)\},\{(y,4),(z,3)\},\{(y,4)\}\}$,

$s_2 = \{\{(y,2),(z,3)\},\{(x,2),(y,3)\}\}$

Then $s_1 \oplus s_2 = \{\{(x,1),(y,2),(z,3)\},\{(x,2),(y,2),(z,3)\},\{(y,2),(z,3)\},$

$\{(x,2),(y,3)\},\{(x,2),(y,3),(z,3)\}\}$

Definition 17 *Difference. For every pair of context sets* s_1, s_2, $s_1, s_2 \in \mathbb{P}S$ *this operator returns a set of contexts* s, $s \in \mathbb{P}S$, *where every context* $c \in s$ *is computed as* $c_1 \ominus c_2$, $c_1 \in s_1, c_2 \in s_2$.

$$\_ \ominus \_ : \mathbb{P}S \times \mathbb{P}S \rightarrow \mathbb{P}S$$

$$s = s_1 \ominus s_2 = \{c_1 \ominus c_2 \mid c_1 \in s_1 \wedge c_2 \in s_2\} \quad \blacksquare$$

Example 17 *:*

Let $s_1 = \{\{(x,1),(y,2)\},\{(x,2),(z,3)\},\{(y,4),(z,3)\},\{(y,4)\}\}$,

$s_2 = \{\{(y,2),(z,3)\},\{(x,2),(y,3)\}\}$

Then $s_1 \ominus s_2 = \{\{(x,1)\},\{(x,2)\},\{(y,4)\},\{(x,1),(y,2)\},\{(z,3)\},\{(y,4),(z,3)\}\}$

Lifting the *undirected range* $\rightleftharpoons$ and *directed range* $\rightharpoonup$ to sets of contexts will produce higher-order sets. So, we do not define lifting for these two operators. However, since the results of applying these two operators are sets of contexts, the lifting operators can be applied to the results.

2.3.2 Relational Operators

We define the three relational operations $\boxtimes$ (join), $\boxdot$(intersection), and $\boxplus$ (union) for sets of contexts. In the following definitions, c denotes a context, $s_i \in \mathbb{P}S$ and $\Delta_i = \bigcup_{c' \in s_i} dim(c')$.

Definition 18 *Join.*

$\_ \boxtimes \_ : \mathbb{P}\, S \times \mathbb{P}\, S \rightarrow \mathbb{P}\, S$

$s = s_1 \boxtimes s_2 = \{c_1 \cup c_2 \mid c_1 \in s_1 \wedge c_2 \in s_2 \wedge c_1 \downarrow \Delta_3 = c_2 \downarrow \Delta_3\}$

where $\Delta_3 = \Delta_1 \cap \Delta_2$. ∎

Example 18 *:*

Let $s_1 = \{\{(x,1),(y,2)\},\{(x,2),(z,3)\},\{(y,4),(z,3)\},\{(y,4)\}\}$, $\Delta_1 = \{x,y,z\}$

$s_2 = \{\{(y,2),(u,1)\},\{(y,5),(u,2)\},\{(y,3)\},\{(y,4)\}\}$, $\Delta_2 = \{y,u\}$, $\Delta_3 = \{y\}$

Then $s_1 \boxtimes s_2 = \{\{(x,1),(y,2),(u,1)\},\{(y,4),(z,3)\},\{(y,4)\}\}$

Definition 19 *Intersection.*

$\_ \boxdot \_ : \mathbb{P}\, S \times \mathbb{P}\, S \rightarrow \mathbb{P}\, S$

$s = s_1 \boxdot s_2 = \{c_1 \cap c_2 \mid c_1 \in s_1 \wedge c_2 \in s_2\}$. *We can prove that* $s = s_1 \boxdot s_2 = (s_1 \boxtimes s_2) \downarrow \Delta_3$,

where $\Delta_3 = \Delta_1 \cap \Delta_2$. ∎

Example 19 *:*

Let $s_1 = \{\{(x,1),(y,2)\},\{(x,2),(z,3)\},\{(y,4),(z:3)\},\{(y,4)\}\}$, $\Delta_1 = \{x,y,z\}$

$s_2 = \{\{(y,2),(u,1)\},\{(y,5),(u,2)\},\{(y,3)\},\{(y,4)\}\}$, $\Delta_2 = \{y,u\}$, $\Delta_3 = \{y\}$

Then $s_1 \boxdot s_2 = \{\{(y,2)\},\{(y,4)\}\}$

Definition 20 *Union.*

$\_ \boxplus \_ : \mathbb{P}\, S \times \mathbb{P}\, S \rightarrow \mathbb{P}\, S$

$s = s_1 \boxplus s_2$ *is computed as follows:*

$\Delta_1 = \bigcup_{c \in s_1} dim(c)$, $\Delta_2 = \bigcup_{c \in s_2} dim(c)$, *and* $\Delta_3 = \Delta_1 \cap \Delta_2$.

1. Compute X_1*:* $X_1 = \{c_i \cup c_j \uparrow \Delta_3 \mid c_i \in s_1 \wedge c_j \in s_2\}$

2. Compute X_2*:* $X_2 = \{c_j \cup c_i \uparrow \Delta_3 \mid c_i \in s_1 \wedge c_j \in s_2\}$

3. The result is: $s = X_1 \bigcup X_2$. ∎

Example 20 *:*

Let $s_1 = \{\{(x,1),(y,2)\},\{(x,2),(z,3)\},\{(y,4),(z:3)\},\{(y,4)\}\}$, $\Delta_1 = \{x,y,z\}$

$s_2 = \{\{(y,2),(u,1)\},\{(y,5),(u,2)\},\{(y,3)\},\{(y,4)\}\}$, $\Delta_2 = \{y,u\}$, $\Delta_3 = \{y\}$

$X_1 = \{\{(x,1),(y,2),(u.,1)\},\{(x,1),(y,2),(u,2)\},\{(x,1),(y,2)\},\{(x,2),(z,3),(u,1)\},$

$\{(x,2),(z,3),(u,2)\},\{(x,2),(z,3)\},\{(y,4),(z,3),(u,1)\},\{(y,4),(z,3),(u,2)\},$

$\{(y,4),(z,3)\},\{(y,4),(u,1)\},\{(y,4),(u,2)\},\{(y,4)\}\}$

$X_2 = \{\{(x,1),(y,2),(u,1)\},\{(x,2),(y,2),(z,3),(u,1)\},\{(y,2),(u,1),(z,3)\},\{(y,2),(u,1)\},$

$\{(x,1),(y,5),(u,2)\},\{(x,2),(y,5),(z,3),(u,2)\},\{(y,5),(u,2),(z,3)\},\{(y,5),(u,2)\},$

$\{(x,2),(y,3),(z,3)\},\{(y,3),(z,3)\},\{(y,3)\},\{(x,1),(y,3)\},\{(x,2),(y,4),(z,3)\},$

$\{(x,1),(y,4)\},\{(y,4),(z,3)\},\{(y,4)\}\}$

Then $s_1 \boxplus s_2 = \{\{(u,1),(x,1),(y,2)\},\{(u,1),(x,2),(z,3)\},\{(u,1),(y,4),(z,3)\},\{(y,3)\},$

$\{(u,1),(y,4)\},\{(u,2),(x,2),(z,3)\},\{(u,2),(y,4),(z,3)\},\{(u,2),(y,4)\},$

$\{(u,2),(x,1),(y,2)\},\{(u,2),(x,1),(y,5)\},\{(x,1),(y,3)\},\{(x,1),(y,4)\},$

$\{(u,1),(x,2),(y,2),(z,3)\},\{(u,2),(x,2),(y,5),(z,3)\},\{(x,2),(y,3),(z,3)\},$

$\{(x,2),(y,4),(z,3)\},\{(u,1),(y,2),(z,3)\},\{(u,2),(y,5),(z,3)\},\{(y,3),(z,3)\},$

$\{(y,4),(z,3)\},\{(x,1),(y,2)\},\{(x,2),(z,3)\},\{(y,4)\},\{(y,2),(u,1)\},\{(y,5),(u,2)\}\}$

In Section 2.1, it is shown that the results of $c_i \rightleftharpoons c_j$ and $c_i \rightharpoonup c_j$ are sets of contexts. So the relational operators $\boxtimes$ (join), $\boxdot$ (intersection), and $\boxplus$ (union) can also be applied to the expressions $c_i \rightleftharpoons c_j$ and $c_i \rightharpoonup c_j$, where c_i and c_j are contexts. Example 21 illustrates the procedure of applying those operators.

Example 21 :

Let $c_1 = \{(d,1)\}$, $c_2 = \{(d,3),(f,4)\}$, $c_3 = \{(d,1),(e,4)\}$, *and* $c_4 = \{(d,3),(e,6)\}$,

we calculate the sets $s_1 \boxtimes s_2$, $s_1 \boxdot s_2$, *and* $s_1 \boxplus s_2$, *where*

$s_1 = c_3 \rightharpoonup c_4$, $\Delta_1 = \{d,e\}$, *and* $s_2 = c_1 \rightharpoonup c_2$, $\Delta_2 = \{d,f\}$, *Then* $\Delta_3 = \{d\}$

Hence $s_1 = \{\{(d,1),(e,4)\},\{(d,1),(e,5)\},\{(d,1),(e,6)\},\{(d,2),(e,4)\},$

$\{(d,2),(e,5)\},\{(d,2),(e,6)\},\{(d,3),(e,4)\},\{(d,3),(e,5)\},\{(d,3),(e,6)\}\}$

$s_2 = \{\{(d,1),(f,4)\},\{(d,2),(f,4)\},\{(d,3),(f,4)\}\}$

$s_1 \boxtimes s_2 = \{\{(d,1),(e,4),(f,4)\},\{(d,1),(e,5),(f,4)\},\{(d,1),(e,6),(f,4)\},$

$\{(d,2),(e,4),(f,4)\},\{(d,2),(e,5),(f,4)\},\{(d,2),(e,6),(f,4)\},$

$\{(d,3),(e,4),(f,4)\},\{(d,3),(e,5),(f,4)\},\{(d,3),(e,6),(f,4)\}\}$

$s_1 \boxdot s_2 = \{\{(d,1)\},\{(d,2)\},\{(d,3)\}\}$

$s_1 \boxplus s_2 = \{\{(d,1),(e,4),(f,4)\},\{(d,1),(e,5),(f,4)\},\{(d,1),(e,6),(f,4)\},$

$\{(d,2),(e,4),(f,4)\},\{(d,2),(e,5),(f,4)\},\{(d,2),(e,6),(f,4)\},$

$\{(d,3),(e,4),(f,4)\},\{(d,3),(e,5),(f,4)\},\{(d,3),(e,6),(f,4)\}\}$

2.3.3 Context Set Expressions

Informally, a context set expression is an expression involving sets of contexts and context set operators. Let s ranges over a set of contexts, S over a context set expression and D over a dimension set. A formal syntax for context set expression S is shown in Figure 4 (left column).

In order to precisely calculate a context set expression, we define the precedence rules for the context set operators. These are shown in Figure 4 (right column) (from the highest precedence at the to row to the lowest precedence in the bottom row). Parentheses will be

syntax			precedence
S ::=	s	$S \mid S$	$\downarrow, \uparrow, /$
	$S \oplus S$	$S \ominus S$	$\mid$
	$S \downarrow D$	$S \uparrow D$	$\oplus, \ominus$
	$S \boxtimes S$	$S \boxdot S$	$\boxtimes, \boxdot, \boxplus$
	$S \boxplus S$	$S/\langle d, t \rangle$	

Figure 4: Formal Syntax of Context Set Expressions and Precedence Rules for Context Set Operators

used to override this precedence when needed. Operators having the same precedence will be applied from left to right.

2.4 *Box* Notation

In many applications it is of special interest to consider a set of contexts, all of which have the same dimension set and the tags corresponding to the dimensions in each context satisfy a given constraint. We use the notation *Box* to denote such a set when the dimension set is $\Delta = \{d_1, \ldots, d_k\} \subset DIM$ and p is a logical expression. Note that in p, we are allowing the dimensions as variables, denoting the current tags. That is, if $p(d_1, d_2) = d_1 < d_2$, it means the current tag of d_1 is less than the current tag of d_2 in the context that has dimensions d_1 and d_2. A formal definition follows:

Definition 21 *A Box set (or a Box for short) is a set of simple contexts with the same domain. Let* $\varnothing \neq \{d_1, \ldots, d_k\} \subseteq DIM$ *be a set of dimensions and p be an expression in which the* d_i $(1 \leq i \leq k)$ *may occur as variables. Then* $Box[d_1, \ldots, d_k \mid p] = \{c \in S \mid dim(c) = \{d_1, \ldots, d_k\}$ *and p is true when, for each* i, d_i *is assigned the value* $c(d_i)\}$. *The*

dimension $\Delta(b)$ of an nonempty box b is the dimension of any (all) its elements. ∎

The set of Box sets (or Boxes for short) are all sets of simple contexts all of which have the same domain. It is easy to show that anything defined by the Box notation is a Box.

Example 22 *Let $\Delta = \{X\}$, and prime(X) be the predicate that is true when the tag associated with the dimension X is a prime number. Box$[X \mid prime(X) \wedge 2 \leq X \wedge X < 12]$ is the set of contexts $\{\{(X,2)\},\{(X,3)\},\{(X,5)\},\{(X,7)\},\{(X,11)\}\}$.*

Example 23 *:*

Let $\Delta = \{U, X\}$.

Box$[X, U \mid \frac{X}{4} + \frac{U}{5} \leq 1] = \{\{(X,0),(U,0)\},\{(X,0),(U,1)\},\{(X,0),(U,2)\},$

$\{(X,0),(U,3)\},\{(X,0),(U,4)\},\{(X,0),(U,5)\},\{(X,1),(U,0)\},\{(X,1),(U,1)\},$

$\{(X,1),(U,2)\},\{(X,1),(U,3)\},\{(X,2),(U,0)\},\{(X,2),(U,1)\},\{(X,2),(U,2)\},$

$\{(X,3),(U,0)\},\{(X,3),(U,1)\},\{(X,4),(U,0)\}\}$

Since a *Box* is a special set of contexts, operators defined in Section 2.3 can be used on *Box*es. However, after applying those operators, the result of applying those operators may not be a *Box*. The operators that gives *Box* as result are called *Box* operators. Let $\mathbb{B}$ denote the set of Boxes introduced in Definition 21.

2.4.1 Box Operators

In this section we state lemmas on those context set operators that are *Box* operators.

Lemma 1 *Projection. The result of projecting a Box b on $D \subseteq DIM$ is a Box b'.* ∎

Proof : By definition 12 $b' = b \downarrow D = \{c \downarrow D \mid c \in b\}$. b' is obviously a box because all the elements have the dimension $\Delta \cap D$, and these elements are all simple contexts (because the projection of a simple context is simple).

Lemma 2 *Hiding. The result of hiding a Box b on a set of dimensions D is a Box.* ∎

Proof : Since $\downarrow$ and $\uparrow$ are complementary operations, $b \uparrow D$ is also a *Box*.

Lemma 3 *Choice. This operator accepts a finite number $b_1, \ldots, b_k$ of Boxes and nondeterministically returns one of the b_is. The definition $b = b_1 \mid b_2 \mid \ldots \mid b_k$ implies that b is one of the b_i, where $1 \leq i \leq k$.* ∎

Relational Operators

Let b_1 and b_2 be two *Box*es. Since a *Box* is a set of contexts, the definitions [18, 19, 20] (Page 29) for relational operators for sets of contexts can be used in a straightforward manner. However, it is natural to expect the result of applying relational operators to *Box* operands is a *Box*. This is in fact true, as stated below.

Lemma 4 *Join. Let b_1 and b_2 be two Boxes, $\Delta(b_1) = \Delta_1$, and $\Delta(b_2) = \Delta_2$. $b_1 \boxtimes b_2 = b$ is a Box, where $\Delta(b) = \Delta_1 \cup \Delta_2$.* ∎

Proof : According to the Definition 18, if c_1 and c_2 are simple and they agree on Δ_3, their union must be simple. And they all have domain $\Delta_1 \cup \Delta_2$.

It is easy to show that the Join of the boxes $Box[d_{11}, d_{12}, \ldots, d_{1n} \mid p_1]$ and $Box[\, d_{21},\ d_{22},\ \ldots, d_{2m} \mid p_2\,]$ is $Box[e_1, e_2, \ldots, e_k \mid p_1 \wedge p_2]$, where $\{e_1, e_2, \ldots, e_k\}$ is the union of $\{d_{11}, d_{12}, \ldots, d_{1n}\}$ and $\{d_{21}, d_{22}, \ldots, d_{2m}\}$.

Below we state the lemmas for intersection and union rules without proofs.

Lemma 5 *Intersection. Let b_1 and b_2 be two Boxes, $\Delta(b_1) = \Delta_1$, and $\Delta(b_2) = \Delta_2$. $b_1 \boxdot b_2 = b$ is a Box, where $\Delta(b) = \Delta_1 \cap \Delta_2$.* ∎

It is easy to show that the Intersection of the boxes $Box[d_{11}, d_{12}, \ldots, d_{1n} \mid p_1]$ and $Box[d_{21}, d_{22}, \ldots, d_{2m} \mid p_2]$ is $Box[f_1, f_2, \ldots, f_l \mid \exists X_i \in ((\Delta_1 - \Delta_2) \cup (\Delta_2 - \Delta_1)) \bullet (p_1 \wedge p_2)]$, where $\{f_1, f_2, \ldots, f_l\}$ is the intersection of $\{d_{11}, d_{12}, \ldots, d_{1n}\}$ and $\{d_{21}, d_{22}, \ldots, d_{2m}\}$.

It is easy to prove $b_1 \boxdot b_2 = b_1 \boxtimes b_2 \downarrow (\Delta_1 \cap \Delta_2)$.

Lemma 6 *Union. Let b_1 and b_2 be two Boxes, $\Delta(b_1) = \Delta_1$, and $\Delta(b_2) = \Delta_2$. $b = b_1 \boxplus b_2$ is a Box, where $\Delta(b) = \Delta_1 \cup \Delta_2$.* ∎

It is easy to show that the Union of the boxes $Box[d_{11}, d_{12}, \ldots, d_{1n} \mid p_1]$ and $Box[d_{21}, d_{22}, \ldots, d_{2m} \mid p_2]$ is $Box[e_1, e_2, \ldots, e_k \mid p_1 \vee p_2]$, where $\{e_1, e_2, \ldots, e_k\}$ is the union of $\{d_{11}, d_{12}, \ldots, d_{1n}\}$ and $\{d_{21}, d_{22}, \ldots, d_{2m}\}$.

Example 24 *:*

Let $DIM = \{X, Y, Z\}$.

$B_1 = Box[X, Y \mid X + Y = 5]$.

$B_2 = Box[Y, Z \mid Y = Z^2 \wedge Z \leq 3]$,

Then $B_1 = \{\{(X,1),(Y,4)\}, \{(X,2),(Y,3)\}, \{(X,3),(Y,2)\}, \{(X,4),(Y,1)\}\}$,

$B_2 = \{\{(Y,1),(Z,1)\}, \{(Y,4),(Z,2)\}, \{(Y,9),(Z,3)\}\}$

$B_1 \boxplus B_2 = Box[X, Y, Z \mid X + Y = 5 \vee (Y = Z^2 \wedge Z \leq 3)]$

$= \{\{(X,1),(Y,4),(Z,1)\}, \{(X,1),(Y,4),(Z,2)\}, \{(X,1),(Y,4),(Z,3)\},$

$\{(X,2),(Y,3),(Z,1)\}, \{(X,2),(Y,3),(Z,2)\}, \{(X,2),(Y,3),(Z,3)\},$

$$\{(X,3),(Y,2),(Z,1)\},\{(X,3),(Y,2),(Z,2)\},\{(X,3),(Y,2),(Z,3)\},$$

$$\{(X,4),(Y,1),(Z,1)\},\{(X,4),(Y,1),(Z,2)\},\{(X,4),(Y,1),(Z,3)\},$$

$$\{(X,1),(Y,1),(Z,1)\},\{(X,2),(Y,1),(Z,1)\},\{(X,3),(Y,1),(Z,1)\},$$

$$\{(X,2),(Y,4),(Z,2)\},\{(X,3),(Y,4),(Z,2)\},\{(X,4),(Y,4),(Z,2)\},$$

$$\{(X,1),(Y,9),(Z,3)\},\{(X,2),(Y,9),(Z,3)\},\{(X,3),(Y,9),(Z,3)\},$$

$$\{(X,4),(Y,9),(Z,3)\}\}$$

$$B_1 \boxtimes B_2 = Box[X,Y,Z \mid X+Y=5 \wedge (Y=Z^2 \wedge Z \leq 3)]$$

$$= \{\{(X,1),(Y,4),(Z,2)\},\{(X,4),(Y,1),(Z,1)\}\}$$

$$B_1 \boxdot B_2 = Box[Y \mid X+Y=5 \wedge (Y=Z^2 \wedge Z \leq 3)]$$

$$= \{\{(Y,1)\},\{(Y,4)\}\}$$

2.4.2 *Box* Expressions

A *Box* expression is an expression involving only *Box* variables and *Box* operators. A formal syntax for *Box* expressions B is extracted from Figure 4 and is shown in Figure 5. A *Box* expression satisfying these syntactic rules is a *Box* expression.

syntax				precedence
B	::= b	$B \mid B$		$\downarrow, \uparrow$
	$B \boxdot B$	$B \boxtimes B$		$\mid$
	$B \boxplus B$	$B \downarrow D$		$\boxdot, \boxplus, \boxtimes$
	$B \uparrow D$			

Figure 5: Formal Syntax of *Box* Expression and Precedence Rules for *Box* Operators

Example 25 *Let b_1, b_2, b_3 be three Boxes. The following equivalences hold according to the precedence defined above.*

1. $b_1 \downarrow D_1 \mid b_2 = (b_1 \downarrow D_1) \mid b_2$

2. $b_1 \mid b_2 \downarrow D_1 = b_1 \mid (b_2 \downarrow D_1)$

3. $b_1 \mid b_2 \boxtimes b_3 = (b_1 \boxtimes b_3) \mid (b_2 \boxtimes b_3)$

4. $b_1 \boxtimes b_2 \mid b_3 = (b_1 \boxtimes b_2) \mid (b_1 \boxtimes b_3)$

5. $b_1 \downarrow D_1 \boxtimes b_2 = (b_1 \downarrow D_1) \boxtimes b_2$

6. $b_1 \boxtimes b_2 \downarrow D_1 = b_1 \boxtimes (b_2 \downarrow D_1)$

2.5 Summary

In this chapter, context is formally defined as a relation over dimension and tag sets. To manipulate contexts dynamically, context operators are introduced. Their formal definitions are illustrated with examples. Precedence rules for context operators are provided. The formal syntax of context expressions is introduced. Context expressions are evaluated according to the precedence rules.

Sets of contexts are formally defined as well. It is shown that some of the context operators can be lifted into context set operators. In addition, three relational operators are defined formally. The short hand notation *Box* for special sets of contexts is introduced. Some context set operators are refined into *Box* operators. They take *Box* operands and return a *Box* as the result.

In the next chapter, we will use these concepts at the language level.

Chapter 3

Formal Syntax and Semantics of Lucx

In this chapter, first the syntax and semantics of Lucid will be reviewed; second, the extended syntax and semantics of Lucx will be given; third various evaluation rules will be given and illustrated; and finally we give a rigorous argument justifying Lucx as a conservation extension of Lucid.

3.1 Syntax and Semantics of Lucid

In this section, we review the syntax and semantics of Lucid as defined by Paquet [46]. Lucid includes function application, conditional expressions, intensional navigation and intensional query. The abstract syntax is shown in Figure 6.

The non-terminals E and Q respectively refer to *expressions* and *definitions*. This syntax assumes that identifiers (id) can refer to *constants*, *data operations*, *variables*, *functions* or *dimensions*. The *data operations* includes the *standard data operators* shown in Figure 7.

$$
\begin{array}{rcl}
E & ::= & id \\
& | & E(E_1, \ldots, E_n) \\
& | & \texttt{if } E \texttt{ then } E' \texttt{ else } E'' \\
& | & \#E \\
& | & E \texttt{ @ } E'\ E'' \\
& | & E \texttt{ where } Q
\end{array}
\qquad
\begin{array}{rcl}
Q & ::= & \texttt{dimension } id \\
& | & id = E \\
& | & id(id_1, \ldots, id_n) = E \\
& | & Q\ Q
\end{array}
$$

Figure 6: Abstract Syntax for Lucid [46]

The `iseod` operation is added to test for the presence of a special *end-of-data* value.

$$
\begin{array}{rcl}
\textit{convention-data-op} & ::= & \textit{unary-op} \\
& | & \textit{binary-op} \\
\textit{unary-op} & ::= & \texttt{! | - | iseod} \\
\textit{binary-op} & ::= & \textit{arith-op} \\
& | & \textit{rel-op} \\
& | & \textit{log-op} \\
\textit{arith-op} & ::= & \texttt{+ | - | * | / | \%} \\
\textit{rel-op} & ::= & \texttt{< | > | <= | >= | == | !=} \\
\textit{log-op} & ::= & \texttt{\&\& | ||}
\end{array}
$$

Figure 7: Standard data operations in Lucid

Paquet [46] has given a structural operational semantics of Lucid. Structural operational semantics style was introduced by Plotkin [49]. Both structural operational semantics and denotational semantics are among the different kinds of dynamic semantics discussed by Mosses [41]. Lucid, being a multidimensional language dealing with infinite streams, can only be implemented by interpreters, whose behaviors are more suitably described by dynamic semantics. Degano and Priami [20] have argued in favor of using operational semantics because it is *close to intuition, mathematically simple*, and allows *easy and early prototyping*. Motivated by these factors, we continue to use operational semantics to provide meaning for the new constructs in Lucx.

In an operational framework, the basic rule is in the format as follows: $\frac{Premises}{Conclusion}$, which means whenever the premises occur (interpreting them as possible computational steps),

then the conclusion will occur as well [20]. The operational semantics of Lucid is shown in Figure 8. In the semantic rules, the notation $\mathcal{D}, \mathcal{P} \vdash E : v$, means that in the definition environment $\mathcal{D}$, and in the evaluation context $\mathcal{P}$, expression E evaluates to a value v.

The definition environment $\mathcal{D}$ retains the definitions of all of the identifiers that appear in a Lucid program shown in Table 1. It is therefore a partial function

$\mathcal{D}$: **Id** $\rightarrow$ **IdEntry**

where **Id** is the set of all possible identifiers and **IdEntry** has five possible kinds of values. These are :

1. *Dimensions* define the coordinates in which one can navigate with the `#` and `@` operators. The **IdEntry** is $(\texttt{dim})$;

2. *Constants* are external entities that provide a single value, whatever the context is. The **IdEntry** is $(\texttt{const}, c)$, where c is the value of the constant;

3. *Data operators* are external entities that provide memoryless functions. The **IdEntry** is $(\texttt{op}, f)$, where f is the function itself;

4. *Variables* carry the multidimensional streams. The **IdEntry** is $(\texttt{var}, E)$, where E is the expression defining the variable;

5. *Functions* are non-recursive user-defined functions. The **IdEntry** is $(\texttt{func}, id_i, E)$, where the id_i are the formal parameters to the function and E is the body of the function.

$$\mathbf{E_{cid}}\ :\ \frac{\mathcal{D}(id) = (\texttt{const}, c) = v}{\mathcal{D}, \mathcal{P} \vdash id : v} \qquad \mathbf{E_{did}}\ :\ \frac{\mathcal{D}(id) = (\texttt{dim}) = v}{\mathcal{D}, \mathcal{P} \vdash id : v}$$

$$\mathbf{E_{opid}}\ :\ \frac{\mathcal{D}(id) = (\texttt{op}, f) = v}{\mathcal{D}, \mathcal{P} \vdash id : v} \qquad \mathbf{E_{fid}}\ :\ \frac{\mathcal{D}(id) = (\texttt{func}, id_i, E) = v}{\mathcal{D}, \mathcal{P} \vdash id : v}$$

$$\mathbf{E_{vid}}\ :\ \frac{\mathcal{D}(id) = (\texttt{var}, E) \qquad \mathcal{D}, \mathcal{P} \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash id : v}$$

$$\mathbf{E_{op}}\ :\ \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{op}, f) \qquad \mathcal{D}, \mathcal{P} \vdash E_i : v_i}{\mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : f(v_1, \ldots, v_n)}$$

$$\mathbf{E_{fct}}\ :\ \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{func}, id_i, E') \qquad \mathcal{D}, \mathcal{P} \vdash E'[id_i \leftarrow E_i] : v}{\mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : v}$$

$$\mathbf{E_{c_T}}\ :\ \frac{\mathcal{D}, \mathcal{P} \vdash E : true \qquad \mathcal{D}, \mathcal{P} \vdash E' : v'}{\mathcal{D}, \mathcal{P} \vdash \texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E'' : v'}$$

$$\mathbf{E_{c_F}}\ :\ \frac{\mathcal{D}, \mathcal{P} \vdash E : false \qquad \mathcal{D}, \mathcal{P} \vdash E'' : v''}{\mathcal{D}, \mathcal{P} \vdash \texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E'' : v''}$$

$$\mathbf{E_{tag}}\ :\ \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash \#E : \mathcal{P}(id)}$$

$$\mathbf{E_{at}}\ :\ \frac{\mathcal{D}, \mathcal{P} \vdash E' : id \qquad \mathcal{D}(id) = (\texttt{dim}) \qquad \mathcal{D}, \mathcal{P} \vdash E'' : v'' \qquad \mathcal{D}, \mathcal{P}\dagger[id \mapsto v''] \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E\ @E'\ E'' : v}$$

$$\mathbf{E_{w}}\ :\ \frac{\mathcal{D}, \mathcal{P} \vdash Q\ :\ \mathcal{D}', \mathcal{P}' \qquad \mathcal{D}', \mathcal{P}' \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E\ \texttt{where}\ Q : v}$$

$$\mathbf{Q_{dim}}\ :\ \frac{}{\mathcal{D}, \mathcal{P} \vdash \texttt{dimension}\ id\ :\ \mathcal{D}\dagger[id \mapsto (\texttt{dim})], \mathcal{P}\dagger[id \mapsto 0]}$$

$$\mathbf{Q_{id}}\ :\ \frac{}{\mathcal{D}, \mathcal{P} \vdash id = E\ :\ \mathcal{D}\dagger[id \mapsto (\texttt{var}, E)], \mathcal{P}}$$

$$\mathbf{Q_{fid}}\ :\ \frac{}{\mathcal{D}, \mathcal{P} \vdash id(id_1, \ldots, id_n) = E\ :\ \mathcal{D}\dagger[id \mapsto (\texttt{func}, id_i, E)], \mathcal{P}}$$

$$\mathbf{QQ}\ :\ \frac{\mathcal{D}, \mathcal{P} \vdash Q\ :\ \mathcal{D}', \mathcal{P}' \qquad \mathcal{D}', \mathcal{P}' \vdash Q' : \mathcal{D}'', \mathcal{P}''}{\mathcal{D}, \mathcal{P} \vdash Q\ Q'\ :\ \mathcal{D}'', \mathcal{P}''}$$

Figure 8: Operational Semantic Rules for Lucid [46]

Table 1: Possible identifiers in the definition environment

type	form
dimension	$(\texttt{dim})$
constant	$(\texttt{const}, c)$
operator	$(\texttt{op}, f)$
variable	$(\texttt{var}, E)$
function	$(\texttt{func}, id_i, E)$

The evaluation context $\mathcal{P}$ associates a tag to each relevant dimension. It is therefore a partial function: $\mathcal{P} : \mathbf{Id} \rightarrow \mathbb{N}$. Each type of identifier can only be used in the appropriate situations. Identifiers of type, `op`, `func` and `dim` respectively resolved by the $\mathbf{E_{opid}}$, $\mathbf{E_{fid}}$, and $\mathbf{E_{did}}$ rules, evaluate to their respective semantic records, as listed in Table 1. Constant identifiers (`const`), resolved by the $\mathbf{E_{cid}}$ rule, evaluate to the corresponding constant records, as listed in Table 1. Function calls, resolved by the $\mathbf{E_{fct}}$ rule, require the renaming of the formal parameters into the actual parameters (as represented by $E'[id_i \leftarrow E_i]$).

The rule for the navigation operator, $\mathbf{E_{at}}$, which corresponds to the syntactic expression E `@` E' E'', evaluates E in context E' E'', where E' evaluates to a dimension and E'' to a value corresponding to a tag in E'. The function $\mathcal{P}' = \mathcal{P}\dagger[id \mapsto v'']$ means that $\mathcal{P}'(x)$ is v'' if $x = id$, and $\mathcal{P}(x)$ otherwise. The rule for the `where` clause, $\mathbf{E_w}$, which corresponds to the syntactic expression E `where` Q, evaluates E using the definitions (Q).

The additions to the definition environment and context of evaluation made by the $\mathbf{Q}$ rules are local to the current `where` clause. This is represented by the fact that the $\mathbf{E_w}$ rule returns neither $\mathcal{D}$ nor $\mathcal{P}$. The $\mathbf{Q_{dim}}$ rule adds a dimension to the definition environment and, as a convention, adds this dimension to the context of evaluation with tag 0. The $\mathbf{Q_{id}}$ and $\mathbf{Q_{fid}}$ add variable and function identifiers along with their definition to the definition

environment.

The initial definition environment $\mathcal{D}_0$ includes the predefined intensional operators, the constants and the data operators, and $\mathcal{P}_0$ defines initial context of evaluation, Hence

$$\mathcal{D}_0, \mathcal{P}_0 \vdash E : v$$

represents the computation of any Lucid expression E, where v is the result.

3.2 Syntax and Semantics of Lucx

$$
\begin{array}{rcl}
E & ::= & id \\
& | & E(E_1, \ldots, E_n) \\
& | & \texttt{if } E \texttt{ then } E' \texttt{ else } E'' \\
& | & \# \\
& | & \mathbf{E} \texttt{ @ } \mathbf{E'} \\
& | & \langle \mathbf{E_1}, \ldots, \mathbf{E_n} \rangle \mathbf{id} \\
& | & \mathbf{select(E, E')} \\
& | & \{\mathbf{E_1}, \ldots, \mathbf{E_n}\} \\
& | & \mathbf{Box[id_1, \ldots, id_n \mid E]} \\
& | & \mathbf{[id_1 : E_1, \ldots, id_n : E_n]} \\
& | & E \texttt{ where } Q
\end{array}
\qquad
\begin{array}{rcl}
Q & ::= & \texttt{dimension } id \\
& | & id = E \\
& | & id(id_1, \ldots, id_n) = E \\
& | & Q\ Q
\end{array}
$$

Figure 9: Abstract syntax for Lucx

The syntax of Lucx is shown in Figure 9. The syntactic rules presented in bold are the proposed extensions to Lucid syntax. In order to achieve contexts as first class objects, the original syntax for the @ operator with the form $\mathbf{E}$ @ $\mathbf{E'E''}$ is changed to $\mathbf{E}$ @ $\mathbf{E'}$. According to the Lucid semantics, the construct $\mathbf{E'E''}$ evaluate to two separate semantic entities, not contexts. In contrast, the $\mathbf{E'}$ part of the new construct $\mathbf{E}$ @ $\mathbf{E'}$ semantically evaluates to a context, thus introducing contexts as first class objects. The syntactic construct

$[\mathbf{id_1} : \mathbf{E_1}, \ldots, \mathbf{id_n} : \mathbf{E_n}]$ introduces the syntax for explicit use of *s_contexts* in the language. The syntactic rule $\{\mathbf{E_1}, \ldots, \mathbf{E_n}\}$ is to introduce a set of contexts and $\mathbf{Box}[\mathbf{id_1}, \ldots, \mathbf{id_n} \mid \mathbf{E}]$ syntax introduces a *Box*. Hence the $\mathbf{E}'$ part of the new construct $\mathbf{E}$ @ $\mathbf{E}'$ can be a simple context, a set of simple contexts, or a *Box*. The syntax $\langle \mathbf{E_1}, \ldots, \mathbf{E_n} \rangle \mathbf{id}$ introduces a tuple $\langle E_1, \ldots, E_n \rangle$ whose dimension is indicated as *id*, and **select** $(\mathbf{E}, \mathbf{E}')$ is the syntactic rule for selecting elements from a tuple.

$$\mathbf{E}_{\#} \; : \; \frac{}{\mathcal{D}, \mathcal{P} \vdash \# : \mathcal{P}}$$

$$\mathbf{E}_{\mathbf{tuple}} \; : \; \frac{\mathcal{D}(id) = (\mathtt{dim}) \qquad \mathcal{P} \dagger [id \mapsto 0] \qquad \mathcal{D}, \mathcal{P} \vdash E_i : v_i}{\mathcal{D}, \mathcal{P} \vdash \langle E_1, \ldots, E_n \rangle id : v_1 \; \mathit{fby.id} \; v_2 \; \mathit{fby.id} \; \ldots \; v_n \; \mathit{fby.id} \; \mathtt{eod}}$$

$$\mathbf{E}_{\mathbf{select}} \; : \; \frac{\mathcal{D}, \mathcal{P} \vdash E : [d : v'] \;\; \mathcal{D}, \mathcal{P} \vdash E' : \langle E_1, \ldots, E_n \rangle d \;\; \mathcal{D}, \mathcal{P} \dagger [d \mapsto v'] \vdash \langle E_1, \ldots, E_n \rangle d : v}{\mathcal{D}, \mathcal{P} \vdash \mathbf{select}(E, E') : v}$$

$$\mathbf{E}_{\mathbf{at(c)}} \; : \; \frac{\mathcal{D}, \mathcal{P} \vdash E' : \mathcal{P}' \qquad \mathcal{D}, \mathcal{P} \dagger \mathcal{P}' \vdash E \; : \; v}{\mathcal{D}, \mathcal{P} \vdash E \; @ \; E' : v}$$

$$\mathbf{E}_{\mathbf{at(s)}} \; : \; \frac{\mathcal{D}, \mathcal{P} \vdash E' \; : \; \{\mathcal{P}_1, \ldots, \mathcal{P}_m\} \qquad \mathcal{D}, \mathcal{P} \dagger \mathcal{P}_{i:1..m} \vdash E \; : \; v_i}{\mathcal{D}, \mathcal{P} \vdash E \; @ \; E' : \{v_1, \ldots, v_m\}}$$

$$\mathbf{E}_{\mathbf{context}} \; : \; \frac{\mathcal{D}(id_j) = (\mathtt{dim}) \quad \mathcal{D}, \mathcal{P} \vdash E_j : v_j \quad \mathcal{P}' = [id_1 : v_1, \ldots, id_n : v_n]}{\mathcal{D}, \mathcal{P} \vdash [id_1 : E_1, \; \ldots, \; id_n : E_n] : \mathcal{P}'}$$

$$\mathbf{E}_{\mathbf{box}} \; : \; \frac{\mathcal{D}(id_1) = (\mathtt{dim}), \ldots, \mathcal{D}(id_n) = (\mathtt{dim})}{\mathcal{D}, \mathcal{P} \vdash \mathbf{Box}[id_1, \ldots, id_n \mid E] : Box[id_1, \ldots, id_n \mid \mathcal{D}, \mathcal{P} \dagger [id_1 \rightarrow id_1] \ldots \dagger [id_n \rightarrow id_n] \vdash E : true]}$$

$$\mathbf{E}_{\mathbf{set}} \; : \; \frac{\mathcal{D}, \mathcal{P} \vdash E_1 : \mathcal{P}_1, \quad \ldots, \quad \mathcal{D}, \mathcal{P} \vdash E_n : \mathcal{P}_n}{\mathcal{D}, \mathcal{P} \vdash \{E_1, \ldots, E_n\} : \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}}$$

$$\mathbf{E}_{\mathbf{cop}} \; : \; \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\mathtt{cop}, f) \qquad \mathcal{D}, \mathcal{P} \vdash C_i : v_i}{\mathcal{D}, \mathcal{P} \vdash E(C_1, \ldots, C_n) : f(v_1, \ldots, v_n)}$$

$$\mathbf{E}_{\mathbf{sop}} \; : \; \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\mathtt{sop}, f) \qquad \mathcal{D}, \mathcal{P} \vdash C_i : \{v_1^i, \ldots, v_{k_i}^i\}}{\mathcal{D}, \mathcal{P} \vdash E(C_1, \ldots, C_n) : f(\{v_1^1, \ldots, v_{k_1}^1\}), \ldots, \{(v_n^1, \ldots, v_{k_n}^n)\})}$$

Figure 10: Changed Semantic Rules for Lucx

Figure 10 shows the changes in the operational semantics of Lucid given in Figure 8. Semantically speaking, the symbol # is a nullary operator, which evaluates to the current evaluation context $\mathcal{P}$. The definition of $\mathcal{P}$ is changed as follows : $\mathcal{P} : id \rightarrow U$, where U is any enumerable tag set. The semantic rule $\mathbf{E_{tupid}}$ evaluates a tuple as a finite stream whose dimension is explicitly indicated as E in the corresponding syntax rule $\langle \mathbf{E_1}, \ldots, \mathbf{E_n} \rangle \mathbf{id}$. Accordingly, the semantic rule $\mathbf{E_{select}}$ picks up one element indexed by E from the tuple E'.

The evaluation rule for the navigation operator, $\mathbf{E_{at(c)}}$, which corresponds to the syntactic expression $\mathbf{E}$ @ $\mathbf{E'}$, evaluates $\mathbf{E}$ in context $\mathbf{E'}$. The evaluation rule for the set navigation operator $\mathbf{E_{at(s)}}$, which corresponds to the syntactic expression $\mathbf{E}$ @ $\mathbf{E'}$, evaluates $\mathbf{E}$ in a set of contexts $\mathbf{E'}$. Hence, the evaluation result should be a collection of results of evaluating $\mathbf{E}$ at each element of $\mathbf{E'}$.

The semantic rule $\mathbf{E_{context}}$ evaluates $[\mathbf{id_1} : \mathbf{E_1}, \ldots, \mathbf{id_n} : \mathbf{E_n}]$ to a context, as defined in Section 2.1 (page 17). The semantic rule $\mathbf{E_{set}}$ evaluates $\{\mathbf{E_1}, \ldots, \mathbf{E_m}\}$ as a set of contexts. And the semantic rule $\mathbf{E_{box}}$ evaluates a Lucx **Box** to the mathematical construct Box set defined in Chapter 2 (Definition 21, page 32).

In [46], Paquet defined 5 types of identifiers listed as follows: dimension, constant, operator, variable, and function. In practice, these entries are put into dictionary, i.e., definition environment $\mathcal{D}$. Similarly, two more types of identifiers should be added into this table. That is, *context operator* and *context set operator*. The table is shown below:

Consequently, the semantic rule E_{cop}(Figure 10, Page 44) is added to evaluate *context operator* and the semantic rule E_{sop}(Figure 10, Page 44) is added to evaluate *context set operator*. We use Lucx syntax for context and *Box* in examples and programs throughout

Table 2: Possible identifiers in the definition environment

type	form
dimension	$(\texttt{dim})$
constant	$(\texttt{const}, c)$
operator	$(\texttt{op}, f)$
variable	$(\texttt{var}, E)$
function	$(\texttt{func}, id_i, E)$
context operator	$(\texttt{cop}, f)$
context set operator	$(\texttt{sop}, f)$

the rest of the thesis.

3.3 Lucx - A Conservative Extension of Lucid

In mathematical logic [24], a logical theory, T_2, is a conservative extension of theory, T_1, if any consequence of T_2, involving symbols of T_1 only, is already a consequence of T_1. Informally, the new theory may possibly be more convenient for proving theorems, but it proves no new theorems about the old theory. The importance of this notion lies in the theorem: *If T_2 is a conservative extension of T_1, and T_1 is consistent, then T_2 is consistent as well.*

Hence, conservative extensions do not bear the risk of introducing new inconsistencies. This can also be seen as a methodology for writing and structuring large theories: start with a theory, T_0, that is known (or assumed) to be consistent, and successively build conservative extensions T_1, T_2, ... of it.

The above notion has been applied to Larch [28] and algebraic specification languages to construct a hierarchy of specifications. Each new level in the specification hierarchy preserves any consequence of the previous level(s), and may contain a function symbol

that does not occur in the specifications at the lower levels. Thus, conservative extensions can be used to define properties of new symbols, but do not introduce inconsistencies or additional properties of existing symbols. Based on this concept we argue why Lucx should be considered as a conservative extension of Lucx.

Lucx is obtained from Lucid by making three kinds of extensions : (1) extensions done on the definition environment (Table 2, Chapter 3), (2) the syntactic extensions (Figure 9, Chapter 3), and (3) the semantic extensions (Figure 10, Chapter 3). We argue below that each one of these extensions is conservative. The introduction of context as a first class object in Lucx allows one to write $A = [d : 1]$ in Lucx; this cannot be done in Lucid. That is, there is some functionality that is new in Lucx.

Extension in the Definition Environment Table 2 (Chapter 3) shows the set ID' of identifiers and the set $IdEntry'$ of possible values in the definition environment $\mathcal{D}'$ of Lucx. From this, we can infer that $ID' = ID \cup \{$**context operator, context set operator**$\}$, and

$$\forall\, id \in ID \bullet \mathcal{D}'(id) = \mathcal{D}(id)$$

The definitions of new identifiers do not interfere with the identifiers in the definition environment of Lucid. To be convinced, we refer to the semantic rules $\mathbf{E_{cop}}$, and $\mathbf{E_{sop}}$ given in Figure 10: they do not interfere with the semantic rules $\mathbf{E_{did}}, \mathbf{E_{cid}}, \mathbf{E_{opid}}, \mathbf{E_{vid}}, \mathbf{E_{fid}}$. Hence the function $\mathcal{D}'$ is a consistent extension of the function $\mathcal{D}$.

Syntactic Extensions The evaluation syntax $E@E'E''$ in Lucid is *changed* to $E@E'$ in Lucx. This is only a syntactic sugaring in the sense that

$$E@[d : 1] = E@d\ 1$$

$$E@[d_1 : 1, d_2 : 3] = E@d_1\ 1@d_2\ 3$$

The introduction of context as a first class object in Lucx only simplifies the Lucid syntax. The semantic rule $\mathbf{E_{at}}$ (Figure 8) is rewritten as $\mathbf{E_{at(c)}}$ (Figure 10) as shown below:

$$\mathbf{E_{at}} \ : \ \frac{\mathcal{D}, \mathcal{P} \vdash E' : id \qquad \mathcal{D}(id) = (\mathtt{dim}) \qquad \mathcal{D}, \mathcal{P} \vdash E'' : v'' \qquad \mathcal{D}, \mathcal{P} \dagger [id \mapsto v''] \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E\ @E'\ E'' : v}$$

$$\mathbf{E_{at(c)}} \ : \ \frac{\mathcal{D}, \mathcal{P} \vdash \mathbf{E}' : \mathcal{P}' \qquad \mathcal{D}, \mathcal{P} \dagger \mathcal{P}' \vdash \mathbf{E} \ : \ v}{\mathcal{D}, \mathcal{P} \vdash \mathbf{E} \ @ \ \mathbf{E}' : v}$$

Notice that $\mathbf{E_{at(c)}}$ and $\mathbf{E_{at}}$ have the same meaning. With this change in the syntax of the semantic rule and by the default convention (syntactic sugaring) explained above, the rule $\mathbf{E_{at(c)}}$ (Figure 10) in Lucx reduces to the Lucid rule $\mathbf{E_{at}}$ when c is a micro-context.

The following five syntactic rules in Lucx are new:

$$\begin{array}{rcl@{\qquad}rcl} E & ::= & \langle \mathbf{E_1}, \ldots, \mathbf{E_n} \rangle \mathbf{id} & E & ::= & \{\mathbf{E_1}, \ldots, \mathbf{E_n}\} \\ & | & \mathbf{select}(\mathbf{E}, \mathbf{E}') & & | & \mathbf{Box}[\mathbf{id_1}, \ldots, \mathbf{id_n} \mid \mathbf{E}] \\ & & & & | & [\mathbf{id_1} : \mathbf{E_1}, \ldots, \mathbf{id_n} : \mathbf{E_n}] \end{array}$$

They do not exist in Lucid. Consequently, the syntactic extensions are conservative.

Semantic Extensions We have already explained that the rule $\mathbf{E_{at(c)}}$ (Figure 10) in Lucx reduces to the Lucid rule $\mathbf{E_{at}}$ (Figure 8) when c is a micro-context. The semantics corresponding to the new syntax rules (Figure 10) do not exist in Lucid and consequently do not interfere with any of the rules in Lucid (by inspection). The rest of Lucx rules are identical with Lucid semantic rules. Consequently we can claim that the semantic extension is conservative.

Based on the above arguments we can claim, without offering a formal proof, that a Lucid program executed (interpreted) in Lucx execution environment will have the same behavior as if it was executed in Lucid execution environment. In this sense Lucx is a conservative extension of Lucid.

3.4 Summary

In this chapter, the formal syntax and semantics of Lucx are given. In the previous version of Lucid, namely Indexical Lucid, by making dimension as first class objects, evaluation of @ expression can be arbitrarily done along the dimension that is defined. Similarly, in Lucx, by making context as first class objects, evaluation of @ expression can be arbitrarily done in different contexts. Moreover, being a first class object, context exists independently in the language. That is, one context may be used to evaluate different @ expressions, at the same time an @ expression can also be evaluated at different contexts. This justifies and finally realizes the Intensional logic as providing the logical base of Lucid language. That is, the intension of an @ expression E in Intensional logic, corresponding to the definition of the @ expression E in Lucx, is separated from the extension of the @ expression E in Intensional logic, corresponding to the evaluation of the @ expression E in Lucx. We can claim that Lucx faithfully follows the Intensional logic and therefore is a genuine Intensional language. This feature also distinguishes the language Lucx from other imperative languages or functional languages, where index (for imperative languages) or evaluation environment (for functional language) are always bound to statements or expressions.

Chapter 4

Discussions on Issues Related To Evaluation

In this chapter, we discuss some issues related to evaluation in Lucx. These discussions, raised at an early stage of Lucx development, are meant to point out some interesting features of Lucx.

4.1 Evaluation of @ Expressions

4.1.1 Rules for @ Expressions with Atomic Values

We define evaluation rules for context expressions, context set expressions, and *Box* expressions.

Evaluation at Context Expressions

The evaluation of $E@C$ at a context expression C may be done in the following way : First evaluate the context expression C, then evaluate the expression $E@C$.

In the following discussion, general evaluation rules will be listed. In the following rules E is an expression, C is a context expression, and c_1 and c_2 are context variables. The semantic rule for $E_{at(c)}$ given in Figure 10 (Page 44) can be applied to an evaluation rule below and recursively expanded with other semantic rules to get the full semantic interpretation for that rule.

1. C is a micro context. $E \ @ \ C$, where $C = [E' : E'']$, is evaluated using the semantics $\mathbf{E_{at(c)}}$ along with $\mathbf{E_{context}}$ given in Figure 10 (Page 44).

2. C is a simple context. $E \ @ \ C$ is evaluated using the semantics $\mathbf{E_{at(c)}}$ along with $\mathbf{E_{context}}$ given in Figure 10 (Page 44).

3. C is a non-simple context. $E \ @ \ C = \{E \ @ \ s \mid s \in \mathcal{S}(C)\}$, $\mathcal{S}(C)$ is a set of simple contexts constructed from a non-simple context C.

4. $C = c_1 \mid c_2$,

$$E@C = \begin{cases} E@c_1, & \text{if } c_1 = c_1 \mid c_2 \\ E@c_2, & \text{if } c_2 = c_1 \mid c_2 \end{cases}$$

5. $C = c_1 \cap c_2$.

   - c_1 and c_2 are both micro contexts, $c_1 \neq c_2$: the expression $E \ @ \ C$ evaluates to E.

- c_1 is a micro context and c_2 is not a micro context:

$$E@C = \begin{cases} E@c_1, & \text{if } c_1 \in c_2 \\ E, & \text{otherwise} \end{cases}$$

- c_2 is a micro context and c_1 is not a micro context:

$$E@C = \begin{cases} E@c_2, & \text{if } c_2 \in c_1 \\ E, & \text{otherwise} \end{cases}$$

- c_1 and c_2 are both simple contexts: the intersection of two simple contexts is either a simple context or NULL.

$$E@C = \begin{cases} E@(c_1 \cap c_2), & c_1 \cap c_2 \neq NULL \\ E, & \text{otherwise} \end{cases}$$

- Either c_1 or c_2 is a non_simple context:

$$E@C = \begin{cases} \{E@c \mid c \in \{c_1 \cap c' \mid c' \in S(c_2)\}\}, & c_1 \text{ is simple} \\ \{E@c \mid c \in \{c_2 \cap c' \mid c' \in S(c_1)\}\}, & c_2 \text{ is simple} \end{cases}$$

- Both c_1 and c_2 are non_simple contexts. The expression $E@C$ evaluates to $\{E@c \mid c \in S(c_1) \cap S(c_2)\}$.

6. $C = c_1 \cup c_2$: Either C is a simple context or a non-simple context. The rule 2 or the rule 3 applies to evaluate $E@C$.

7. $C = c_1 \oplus c_2$: According to the set theory and the Definition 4(Page 18), the expression $E@C$ evaluates to $E@c_2 \cup c_1 \uparrow (dim(c_1) \cap dim(c_2))$. Once the result of $c_1 \uparrow (dim(c_1) \cap dim(c_2))$ is computed, the rule 6 applies to E.

8. $C = c_1 \ominus c_2$: The context expression is evaluated using the definition of the operator $\ominus$ and the expression $E@C$ is evaluated.

9. $C = c_1 \rightleftharpoons c_2$: The context expression is evaluated using the definition of the operator $\rightleftharpoons$ and the expression $E@C$ is evaluated. The expression $E@C$ is the set $\{E@c \mid c \in S(c_1, c_2)\}$, where $S(c_1, c_2)$ is the set of simple contexts of $c_1 \rightleftharpoons c_2$.

10. $C = c_1 \rightharpoonup c_2$: The context expression is evaluated using the definition of the operator $\rightharpoonup$ and the expression E is evaluated. The expression $E@C$ is the set $\{E@c \mid c \in S(c_1, c_2)\}$, where $S(c_1, c_2)$ is the set of simple contexts of $c_1 \rightharpoonup c_2$.

Evaluation of Expressions at Context Set Expressions

The evaluation rule for $E@s$, where $s = \{c_1, \ldots, c_m\}$, is:

$$E@s = E@\{c_1, \ldots, c_m\} = \{E@c_1, \ldots, E@c_m\},$$

where $E@c_k, 1 \leq k \leq m$, will be evaluated following the evaluation rules listed in the previous section. For any other context set expression s defined in Figure 4 (Page 32), we compute s using the rules for context set operators, and then evaluate the expression E on every context expression in the result. As an example, if s_1 and s_2 are sets of contexts

$$E@s_1 \oplus s_2 = \{E@c_i \oplus c_j \mid c_i \in s_1, c_j \in s_2\}$$

Evaluation rules at *Box*es The *Box* operators join($\boxtimes$), intersection($\boxdot$), and union($\boxplus$) when applied to *Box* operands give *Box* as result. Hence the evaluation rules for set of contexts apply to *Box* expressions. Let b_1, b_2 be two *Box*es, $\Delta(b_1) = \Delta_1$, $P(b_1) = p_1$, $\Delta(b_2) = \Delta_2$, and $P(b_2) = p_2$.

1. [join rule J:]

   $E@b_1 \boxtimes b_2 = \{E@c_i \cup c_j \mid c_i \in b_1, c_j \in b_2, c_i \downarrow \Delta = c_j \downarrow \Delta\}$

   $= \{E@c_i@c_j \mid c_i \in b_1, c_j \in b_2\}$

   The equality follows the Rule 6.2 introduced in the next section 4.1.1(Page 54).

2. [intersection rule I:]

   $E@b_1 \boxdot b_2 = \{E@c \mid c \in (b_1 \boxtimes b_2) \downarrow \Delta\}$

3. [union rule U:]

   $E@b_1 \boxplus b_2 = \{E@c \mid c \in X_1\} \cup \{E@c \mid c \in X_2\}$,

   where $X_1 = \{s_i \cup s_j \uparrow \Delta \mid s_i \in b_1 \wedge s_j \in b_2\}$ and $X_2 = \{s_j \cup s_i \uparrow \Delta \mid s_i \in b_1 \wedge s_j \in b_2\}$.

Discussion on evaluation rules of special case of $\cup$

Rules for some special cases are stated below. They can be formally verified by invoking the semantics of context operators and set theory results. We omit their proofs.

- [6.1] c_1 and c_2 are simple contexts, and $dim(c_1) \cap dim(c_2) = \varnothing$. That is, $c_1 \cup c_2$ is a simple context. $E@C = (E@c_1)@c_2 = (E@c_2)@c_1$

- [6.2] c_1 and c_2 are simple contexts, $dim(c_1) \neq dim(c_2)$, $\Delta = dim(c_1) \cap dim(c_2) \neq \varnothing$, and $c_1 \downarrow \Delta = c_2 \downarrow \Delta$. That is, $c_1 \cup c_2$ is a simple context. $E@C = (E@c_1)@c_2 = (E@c_2)@c_1$

- [6.3] c_1 and c_2 are simple contexts, and $dim(c_1) = dim(c_2)$. Then $c_1 \cup c_2$ is not a simple context.

case 1 : In general, the evaluation of E at $c = c_1 \cup c_2$ is given by

$E@c = E@S(c) = \{E@c' \mid c' \in S(c)\}$ ∎

The proof follows from the definition of $S(c)$ and the semantics of @.

case 2 : Let c_1 and c_2 be simple contexts satisfying the conditions

- $\Delta = dim(c_1) \cap dim(c_2) \neq dim(c_1 \cap c_2) \neq \varnothing$. That is, $c_1 \downarrow \Delta \neq c_2 \downarrow \Delta$
- $c_4 = c_1 \uparrow \Delta \cup c_2 \uparrow \Delta$ is a simple context.

Then the evaluation of expression E at c is given by

$E@c = E@S(c_3)@c_4 = E@c_4@S(c_3)$,

where $c_3 = c_1 \downarrow \Delta \cup c_2 \downarrow \Delta$ is a non-simple context and its intersection with c_4 is NULL ∎.

- [6.4] Let c_1 and c_2 be any two contexts satisfying the conditions
  - $\Delta = dim(c_1) \cap dim(c_2) \neq dim(c_1 \cap c_2) \neq \varnothing$. That is, $c_1 \downarrow \Delta \neq c_2 \downarrow \Delta$
  - $c_4 = c_1 \uparrow \Delta \cup c_2 \uparrow \Delta$ is a non-simple context.

  Then the evaluation of expression E at c is given by

  $$E@S(c) = E@S(c_3)@S(c_4) = E@S(c_4)@S(c_3)$$

  where $c_3 = c_1 \downarrow \Delta \cup c_2 \downarrow \Delta$ is a non-simple context and its intersection with c_4 is NULL ∎.

We introduce the symbol $\bigotimes$ to sum up the evaluation rules for union for contexts:

Theorem 1 *If c_1 and c_2 are contexts then for any expression E the evaluation of $E@c$ satisfies the equation*

$$E@c_1 \cup c_2 = E@c_1 \bigotimes E@c_2 \;=\; \{E@c \mid c \in S(c_1 \cup c_2)\},$$

where

$$E@c_1 \bigotimes E@c_2 = \begin{cases} E@c_1@c_2, & \text{if } c_1 \cup c_2 \text{ is a simple context} \quad \text{[Rule 6.1, 6.2]} \\ E@S(c), & \text{if } c_1 \text{ and } c_2 \text{ are simple contexts} \\ & \text{and } dim(c_1) = dim(c_2) \quad \text{[Rule 6.3, case 1]} \\ E@S(c_3)@c_4, & \text{if } c_4 = c_1 \uparrow \Delta \cup c_2 \uparrow \Delta \text{ is a simple context and} \\ & c_3 = c_1 \downarrow \Delta \cup c_2 \downarrow \Delta \text{ is a non-simple context} \\ & \text{[Rule 6.3, case 2]} \\ E@S(c_3)@S(c_4), & \text{if } c_4 = c_1 \uparrow \Delta \cup c_2 \uparrow \Delta \text{ is a non-simple context} \\ & \text{and } c_3 = c_1 \downarrow \Delta \cup c_2 \downarrow \Delta \text{ is a non-simple context} \\ & \text{[Rule 6.4]} \quad \blacksquare \end{cases}$$

4.1.2 Rules for @ Expressions with Tuples as Values

When E is a tuple stream the join rule has some analogy to natural join operation in relational algebra, which we shall explore.

Tuples and Intensional Relations A tuple is a finite stream. The size of a tuple is the number of components it has. Although the syntax of a tuple in Lucx is $\langle E_1, \ldots, E_n \rangle E$, in the subsequent discussions, where the dimension name is not quite relevant, we omit the reference to the dimension in tuple representation. If one or more component of a tuple is not a constant we call it a tuple expression. The evaluation $E@c$ may result in a tuple

expression. We call a stream of tupe expressions as an *relational expression*, which we call as *Intensional Relations*(IR). Evaluation of the relation *IR* at certain contexts may be a stream of tuples. As an example, the stream $\langle 1,7,7\rangle, \langle 3,0,0\rangle, \langle 4,12,9\rangle$ is the result of evaluating the IR $\langle 1,5+x,2+y+x\rangle, \langle 3,2-x,1-y+x\rangle, \langle 4,3+x^2,1+y+x^2\rangle$ at certain contexts.

There is some interplay between the *Box* relational operators and operators $\Cup$ (union), Π (projection on components), and $\bowtie$ (natural join) in relational algebra. Let us consider the *join rule* for *Box*es (Section 4.1.1, Page 53). A naive calculation using the formula given in the previous section will require the computation of all possible unions and projection calculation for every pair of contexts chosen from the *Box*es. Our goal is to reduce this complexity by obtaining the extensions of the IRs $E@b_1$ and $E@b_2$ that will converge to the IR $E \mathbin{@} b_1 \boxtimes b_2$. This is quite analogous to the natural join calculation in relational algebra, where tuples from each relation are extended minimally while agreeing on certain attribute names. Based on this observation we calculate $E@b_1 \boxtimes b_2$ as follows:

- calculate the intensional relation $R_1 = E@b_1$, which includes the evaluations of E for every $c_i \in b_1$
- evaluate R_1 at $\overline{c_j} = c_j \uparrow \Delta_3$, where $c_j \in b_2$, and $c_j \downarrow \Delta_3 = c_i \downarrow \Delta_3$.
- calculate the intensional relation $R_2 = E@b_2$ which includes the evaluations of E for every $c_j \in b_2$
- evaluate R_2 at $\overline{c_i} = c_i \uparrow \Delta_3$, where $c_i \in b_1$, and $c_j \downarrow \Delta_3 = c_i \downarrow \Delta_3$.
- the intersection of these two extensions is a relation (if fully evaluated) or an IR; in

the latter case the variables in the resulting IR cannot be evaluated by any context in b_1 or in b_2;

Based on the above argument, we write the join rule (J) as the following Theorem.

Theorem 2

$$E@b_1 \boxtimes b_2$$

$$= \{(E@b_1)@\overline{c_j} \mid c_j \in b_2 \wedge c_j \downarrow \Delta = c_i \downarrow \Delta\} \cap \{(E@b_2)@\overline{c_i} \mid c_i \in b_1 \wedge c_j \downarrow \Delta = c_i \downarrow \Delta\}$$

We use a short hand notation, that is consistent with the relational algebra analogy:

$$E@b_1 \boxtimes b_2 = E@b_1 \bowtie E@b_2 \quad \blacksquare$$

According to this rule, evaluation at the join of *Box*es can be executed in parallel. Hence the efficiency of evaluation can be improved. This rule will be used later in Chapter 8 to improve the efficiency of constraint problem solving. The other two rules for *Box*es are related to relational algebra operations as given below:

- [intersection rule I:] $E \mathbin{@} b_1 \boxdot b_2 = \Pi_\Delta E \mathbin{@} b_1 \Cap \Pi_\Delta E \mathbin{@} b_2$, where Π_Δ is the projection operator in relational algebra, and $\Cap$ is the intersection operator in relational algebra.

- [union rule U:] $E \mathbin{@} b_1 \boxplus b_2 = E \mathbin{@} b_1 \Cup E \mathbin{@} b_2$, where $\Cup$ is the union operator for relations.

4.2 Context-Dependent Expression

Contexts are first class objects in Lucx, and hence should be usable as function parameters. Contexts can also be used as switches (as in imperative programming). A switch selects

one function definition from a list of alternatives. Depending on the actual context value assumed by the context parameter, one out of several function definitions will be chosen for activation.

An intensional expression E is a *context-dependent expression* because it is defined differently in different contexts. As an example, let b_1 and b_2 be two boxes, and E be an expression defined as E_1 for all contexts in $b_1 \ominus b_2$, defined as E_2 for all contexts in $b_2 \ominus b_1$, and E_3 in $b_1 \boxdot b_2$. Then we use the symbol λ and μ to respectively bind the expression and context : $\lambda \cdot E = (E_1, E_2, E_3), \mu \cdot E = (b_1 \ominus b_2, b_2 \ominus b_1, b_1 \boxdot b_2)$. The corresponding components of the tuples $\lambda \cdot E$ and $\mu \cdot E$ are bound. For a given context c, the evaluation of expression E is done as follows:

$$E@c = \begin{cases} E_1@c, & c \in b_1 \wedge c \notin b_2 \\ E_2@c, & c \in b_2 \wedge c \notin b_1 \\ E_3@c, & c \in b_1 \boxdot b_2 \end{cases}$$

Contexts may be used as parameters in a function definition. Let $f : X \times Y \times Z \times C \rightarrow W$, where C is a set of contexts; and $f(x, y, z, c), x \in X, y \in Y, z \in Z, c \in C$, be defined such that for different context values, the function's definitions are different. Let $b_1 = Box[D, E, F \mid D^2 + E^2 \leq \frac{F^2}{4} \wedge 0 \leq F \leq 4], b_2 = Box[D, E, F \mid D^2 + E^2 + F^2 \leq 9 \wedge F \geq 0]$, and $\mu \cdot f = \{b_1 \ominus b_2, b_2 \ominus b_1, b_1 \boxdot b_2\}$, and $\lambda \cdot f = \{2x^3 + y - 6, x + y^2, x^3 + y\}$. The definition of the function $f(x, y, z, c)$ is shown as a Lucx program in Example 26.

Example 26 :

```
f(x, y, z, c)
        = if c ∈ b1 ⊡ b2, then z^3 + y
```

$$\texttt{else if } (c \in b_1 \&\& c \notin b_2), \texttt{ then } 2x^3 + y - 6$$

$$\texttt{else } x + y^2$$

where

$$b_1 = \texttt{Box}[D, E, F \mid D^2 + E^2 \leq \tfrac{F^2}{4} \wedge 0 \leq F \leq 4]$$

$$b_2 = \texttt{Box}[D, E, F \mid D^2 + E^2 + F^2 \leq 9 \wedge F \geq 0]$$

end

A special case of context-dependent function is the function that produces a context as the result. Based upon the context, we can activate a function. In turn, such an activation can produce a new context. By repeating this two-step process we can program control systems with feedback loops. They are also fundamental in the development of self-adaptive context-aware systems [62].

4.3 Summary

In this chapter we discussed some issues related to evaluation in Lucx. We provided rules for evaluation of @ expressions and defined context-dependent functions. A more formal treatment of issues related to evaluation rules are left for future work. A context dependent function can have a context as a result. Thus, in a program contexts can play the role of switches, enabling the dynamic selection and execution of different functions. This feature is particularly necessary for control systems with feedback loops.

Chapter 5

Implementing Lucx in GIPSY

The GIPSY, an Intensional Programming investigation platform under development, allows the automated generation of compiler components for the different variants of the Lucid family of languages. It provides an execution environment allowing for these programs to be executed in multi-threaded or distributed mode [44]. We discuss the translation rules for Lucx operators into Lucid and provide an architecture for implementing Lucx programs in the GIPSY. The GIPSY architecture is shown in Figure 11.

5.1 GIPSY Architecture

The Generic Intensional Programming Language (GIPL), consisting only of @ and # operators, is one of the members of the Lucid family of languages. As its name implies, it is a generic language, i.e. all other languages of the family can be translated into it. All the other members of the Lucid family of languages, such as Indexical Lucid and Lucx, are

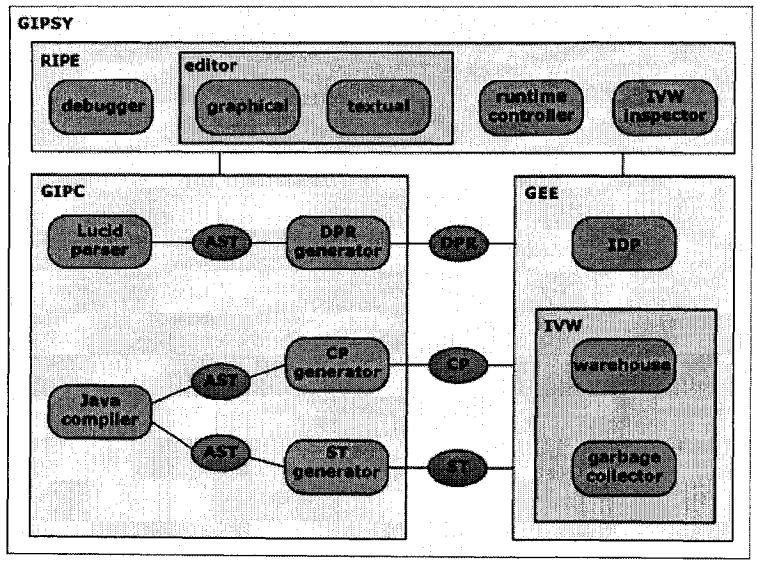

Figure 11: GISPY Architecture

called Specific Intensional Programming Languages (SIPL).

Using concrete language specifications, the GIPSY allows for the automated generation of compiler components for all SIPLs. These specifications take the form of (1) specification of the syntax of the language and (2) specification of the translation rules translating the syntactic constructs into the generic GIPL constructs. For example, in [46], translation rules are provided to translate Indexical Lucid operators such as *first, next, fby* into GIPL.

One of the main precepts of the GIPSY is that the General Execution Engine (GEE) can execute programs written in any IPL that is part of the Lucid family of languages. This is achievable through the generic nature of the GIPL, i.e. all SIPLs are translated into the GIPL, and the GEE then executes the generic version of the program. For each SIPL, the GIPC (General Intensional Programming Compiler) generates and uses a specific parser (SIPL parser), which is automatically generated using the *JavaCC* tool, with a grammatical

definition provided by the user. This parser generates an abstract syntax tree (SIPL AST), which has to be translated into its generic counterpart using an AST translator (SIPL-GIPL AST translator). That translator is also automatically generated by the GIPC using the translation rules of this SIPL as input. The GIPC architecture is shown in Figure 12.

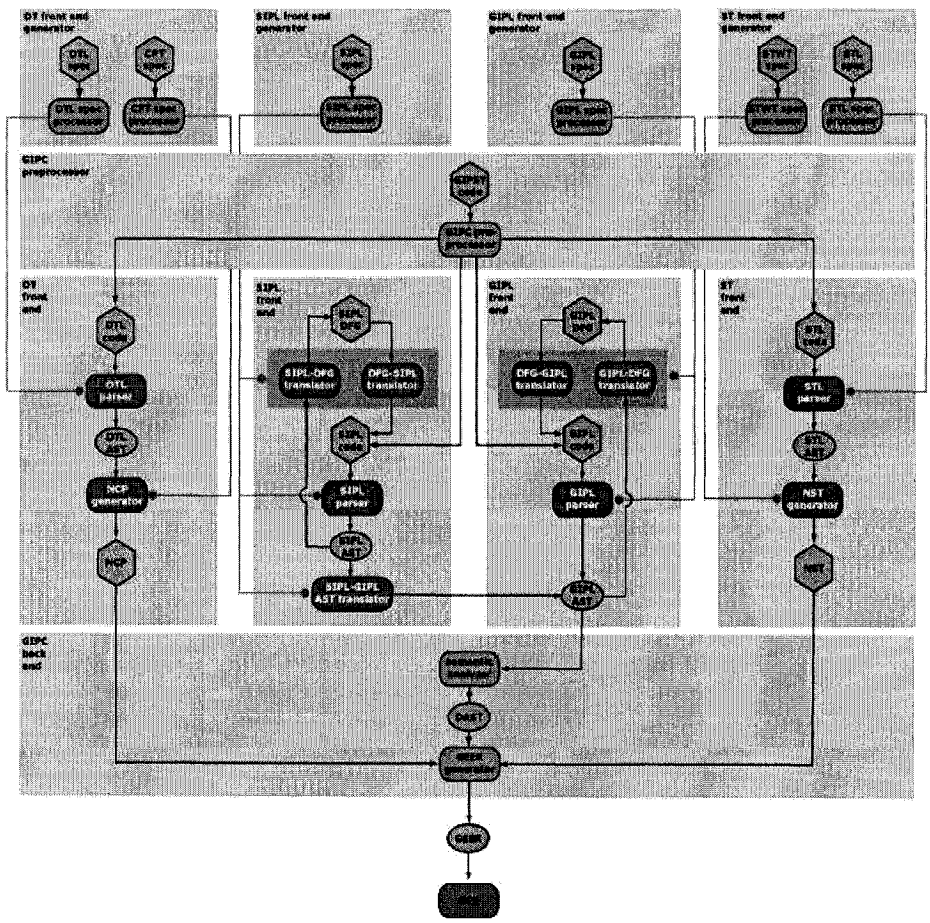

Figure 12: GIPC Architecture

Currently, the compiler for Indexical Lucid has been implemented successfully in the GIPSY. The parser for Indexical Lucid is generated using *JavaCC*, and the SIPL-GIPL AST translator uses the translation rules translating Indexical Lucid operators into GIPL.

According to the design of the GIPSY, a front end that provides the SIPL parser and SIPL-GIPL AST translator is required to implement the compiler for a new SIPL.

5.2 Implementing Lucx in GIPSY

Lucx is a conservative extension of Lucid and is thus a SIPL. Lucx parser and Lucx-GIPL AST translator as a Lucx front end to GIPC are to be provided in the GIPSY. Lucx parser can also be automatically generated using the *JavaCC* tool. In this section, we provide the translation rules for translating Lucx operators into Indexical Lucid operators. Combined with the translation rules for Indexical Lucid operators provided in [46], we will achieve a two-pass Lucx-GIPL AST translator. Once these two modules are integrated into GIPSY, the programs written in Lucx can be compiled and run in GIPSY.

5.2.1 Translation Rules for Context Operators

Primitive Functions The following primitive functions should be hard coded, and hence will be written in Java and incorporated into the GEE.

- $\underline{dim_m(m)}$ returns the dimension of a micro context m(Definition 3, Page 17).
- $\underline{tag_m(m)}$ returns the tag of a micro context m(Definition 3, Page 17).
- $\underline{dim(s)}$ returns the set of dimensions of a context s(Definition 3, Page 17).
- $\underline{tag(s)}$ returns the set of tags of a context s(Definition 3, Page 17).
- $\underline{construct(d,t)}$ constructs a micro context given dimension d and tag t.

- $\underline{construct_s(c_1, \ldots, c_m)}$ constructs a set of contexts $\{c_1, \ldots, c_m\}$ given simple contexts $c_1, \ldots, c_m$.

- $\underline{construct_d(d_1, \ldots, d_m)}$ constructs a dimension set $\{d_1, \ldots, d_m\}$ given dimensions $d_1, \ldots, d_m$.

- $\underline{add(c, S)}$ adds the context c into the set S if c doesn't exist in S.

- $\underline{IsEmpty(s)}$ checks whether the set s is an empty set.

Assumptions The assumptions we make are as follows:

- set operators, such as $\cap, \in$ used below, do not exist in Lucid; however for simplicity, these operators are assumed to be hard coded in Java and incorporated into the GEE as well. The symbol $\varnothing$ denotes the empty set.

- Data operations such as &&, ==, ! = are defined at atomic level and hence are implementable in Java. The *iseod* operation has been added in [46] to test for the presence of a special $end - of - data(eod)$ value. Hence we use the same notation.

- In Lucx a simple context is represented by a finite stream of micro contexts in it, with *eod* automatically added to the end of this stream. Representing a set by a sequence does not cause any loss of generality, because all the functions that process the stream are immune to the ordering in it. We also assume that all the functions below will terminate automatically when they meet the symbol *eod*.

Elementary Constructs The following Lucx constructs, which are heavily used in a program, can be derived from Indexical Lucid. Many of these constructs are defined as recursive functions. For those constructs, whose counterparts are formally defined in the previous chapters, we attach the corresponding references.

- *IsMember*(m, E) function examines whether or not a micro context m is included in a simple context E.

  $\texttt{IsMember(m,E)} = \texttt{if}(\texttt{dim}_{\texttt{m}}(\texttt{m}) == \texttt{dim}_{\texttt{m}}(\texttt{first E}) \&\& \texttt{tag}_{\texttt{m}}(\texttt{m}) == \texttt{tag}_{\texttt{m}}(\texttt{first E}))$

  $\texttt{then true else if}(\texttt{iseod(E)})\ \texttt{then false}$

  $\texttt{else IsMember}(\texttt{m}, \texttt{next E})$

- *IsPart*$(e,\ s)$ decides whether or not the dimension e is a member of the dimension set s.

  $\texttt{IsPart(e, s)} = \texttt{if}(\texttt{e} == \texttt{first s})\ \texttt{then true}$

  $\texttt{else if}(\texttt{iseod(s)})\ \texttt{then false else IsPart}(\texttt{e}, \texttt{next s})$

- *InterEmpty*$(s,\ s')$ decides whether the intersection of dimension sets s and s' is an empty set or not.

  $\texttt{InterEmpty(s, s')} = \texttt{if}(\texttt{IsPart}(\texttt{first s}, \texttt{s}'))\ \texttt{then false}$

  $\texttt{else if}(\texttt{iseod(s)})\ \texttt{then true else IsEmpty}(\texttt{next s}, \texttt{s}')$

- *extract*$(E\ ,\ E')$ generates a context whose elements are part of E, but the dimensions of those elements are not included in $dim(E')$.

  $\texttt{extract}(\texttt{E}, \texttt{E}') = \texttt{if}(\texttt{IsPart}(\texttt{dim}_{\texttt{m}}(\texttt{first E}), \texttt{dim}(\texttt{E}')) == \texttt{false})$

then first E fby extract(next E, E′)

else extract(next E, E′)

- *Override*$_c(E, E')$ (Definition 4, Page 18) generates a new context, which is the conflict-free union of E and E'.

  Override$_c$(E, E′) = E′ fby extract(E, E′)

- *Difference*$_c(E,\ E')$ returns the difference set of E and E'.

  Difference$_c$(E, E′) = if(IsMember(first E, E′) == true)

  then Difference$_c$(next E, E′)

  else first E fby Difference$_c$(next E, E′)

- *Projection*$_c(E, D)$ (Definition 6, Page 18) retrieves all the sub contexts in E whose dimensions are in D.

  Projection$_c$(E, D) = if((IsPart(dim$_m$(first E)), D) == true)

  then first E fby Projection$_c$(next E, D)

  else Projection$_c$(next E, D)

- *Hiding*$_c(E, D)$ (Definition 7, Page 19) removes all the contexts in E whose dimensions are in D. According to the rule $c \uparrow D = c \ominus (c \downarrow D)$ provided in Section 2.1 (Page 19), we implement *Hiding*$_c(E, D)$ as follows:

  Hiding$_c$(E, D) = Difference$_c$(E, Projection$_c$(E, D))

- *Substitution*$_c(E, \langle d, t \rangle)$ (Definition 8, Page 19) replaces all the micro contexts in E whose dimension is d with the micro context $[d : t]$.

  $\texttt{Substitution}_\texttt{c}(\texttt{E}, \langle \texttt{d}, \texttt{t} \rangle) = \texttt{Union}(\texttt{construct}(\texttt{d}, \texttt{t}),\ \texttt{Hiding}_\texttt{c}(\texttt{c},\ \texttt{construct}_\texttt{d}(\texttt{d})))$

- *DisUnion*(c, E) produces the union of the context c and each element c_i of the context set E, where $c_i \in E$. Each result will be added into the set S.

  $\texttt{DisUnion}(\ \texttt{c}, \texttt{E}\ ) = \texttt{add}(\texttt{Union}(\texttt{c},\ \texttt{first E}), \texttt{S})\ \texttt{fby}\ \texttt{DisUnion}(\texttt{c}, \texttt{next E})$

- *PairwiseUnion*(S_1, S_2) for every context pair $c_1 \in S_1$ and $c_2 \in S_2$, produce the union of c_1 and c_2, and adds the result into the set S.

  $\texttt{PairwiseUnion}(\texttt{S}_1, \texttt{S}_2) = \texttt{DisUnion}(\texttt{first}\ \texttt{S}_1, \texttt{S}_2)$

  $\texttt{fby}\ \texttt{PairwiseUnion}(\texttt{next}\ \texttt{S}_1, \texttt{S}_2)$

Translation Rules for Context Operators

- *Intersection Function*: *Intersection*$(E, E')(\cap)$

  $\texttt{Intersection}(\texttt{E}, \texttt{E}') = \texttt{if}\ (\texttt{IsMember}(\texttt{first}\ \texttt{E}, \texttt{E}') == \texttt{true})$

  $\texttt{then first E fby Intersection}(\texttt{next E}, \texttt{E}')$

  $\texttt{else Intersection}(\texttt{next E}, \texttt{E}')$

- *Union Function*: *Union*$(E, E')(\cup)$

  $\texttt{Union}(\texttt{E}, \texttt{E}') = \texttt{E}'\ \texttt{fby}\ \texttt{Difference}_\texttt{c}(\texttt{E}, \texttt{E}')$

- *Undirected Range Function* : *Range*$(E, E')(\rightleftharpoons$, Definition 9, Page 20)

In order to provide the Undirected Range Function, we provide several elementary constructs as follows.

1. *subrange*(m_1, m_2) guarantees that the construction is sorted increasingly according to tag values, where $tag_m(m_1), tag_m(m_2) \in \mathbb{N}$.(Definition 9, Page 20 : Step 2 (a, b))

   $\mathtt{subrange}(\mathtt{m}_1, \mathtt{m}_2) = \mathtt{if}(\mathtt{tag}_\mathtt{m}(\mathtt{m}_1) < \mathtt{tag}_\mathtt{m}(\mathtt{m}_2))$

   $\quad \mathtt{then}\ \mathtt{construct}_{\mathtt{sub}}(\mathtt{dim}_\mathtt{m}(\mathtt{m}_1), \mathtt{tag}_\mathtt{m}(\mathtt{m}_1), \mathtt{tag}_\mathtt{m}(\mathtt{m}_2))$

   $\quad \mathtt{else}\ \mathtt{construct}_{\mathtt{sub}}(\mathtt{dim}_\mathtt{m}(\mathtt{m}_1), \mathtt{tag}_\mathtt{m}(\mathtt{m}_2), \mathtt{tag}_\mathtt{m}(\mathtt{m}_1))$

2. *construct*$_{sub}(d, t_1, t_2)$ generates a set of contexts S_a, whose elements have the same dimension d and tag values ranging from t_1 to t_2, where $t_1, t_2 \in \mathbb{N}$.(Definition 9, Page 20 : Step 2 (c))

   $\mathtt{construct}_{\mathtt{sub}}(\mathtt{d}, \mathtt{t}_1, \mathtt{t}_2) = \mathtt{if}(\mathtt{t}_1 \leq \mathtt{t}_2)$

   $\quad \mathtt{then}\ \mathtt{add}(\mathtt{construct}(\mathtt{d}, \mathtt{t}_1), \mathtt{S}_\mathtt{a})\ \mathtt{fby}\ \mathtt{construct}_{\mathtt{sub}}(\mathtt{d}, \mathtt{t}_1 + 1, \mathtt{t}_2)$

   $\quad \mathtt{else}\ \mathtt{S}_\mathtt{a}$

3. *construct*$_Y(E, E', D)$ construct a set Y, each element of which is a set Y_i. Y_i is constructed for each pair $m_i \in E$ and $m_i \in E'$ which share the same dimension. (Definition 9, Page 20 : Step 3).

   $\mathtt{construct}_\mathtt{Y}(\mathtt{E}, \mathtt{E}', \mathtt{D}) =$

   $\quad \mathtt{subrange}(\mathtt{Projection}_\mathtt{c}(\mathtt{E}, \mathtt{first\ D}), \mathtt{Projection}_\mathtt{c}(\mathtt{E}', \mathtt{first\ D}))$

   $\quad\quad \mathtt{fby}\ \mathtt{construct}_\mathtt{Y}(\mathtt{E}, \mathtt{E}', \mathtt{next\ D})$

4. *construct*$_Z(Y)$ constructs the set Z of contexts for each element in Y(Definition 9, Page 20 : Step 4)

$\texttt{construct}_\texttt{Z}(\texttt{Y}) = \texttt{if (iseod(Z))}$

$\texttt{then Z} = \texttt{PairwiseUnion(first Y, first next Y)}$

$\texttt{fby construct}_\texttt{Z}\texttt{(next next Y)}$

$\texttt{else Z} = \texttt{PairwiseUnion(prev Z, first Y) fby construct}_\texttt{Z}\texttt{(next Y)}$

5. *append*$_s(c_1, c_2, c_3, S)$ unions the contexts c_1, c_2, c_3 and adds the result to the set S. (Definition 9, Page 20 : Step 7)

$\texttt{append}_\texttt{s}(\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{S})$

$= \texttt{if(IsEmpty(S)) then S} = \texttt{construct}_\texttt{s}\texttt{(Union(c}_1, \texttt{Union(c}_2, \texttt{c}_3\texttt{)))}$

$\texttt{else add(Union(c}_1, \texttt{Union(c}_2, \texttt{c}_3\texttt{)), S)}$

$\texttt{Range(E, E}') = S$

where

$\texttt{S} = \texttt{if(InterEmpty(dim(E), dim(E}')\texttt{) then construct}_\texttt{s}\texttt{(Union(E, E}'\texttt{))}$

$\texttt{else concatenate}_\texttt{a}\texttt{(S}_\texttt{a}, \texttt{C}_\texttt{b}, \texttt{C}_\texttt{c}, \texttt{S);}$

$\texttt{concatenate}_\texttt{a}\texttt{(S}_\texttt{a}, \texttt{C}_\texttt{b}, \texttt{C}_\texttt{c}, \texttt{S)} = \texttt{append}_\texttt{s}\texttt{(first S}_\texttt{a}, \texttt{C}_\texttt{b}, \texttt{C}_\texttt{c}, \texttt{S)}$

$\texttt{fby concatenate}_\texttt{a}\texttt{(next S}_\texttt{a}, \texttt{C}_\texttt{b}, \texttt{C}_\texttt{c}, \texttt{S)}$ [*Definition* 9 : *Step* 7]

$\texttt{S}_\texttt{a} = \texttt{construct}_\texttt{Z}\texttt{(construct}_\texttt{Y}\texttt{(E, E}', \texttt{D))}$ [*Definition* 9 : *Step* 4]

$\texttt{C}_\texttt{b} = \texttt{Hiding}_\texttt{c}\texttt{(E, D)}$ [*Definition* 9 : *Step* 5]

$\texttt{C}_\texttt{c} = \texttt{Hiding}_\texttt{c}\texttt{(E}', \texttt{D)}$ [*Definition* 9 : *Step* 6]

$\mathtt{D} = \mathtt{dim(E)} \cap \mathtt{dim(E')}$

end

- *Directed Range Function* : *DirRange*(E, E')($\rightharpoonup$, Definition 10, Page 21)

  According to the definitions, the only difference between *Range*(E, E') and *DirRange*(E, E') is in the definition of function *subrange*(m_1, m_2), which is shown as follows.

  $\mathtt{subrange(m_1, m_2)} = \mathtt{if(tag_m(m_1)} < \mathtt{tag_m(m_2))}$

  $\mathtt{then\ construct_{sub}(dim_m(m_1), tag_m(m_1), tag_m(m_2))}$

  $\mathtt{else\ return}\ \varnothing$

  The rest of the constructions are as described above.

- *Choice Function* : *Choice*(E, E_1)(|, Definition 5, Page 18). This function may call some random number generation function in Java to choose either E or E_1.

- *Subcontext Function* : *Subcontext*(E, E')($\subseteq$)

  $\mathtt{Subcontext(E, E')}$

  $= \mathtt{if(IsMember(first\ E, E'))}$

  $\mathtt{then\ if\ (iseod(next\ E))\ then\ true}$

  $\mathtt{else\ Subcontext(next\ E, E')}$

  $\mathtt{else\ false}$

- *Supset Function* : *Supset*(E, E')($\supseteq$)

  $\mathtt{Supcontext(E, E')} = \mathtt{Subcontext(E', E)}$

- *Equality Function* : *Equality*$(E, E')(=)$

  $\texttt{Equality(E, E')} = \texttt{if(Subcontext(E, E')\&\&Supcontext(E, E'))}$

  $\texttt{then true else false}$

5.2.2 Translation Rules for Lifting Operators

Primitive Functions :

- *IsSet*(E) is used to check whether E is a set of contexts or a context. If yes, then return true, else false.

- *IsBox*(E) is used to check whether E is a *Box* or not. If yes, then return true, else false. Since the lifted operators *Override*$(\oplus)$ and *Difference*$(\ominus)$ cannot be applied to *Box*, the function *IsBox*(E) will be used in the translation rules of these two operators. Other lifted operators and relational operators have the same semantic meaning both for context sets and for *Box*es, so there is no need to provide translation rules for *Box*es separately.

Translation Rules for Lifting Operators

- *Projection Function*: *Projection*$(E, D)(\downarrow$, Definition 12, Page 26)

  $\texttt{Projection(E, D)} = \texttt{E}_1$

  where

  $\texttt{E}_1 = \texttt{if (IsSet(E))}$

then $\texttt{Projection}_c(\texttt{first E, D})$ fby $\texttt{Projection}(\texttt{next E, D})$

else $\texttt{Projection}_c(\texttt{E, D})$

end

- *Hiding Function*: *Hiding*$(E, D)(\uparrow$, Definition 13, Page 27)

$\texttt{Hiding}(\texttt{E, D}) = \texttt{E}_1$

where

$\texttt{E}_1$ = if (IsSet(E))

then $\texttt{Hiding}_c(\texttt{first E, D})$ fby $\texttt{Hiding}(\texttt{next E, D})$

else $\texttt{Hiding}_c(\texttt{E, D})$

end

- *Substitution Function*: *Substitution*$(E, \langle d, t\rangle)$ (/, Definition 14, Page 27)

$\texttt{Substitution}(\texttt{E}, \langle \texttt{d, t}\rangle) = \texttt{E}_1$

where

$\texttt{E}_1$ = if(IsSet(E))

then $\texttt{Substitution}_c(\texttt{first E}, \langle \texttt{d, t}\rangle)$ fby $\texttt{Substitution}(\texttt{next E}, \langle \texttt{d, t}\rangle)$

else $\texttt{Substitution}_c(\texttt{E}, \langle \texttt{d, t}\rangle)$

end

- *Override Function*: *Override*$(E, E')(\oplus$, Definition 16, Page 27)

$\texttt{Override}(\texttt{E, E}') = \texttt{E}_1$

where

E_1 = if (IsBox(E)) then false else if (IsSet(E))

then Override_c(first E, E′) fby Override(next E, E′)

else Override_c(E, E′)

end

- *Difference Function*: *Difference*$(E, E')(\ominus$, Definition 17, Page 28)

Difference(E, E′) = E_1

where

E_1 = if (IsBox(E)) then false else if(IsSet(E))

then Difference_c(first E, E′) fby Difference(next E, E′)

else Difference_c(E, E′)

end

5.2.3 Translation Rules for Relational Operators

Elementary Constructs :

- *compareSet*(c, s, D) compares the context c with each element $c_i \in s$. If the results of the projection of c and c_i on D are the same, c and c_i will be combined. The combined context will be added into the produced context set S.

compareSet(c, s, D) = S

where

if (Equality(Projection$_c$(c, D), Projection$_c$(first s, D)))

then add(Union(c, first s), S) fby compareSet(c, next s, D)

else compareSet(c, next s, D)

end

Translation Rules for Relational Operators

- *Join Function*: *JoinSet*(S_1, S_2) ($\boxtimes$, Definition 18, Page 29)

  JoinSet(S_1, S_2) = compareSet(first S_1, S_2, D) fby JoinSet(next S_1, S_2)

  where

  D = (dim(S_1) $\cap$ dim(S_2))

  end

- *Set Intersection Function*: *InterSet*(S_1, S_2) ($\boxdot$, Definition 19, Page 29)

  InterSet(S_1, S_2) = S

  where

  S = Projection(JoinSet(S_1, S_2), (dim(S_1) $\cap$ dim(S_2)))

  end

- *Set Union Function*: *UnionSet*(S_1, S_2)($\boxplus$, Definition 20, Page 29)

  UnionSet(S_1, S_2) = PairwiseUnion(X_1, X_2) [*Definition* 20, *Step* 3]

where

$\mathtt{X_1} = \mathtt{PairwiseUnion}(\mathtt{S_1}, \mathtt{Hiding}(\mathtt{S_2}, \mathtt{D}))$ [*Definition* 20, *Step* 1]

$\mathtt{X_2} = \mathtt{PairwiseUnion}(\mathtt{S_2}, \mathtt{Hiding}(\mathtt{S_1}, \mathtt{D}))$ [*Definition* 20, *Step* 2]

$\mathtt{D} = (\mathtt{dim}(\mathtt{S_1}) \cap \mathtt{dim}(\mathtt{S_2}))$

end

5.3 A Proof of Translation Rules

In this section, we show our approach of proving the equivalence between the formal definitions of context operators and the translation rules for them given in the previous section. The translation rules are recursively defined. This suggests a proof by induction approach. As an example of the proof approach, the proof of the override operator($\oplus$) is shown below.

The translation rule for override operator is : $Override_c(c_1, c_2) = c_2 \, fby \, extract(c_1, c_2)$. In order to prove that this rule correctly translates the definition of override operator, we need the following propositions.

Proposition 1 *The function* $extract(c_1, c_2)$ *constructs the set:*

$Y = \{m \mid (m \in c_1 \wedge dim_m(m) \notin dim(c_2))\}$.

Proof We give an inductive proof based on the length of c_1.

Base step $c_1 = NULL$

$extract(NULL, c_2) = NULL$ [Definition of extract]

Y = NULL [Set Theory]

Hence, $extract(c_1, c_2) = Y = NULL$.

Induction step $c_1 \neq NULL$. Let k be the length of c_1, $k \geq 1$. Assume that $extract(c_1, c_2)$ constructs the set Y_k for all streams c_1 of length $k \geq 1$. We prove that $extract(c_1', c_2)$, where c_1' is of length $k+1$, will construct the set Y_{k+1}.

We write $c_1' = first\ c_1'\ fby\ next\ c_1'$. The length of $next\ c_1'$ is k.

Apply the function definition of $extract(c_1', c_2)$:

(1). If $(IsPart(dim_m(first\ c_1'),\ dim(c_2)) == false)$, then we construct:

$first(c_1')\ fby\ extract(next\ c_1', c_2)$

By induction, $extract(next\ c_1', c_2)$ constructs the set Y_k,

Hence $first(c_1')\ fby\ extract(next\ c_1', c_2)$ constructs the set $Y_k \cup \{first(c_1')\} = Y_{k+1}$.

Proved.

(2). If $(IsPart(dim_m(firstc_1'), dim(c_2)) \neq false)$, then we construct:

$extract(next\ c_1', c_2)$. According to induction step, $extract(next\ c_1', c_2)$ constructs the set Y_k correctly because $next\ c_1'$ is a stream of length k. In this case $Y_k = Y_{k+1}$

Hence the result follows.

Proposition 2 $c_1 \oplus c_2 = Override_c(c_1, c_2)$.

Proof

$c_1 \oplus c_2 = \{m \mid (m \in c_1 \wedge dim_m(m) \notin dim(c_2)) \vee m \in c_2\}$ [Definition 4]

$= \{m \mid m \in c_2\} \cup \{m \mid (m \in c_1 \wedge dim_m(m) \notin dim(c_2))\}$ [Set Theory]

$= c_2\ fby\ \{m \mid (m \in c_1 \wedge dim_m(m) \notin dim(c_2))\}$ [Definition 1]

$= c_2\ fby\ extract(c_1, c_2)$ [Proposition 1]

$= Override_c(c_1, c_2)$ [$Override_c$ Definition]

5.4 Applying Translation Rules for Program Development

Lucx programs contain context operators and context set operators. The translation rules implement the definitions of these operators in Indexical Lucid. Provided with Java implementations of primitive functions, we claim that every Lucx program can be presented as an Indexical Lucid program, and an equivalent dataflow diagram can be developed. Example 27 shows the Indexical Lucid program developed from the Lucx program of Example 26 (Page 59). The corresponding dataflow diagram is shown in Figure 13. Since the GIPSY has provided the tools transforming dataflow diagram into Indexical Lucid program, this dataflow diagram can also be transformed into Indexical Lucid program and compiled on the GIPSY platform.

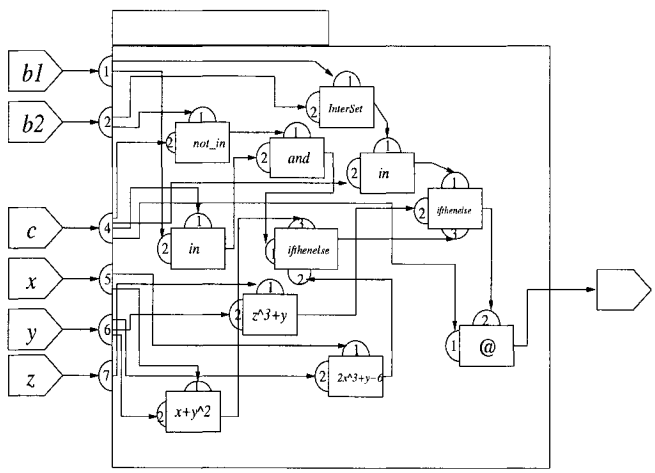

Figure 13: Translated programs

Example 27 *:*

$\mathtt{f(x,y,z,c)}$

$= \mathtt{if\ c\ } \in \mathtt{\ b_1} \boxdot \mathtt{b_2,\ then\ z^3 + y}$

$$\texttt{else if } (c \in b_1 \&\& c \notin b_2), \texttt{ then } 2x^3 + y - 6$$

$$\texttt{else } x + y^2$$

where

$$b_1 = \texttt{Box}[D, E, F \mid D^2 + E^2 \leq \tfrac{F^2}{4} \wedge 0 \leq F \leq 4]$$

$$b_2 = \texttt{Box}[D, E, F \mid D^2 + E^2 + F^2 \leq 9 \wedge F \geq 0]$$

$$\texttt{InterSet}(b_1, b_2) = \texttt{Projection}(\texttt{JoinSet}(b_1, b_2), (\texttt{dim}(b_1) \cap \texttt{dim}(b_2)))$$

$$\texttt{JoinSet}(b_1, b_2) = \texttt{compareSet}(\texttt{first } b_1, b_2, (\texttt{dim}(b_1) \cap \texttt{dim}(b_2)))$$

$$\texttt{fby JoinSet}(\texttt{next } b_1, b_2)$$

$$\texttt{Projection}(\texttt{JoinSet}(b_1, b_2), (\texttt{dim}(b_1) \cap \texttt{dim}(b_2))) = \texttt{if } (\texttt{IsSet}(E))$$

$$\texttt{then Projection}_c(\texttt{first } E, D) \texttt{ fby Projection}(\texttt{next } E, D)$$

$$\texttt{else Projection}_c(E, D)$$

end

5.5 Summary

In this chapter, we have given one approach to integrating Lucx into the GIPSY. The translation rules for context operators, context set operators and *Box* operators are given. These translation rules form a basis for Lucx-GIPL AST translator. The equivalence between formal definitions of the operators and the translation rules requires a formal proof. We have outlined an induction-based proof approach for proving the equivalence between the formal definition and the translation rule of override operator. Providing proofs for the rest of the rules will be taken up as part of future work.

Another possible approach of integrating Lucx into GIPSY is to implement these operators using Java functions and embed these Java functions into GEE. The approach that we have taken, namely providing translation rules, is appropriate to formally show the correctness of an implementation. Once we have established the equivalence between the formal definitions and their translation rules, Java implementations for Lucx operators will improve the evaluation efficiency while assuring the correctness of what is implemented.

Chapter 6

Programming Timed Systems in Lucx

6.1 Introduction

In this chapter we discuss an approach to programming timed systems in Lucx language. To program timed systems, specification and programming languages must have the expressive power to describe precisely what the actions are and when these actions must be taken. Lucx language is declarative, and hence Lucx programs implementing the actions are precise, abstract, and yet have sufficient details for understandability. As shown later in the chapter, context in Lucx can be used to specify time constraints, either with a global clock or with several local clocks. Time-dependent functionality can be programmed as function evaluations at contexts. Stream processing in Lucx seems natural to programming the asynchronous arrival of environmental events in *reactive* systems and *hybrid* systems, the two important kinds of real-time systems.

Reactive systems are systems that maintain a *continual* interaction with their environment. The two properties that characterize reactive systems are that the process always reacts to a stimulus from its environment (*stimulus synchronisation*), and the time elapsed between a stimulus and its response is acceptable to the relative dynamics of the environment, so that the environment is still receptive to the response (*response synchronization*). Hybrid systems are systems that combine *continuous* and *discrete* behavior. The continuous and discrete parts in the system interact at discrete points in time. The physical nature of a component as well as its interaction with other hybrid system components are usually expressed as algebraic and/or differential equations. Most reactive systems are message-intensive, whereas hybrid systems are computation-intensive. The extent of synchronization, due to data (function) computation, distinguishes hybrid systems from reactive systems. Yet, both reactive and hybrid systems are complex to model and analyze. Many of the *safety-critical* systems are either reactive or hybrid.

The synchronous dataflow languages Lustre [17] and RLucid [48] have been applied for reactive programming. Clocks were added to Lustre programs so that certain parts of the programs need not always run. This enabled the introduction of constrained reaction. In RLucid the operator *before* was introduced to deal with real time. That is, one can write the expression E_1 *before* E_2 to determine whether the first value in the stream E_1 arrived at time $t_1 < t_2$, where t_2 is the time of arrival of the first value of E_2. SIGNAL [42] language manipulates *signals* that are timed sequences of typed values. In all these approaches time is discrete, and streams implicitly have the time dimension, although clocks associated with dimensions may be different. For hybrid system specification, hybrid [15] and timed

automata [56] are the two main formalisms. Not much work has been reported in language aspects for programming hybrid systems.

In the rest of this chapter we provide a detailed discussion on real-time reactive programming. At the end of the chapter we point out the necessary extensions to program hybrid systems in Lucx.

6.2 Abstract Models of Timed Systems

In this section we discuss a formal basis of timed systems on which Lucx programs are to be written. Time, measured by one or more clocks, may be either discrete or continuous. We denote a set of clocks by $\mathcal{C}$. A *clock valuation* v is defined as a higher-order function $v : \mathcal{C} \rightarrow (\Omega \rightarrow \overline{\Pi})$ such that the function $v(c), c \in \mathcal{C}$ is monotonically increasing function. Moreover, for $c, c' \in \mathcal{C}$, the functions $v(c)$ and $v(c')$ increase at the same rate. For continuous time model, $\Omega = \mathcal{R}^{\infty}$, and the function $v(c)$ is continuous for every $c \in C$. For discrete time model, $\Omega = \mathbb{N} \cup \{0\}$. For both time models, $v(c)(0) = 0$.

Global Clock Let $\mathbb{C}$ denote the global clock, $\mathbb{N}$ denote the set of nonnegative integers, $\mathbb{R}$ denote the set of reals, and $\mathbb{R}^{\geq 0}$ the set of nonnegative reals. We assume that continuously varying time is modeled as $\overline{\Pi} = \{t \mid t \in \mathbb{R}^{\geq 0}\}$. The model of discrete time is $\overline{\Pi} = \mathbb{N} \cup \{0\} \cup \{+\infty\}$.

Multiple Clocks We let $\mathcal{C}$ denote the set of clocks in the system. Let us consider applications in which a clock c is never compared with a time constant greater than m. Then,

the actual clock valuation, once it exceeds m, is of no consequence in deciding the allowed execution paths in the program. Hence every clock $c \in C$ has a bounded support, which we denote as $Intv(c)$.

For continuous time model an equivalence relation for clock valuations is defined [56]: $v \cong v'$ iff, for all $c_1,\ c_2 \in C,\ x \in \mathbb{R}^{\geq 0}$:

1. $Intv(c_1) = Intv(c_2)$

2. $\lfloor v(c_1)(x) \rfloor = \lfloor v'(c_1)(x) \rfloor$ and $(fract(v(c_1)(x)) = 0$ iff $fract(v'(c_1)(x)) = 0)$,

3. $fract(v(c_1)(x)) \leq fract(v(c_2)(x))$ iff $fract(v'(c_1)(x)) \leq fract(v'(c_2)(x))$.

If two clock valuations v and v' are equivalent, then $v(c)(x)[\delta] = v'(c)(x)[\delta]$ for any clock predicate δ.

A *clock region* is an equivalence class of clock valuations induced by equivalence relation $\cong$. We say that a clock region α satisfies a clock constraint δ iff every $v \in \alpha$ satisfies δ. Each region can be uniquely characterized by a (finite) set of clock constraints it satisfies. Each region can be represented by specifying

(1) for every clock c, one clock constraint from the set $\{v(c)(x) = m \mid m = 0, 1, \ldots, m_c\}$ $\cup \{m - 1 < v(c)(x) < m \mid m = 1, \ldots, m_c\} \cup \{v(c)(x) > m_c\}$, where m_c is the supremum of $Intv(c)$, and $x \in \mathbb{R}^{\geq 0}$

(2) for every pair of clocks c_1 and c_2 such that $m_1 - 1 < v(c_1)(x) < m_1$ and $m_2 - 1 < v(c_2)(x) < m_2$ appear in (1) for some m_1,m_2, whether $fract(v(c_1)(x))$ is less than, equal to, or greater than $fract(v(c_2)(y))$.

As an example, consider clocks c_1 and c_2 with $m_1 = 4$, and $m_2 = 6$. This gives rise to 59 clock regions, as shown in Figure 14. Each region is interpreted by the clock values according to the equivalence relation definition. For instance, (open) regions $\alpha 1$ and $\alpha 16$ are defined by the inequalities

$\alpha 1: \quad 0 < v(c_1)(x) < 1, 0 < v(c_2)(y) < v(c_1)(x)$

$\alpha 16: \quad 3 < v(c_1)(x) < 4, v(c1)(x) - 2 < v(c_2)(y) < 2$

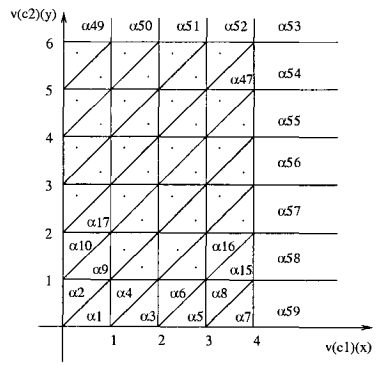

Figure 14: Clock Regions

A clock region α' is a *time-successor* of a clock region α iff for each $v \in \alpha$, there exists a positive $t \in \mathbb{R}$ such that $v + t \in \alpha'$. The time-successors of a clock region α are all the clock regions that will be visited by a clock valuation $v \in \alpha$ as time progresses. The time-successors of a region α can be derived by moving along a line drawn from some point in α in the diagonally upwards direction(parallel to the line $x = y$). For instance, in Figure 14, the successors of region $\alpha 1$ are : $\alpha 4$, $\alpha 11$, $\alpha 14$, $\alpha 21$, $\alpha 24$, $\alpha 31$, $\alpha 53$, $\alpha 54$, $\alpha 55$, $\alpha 56$.

Timed Systems The object-oriented (OO) formal approach [1, 43] is taken as the basis for programming real-time reactive systems in Lucx. The main reason for this choice is

that several tools exist in TROMLAB [1] for developing a timed system according to the OO formalism, and an implementation of Lucx in GIPSY might provide a link between these two frameworks.

The OO approach formalizes a reactive system as a composition of two parts:

- *a formal model of the environment*: the environmental objects are abstracted, their interfaces to the system elements are defined, and their behavior models are developed;

- *a formal model of the system elements*: the reactive process is modeled and its functional and timing requirements are specified.

For verification purposes, a formal statement of the desired properties of the system is formulated; but it is not part of the OO model.

The behavior model for the objects in the OO approach is an extended state machine (ESM) [43]. An ESM has a finite number of states. The states in an ESM roughly correspond to different modes or situations of interest. A state may include one or more discrete data type variables whose values change whenever the state is reached in the execution of the object. State transitions are labeled with event (action) names, and specified in *guard-action* paradigm. The guard g on a transition from state s_i to s_j is of the form $var_g \wedge tc$, where var_g, a conjunction of predicates on the variables in state s_i, serves as a precondition for enabling the transition and tc is a conjunction of time constraint predicates of the form $lower \leq tc < upper$. The action a is a conjunction of two predicates, one on the variables in the post state s_j, and the other on the clock variables that need to be initialized. The

clock initialization predicate is optional. The state transition semantics is shown below(let $\mathcal{E}$ denote a finite non-empty set of events e):

$$\frac{s_i \wedge e \in \mathcal{E}(s_i) \wedge var_g(s_i) \wedge tc(t)}{s_i \xrightarrow{e} s_j \wedge var_a(s_j) \wedge rtc(t)}$$

An *execution* is a sequence of transitions starting from an initial state. The behavior of the ESM is the set of executions. The behavior model of a non-trivial system consists of several ESMs, and they interact through messages. For all ESMs, in the initial state the current time is 0, an unconstrained transition is instantaneous and hence does not change clock values, a time constrained transition makes the time progress and the time passage is additive.

6.3 Lucx Programs for Abstract Models

In this section we discuss the representation of *events*, *variables*, *functions*, and *states* of timed systems as streams in Lucx. We discuss the representations for both discrete and continuous time models. These stream representations are similar to the representations in the functional model introduced by Alagar and Ramanathan [8].

With the dimension name (global clock reference) c and the tag set Ω, global clock reference at an instance x is given by the micro_context $[c : x]$. If v is a clock evaluation function, the expression $v@[c : x]$ evaluates to the time shown in the global clock at instance x. Because there is only one global clock, we can often ignore explicit reference to c.

The clock regions corresponding to a set of clocks is represented as a finite stream of *Box* expressions. Every *Box* expression in the stream corresponds to one region. Each *Box*

expression is defined by the dimension set $\Delta = \{c_1, \ldots, c_k\}$, and a constraint on the clock valuations. For example, $Box[\Delta \mid p_1]$, where $\Delta = \{c_1, c_2\}$ and $p_1 = 0 < v(c_1)(x) < 1, 0 < v(c_2)(y) < v(c_1)(x)$ refers to the region α_1 in Figure 14.

In Lucx the time defined by the real-time clock of a computer is not the only possible time that can be expressed. Any input event can define its own time by means of its repeated occurrences. Replacing the v by *TIME*, and C by $\mathcal{E}$ in the definition of clock valuation we get the function $TIME : \mathcal{E} \rightarrow (\Omega \rightarrow \overline{\Pi})$, which is the time defined by the occurrences of events in $\mathcal{E}$. Lucx representations for these two kinds of times are different, and consequently the program is capable of figuring out how the different times should be combined. In the rest of this chapter, we consider only two types of time: 1. the time defined by the global clock; and 2. the time defined by the event occurrences.

Event Streams Let $\mathcal{E}$ denote a finite non-empty set of *events* in the formal model of the system. An event $e \in \mathcal{E}$ may occur any number of times within the system. The function $TIME : \mathcal{E} \rightarrow (\mathbb{N} \rightarrow \overline{\Pi})$ defines for $e \in \mathcal{E}$, the function $TIME(e)$, whose value at $k \in \mathbb{N}$ is $t_k = TIME(e)(k)$, $t_k \in \overline{\Pi}$, interpreted as the time of *kth* occurrence of the event e. The function $TIME(e) = \{\langle k, t_k \rangle\}$ is represented in the language as a 1-dimensional stream $\overline{\mathbf{e}}$, $\overline{\mathbf{e}}_k = t_k$. In the language the representation for an event e under continuous time model is the stream $\overline{\mathbf{e}}$, and under discrete time model, the representation can be either $\overline{\mathbf{e}}$ or a boolean stream $\mathbf{e}$ such that $\mathbf{e}_k$ is *true*.

The function $COUNT : \mathcal{E} \rightarrow (\overline{\Pi} \rightarrow \mathbb{N})$ defines for $e \in \mathcal{E}$, the function $COUNT(e)$, whose value at $t \in \overline{\Pi}$ is $k = COUNT(e)(t)$, $k \in \mathbb{N}$ is the number of occurrences of the

event e up to and including the time t. That is, the function *COUNT* is a pseudo inverse of *TIME* function. That is, $TIME(e)(0) = 0$; $TIME(e)(+\infty) = +\infty$, and $COUNT(e)(0) = 0$; $COUNT(e)(+\infty) = +\infty$.

Example 28 *Let the times for* 1^{st}, 2^{nd}, 3^{rd}, 4^{th} ... *occurrences of an event e be* $1, 4, 5, 7, \ldots$

For discrete time, the representation of the event e is the stream

$\mathbf{e} =$ *true false false true true false true false* ...

The representation of the stream $\bar{\mathbf{e}}$ *is*

$\bar{\mathbf{e}} = 1 \quad 4 \quad 5 \quad 7 \ldots$

The representation of the stream COUNT(e) is

$COUNT(e) = 1 \quad 1 \quad 1 \quad 2 \quad 3 \quad 3 \quad 4 \ldots$

Let the times for 1^{st}, 2^{nd}, 3^{rd}, 4^{th}, ... *occurrences of an event f be* $1.3, 4.5, 5.6, 7.8, \ldots$, *in some clock valuation. The representation is the stream*

$\bar{\mathbf{f}} = 1.3 \quad 4.5 \quad 5.6 \quad 7.8, \ldots$

The representation of the stream COUNT(f) is

$COUNT(f)(1) = 0, \ \ldots \ COUNT(f)(1.3) = 1, \ \ldots, \ COUNT(f)(2) = 1, \ \ldots,$

$COUNT(f)(4.5) = 2, \ \ldots, \ COUNT(f)(5) = 2, \ \ldots, \ COUNT(f)(5.6) = 3, \ \ldots$

The following primitive functions defined in the language are useful to manipulate event streams. Let *now* denote current clock valuation. The arguments to the following functions are streams corresponding to events.

- The function `includes`$(\mathbf{e}, \mathbf{f})$ returns *true* if $TIME(e) \leq TIME(f)$, otherwise it returns false.

- The function $\texttt{sum}(\mathbf{e}, \mathbf{f})$ returns the stream obtained by merging the two input streams. The resulting stream represents the event $e + f$, which occurs whenever e occurs or f occurs.

- The function $\texttt{last\_time}(\bar{\mathbf{e}}, \texttt{t})$ returns the latest time $t_1 < t < now$ at which e occurred.

- The function $\texttt{next\_time}(\bar{\mathbf{e}}, \texttt{t})$ returns the most recent time $t_1 > t < now$ at which e occurred.

- The function $\texttt{extract}(\mathbf{e}, \texttt{p})$, where p is a predicate, extracts the sub-stream $\mathbf{f}$ of stream $\mathbf{e}$ such that the predicate p is true at every occurrence of f. For instance, if the predicate is $\texttt{COUNT}(\texttt{e}, \texttt{t}) = \texttt{COUNT}(\texttt{g}, \texttt{t})$ the function $\texttt{extract}(\mathbf{e}, \texttt{p})$ extracts the sub-stream $\mathbf{f}$ of stream $\mathbf{e}$ such that $\texttt{COUNT}(\texttt{f}, \texttt{t}) = \texttt{k}$ implying that there exists an increasing sequence $0 \leq t_1 < t_2 < \ldots < t_{k-1} < t_k = t$, such that $\texttt{COUNT}(\texttt{g}, \texttt{t}_\texttt{i}) = \texttt{COUNT}(\texttt{e}, \texttt{t}_\texttt{i})$, for $i = 1, \ldots, k$.

Variable Stream Let $\mathcal{V}$ denote a finite non-empty set of *variables* in the formal model of the system. Whenever the variable v changes its value, the event $ASSIGN(v) \in \mathcal{E}$ occurs. The stream representation of $ASSIGN(v)$ is as defined before for events. The function $TIME(ASSIGN(v)) : (\mathbb{N} \rightarrow \overline{\Pi})$ defines $t_k = TIME(ASSIGN(v))(k)$ for $v \in \mathcal{V}$ and $k \in \mathbb{N}$. The time t_k is interpreted as the time of *kth* assignment for the variable v. The function $TIME(ASSIGN(v)) = \{\langle k, t_k \rangle\}$ is represented in the language as a 1-dimensional stream $\bar{\mathbf{v}}$, $\bar{\mathbf{v}}_k = t_k$.

The function $VAL : \mathcal{V} \rightarrow (\mathbb{N} \rightarrow \overline{\Pi})$ defines for $v \in \mathcal{V}$, the function $VAL(v)$, whose value at $k \in \mathbb{N}$ is $v_k \in \overline{\Pi}$, the value at the $k-th$ assignment. We represent $VAL(v)$ as the stream $\mathbf{v}$. Thus, for time t, $\overline{\mathbf{v}}_{k-1} < t < \overline{\mathbf{v}}_k$, the value of variable v remains as $\mathbf{v}_{k-1}$.

Example 29 *Let the times for* $1^{st}, 2^{nd}, 3^{rd}, 4^{th}, \ldots$ *assignments of a variable v be* $0.5, 1.0, 1.5, 2.0, \ldots$, *and the corresponding values of each assignment is* $1, 3.3, 4.5, 7.8, \ldots$, *So the representation of the value v is the stream:*

$\mathbf{v} = 1, \quad 3.3, \quad 4.5, \quad 7.8, \quad \ldots$

Meanwhile, the representation of the stream $\overline{\mathbf{v}}$ *is*

$\overline{\mathbf{v}} = 0.5, \quad 1.0, \quad 1.5, \quad 2.0, \quad \ldots$

The following primitive functions defined in the language are useful to manipulate variable streams.

1. The function $\texttt{last\_assign}(\mathbf{v}, \texttt{t})$ returns the latest time $t_1 < t <$ *now* at which the variable v changed its value.

2. The function $\texttt{next\_assign}(\mathbf{v}, \texttt{t})$ returns the most recent time $t_1 > t <$ *now* at which the variable v changes its value.

Function Call Streams A function call stream is a sequence of function calls that have been defined in the program. The evaluations of the function call $f(v_1, \ldots, v_n)$ at different instances produce a stream $\mathbf{v_f}$ of values. A predicate p is evaluated, as a function of its free variables, whenever a free variable in p gets a new value in the system. The following functions manipulate function streams: **F** is a function call, and *now* is the current clock valuation.

1. The function $\texttt{eval}(\mathbf{F}, \mathbf{w}, \texttt{t})$, evaluates the function call $\mathbf{F}$ at time t by binding the stream variables in $\mathbf{F}$ to the streams in the tuple $\mathbf{w}$, in the order specified. For each variable the latest assigned value is used in evaluating the function. The current value of the function is stream $\mathbf{v}_{\mathbf{f}t}$.

2. The function $\texttt{eval}(\mathbf{F}, \mathbf{w}, \texttt{p})$, evaluates the function call $\mathbf{F}$ whenever the predicate p on a subset of the variables $v_1, \ldots, v_n$ of the function $f(v_1, \ldots, v_n)$ becomes true. That is, if at time t the predicate p becomes true, the function $\texttt{eval}(\mathbf{F}, \mathbf{w}, \texttt{t})$ is invoked.

Streams for ESM Models

1. State transitions are modelled as a 3-dimensional stream **tf**, with state names as tags along the dimensions $STATE_{from}$, state machine names as tags along the dimension *Machine*, and transition numbers as tags along the dimension *TRAN*. The evaluation $\mathbf{tf}\ @\ [Machine : M_n, STATE_{from} : s_i,\ TRAN : k]$ is the tuple $\langle e, s_j \rangle$, where s_j is the transited state and e is the event triggering the transition from s_i to s_j for that particular state machine M_n.

2. The CNF parts of the guard p in a transition are represented by a 1-dimensional stream $\mathbf{var_g}$. Its dimension is *TRAN* having transition numbers as tags. The evaluation $\mathbf{var_g}@[TRAN : k]$ is a tuple of clauses in the CNF for that transition. A tuple, being a finite stream, has a selector function which retrieves a specific component of the tuple for the given tag. We use the notation $1^{st}, 2^{nd}, \ldots$ to denote the selector functions. With this notation and the use of predefined functions a CNF is evaluated. That is, the function $evalc(\mathbf{var_g}@[TRAN : k], \mathbf{w}, t)$ evaluates the conjunction of

clauses in its first argument by binding the variables to the stream **w** at time t. It uses the function $evald(\mathbf{tu}, \mathbf{w}, t)$ that evaluates the disjunction of predicates in the tuple **tu**. The function *evald* uses the function $eval(p, \mathbf{w}, t)$ that evaluates the predicate by binding the variable in p to the value stream **w** at time t.

3. The time constraint *tc* is represented as a 2-dimensional stream with one dimension name *TRAN* having transition numbers as tags, and another dimension *CLOCKVAR* having clock indices as tags. The evaluation **tc** @ $[TRAN : k, CLOCKVAR : i]$ is a tuple of integers $\langle lower_k, upper_k \rangle$, with the meaning that $[lower_k, upper_k]$ is the time interval specified as part of the specification of the transition whose number is k.

4. The conjunction var_a is represented as a 1-dimensional stream $\mathbf{var_a}$ with the dimension *TRAN* having transition numbers as tags. The evaluation $\mathbf{var_a}$ @ $[TRAN : k]$ a tuple of clauses in the CNF for that transition.

5. The predicate *rtc* is represented as a 2-dimensional stream **rtc** with one dimension name *TRAN* having transition numbers as tags, and another dimension *CLOCKVAR* having clock indices as tags. The evaluation **rtc** @ $[TRAN : k, CLOCKVAR : i]$ is a boolean value indicating whether or not the i^{th} clock should be reset at the transion k.

Dynamic Behavior of ESM The dynamic behavior of ESM is the set of traces produced according to the state transition semantics given in Section 6.2. We represent each trace of a machine by a stream of tuples $\langle s, v_s \rangle$ in the program, where $s \in \mathcal{S}$, a finite set of states in the formal model, and v_s is the set of variables in state s. An element of the trace

is computed by applying the state transition semantics to the element that was generated at the previous step. If event e occurs at time t, and is admissible for the current element in the trace, the transition happens instantaneously; if it is not admissible in this state, transition does not happen, but time is allowed to progress.

A timed system consists of several synchronously interacting state machines. The Lucx representation of the timed system is a 2-dimensional stream **P**. One of its dimension *MACHINE* has tags corresponding to the machines in the system. The evaluation $\mathbf{P}@[MACHINE : i]$ is the stream $\mathbf{M_i}$ for the state machine M_i in the model. The other dimension of the stream **P** is *TIME* having discrete instances as tags. The justification is that events in the system happen at discrete times. At instant k, the expression $\mathbf{P}@[TIME : k]$ evaluates to a 1-dimensional stream, showing the status of all the machines in the system at time $t_k = t$. The system state changes if constraints are satisfied for the state in a tuple on the tth column, otherwise time is allowed to progress. We calculate the function $progress(\mathbf{M_i}, t, e)$ to determine the current state tuple $\mathbf{M_{i(}}_{t)} = \langle s_i, v_{s_i} \rangle_{(t)}$. For all other rows, there is no change in time t:

$$\mathbf{P_t} = \langle \mathbf{M_{1_{prev\ t}}}, \ldots, \mathbf{M_{(i-1)_{prev\ t}}}, \mathbf{M_{i_t}}, \mathbf{M_{(i+1)_{prev\ t}}}, \ldots, \mathbf{M_{n_{prev\ t}}} \rangle$$

If the state machines $\mathbf{M_i}$ and $\mathbf{M_j}$ synchronize at time t on an event, then the state changes happen simultaneously in both machines, in rows i and j of the 2-dimensional stream while no other row in the stream will change.

If no event occurs but the sampling time is up, time should progress. A 2-dimensional stream **C** is also defined in the timed system. When time passes, the values of clocks are recorded in the stream **C**, one dimension of which is *TIME*, where $f_{dimtotag} = \mathbb{N}$ for discrete time or $f_{dimtotag} = \mathbb{R}$ for continuous time. Another dimension of **C** is *CLOCKVAR* having

clock indices as tags. The evaluation $\mathbf{C}$ @ $[TIME : t, CLOCKVAR : i]$ is the value of the clock variable i at time t. In Example 30, we define the function $timePassage(\mathbf{C}, t)$ to describe the behavior of the timed system when the sampling time is up. For simplicity, we use the discrete time. The program $eventOccur(e, t)$ is to describe the system behavior when the environmental event e occurs. We assume only one variable in $\mathbf{var_g}$ and $\mathbf{var_a}$ for simplicity. We use the functions 1^{st}, 2^{nd}, and 3^{rd} to select respectively the first, second and third components of tuple streams.

Example 30 *:*

- *timePassage*$(\mathbf{C}, t)$ *: when the sampling time is up,* $C = prev\ C + 1$. *In particular, each value of the clock variables will be increased by 1.*

  first $\mathbf{C}$@[TIME : t] = (first $\mathbf{C}$ @ [TIME : prev t] + 1) fby timePassage(next $\mathbf{C}$, t)

- *eventOccur*(e, t) *gets the environmental event e when it occurs and calls the function progressMachine*(M, e, t).

  M = $\mathbf{P}$ @ [Time : prev t];

  progressMachine(M, e, t);

- *progressMachine*(M, e, t) *calls the function progress*(M_i, e, t) *for every machine* M_i *which is in P @ [Time : prev t].*

  progressMachine(M, e, t) = progress(first M, e, t)

  fby progressMachine(next M, e, t);

- *progress*(M_i, e, t) *gets the previous state and variable in the stream* $\mathbf{P}$ *and calls the function progressEvent*$(tf_{temp}, e, state, var)$.

$\texttt{state}_{\texttt{from}} = 1^{\texttt{st}}.(\mathbf{P} \ @ \ [\texttt{Machine} : \texttt{M}_\texttt{i}, \ \texttt{Time} : \texttt{prev t}]);$

$\texttt{variable} = 2^{\texttt{nd}}.(\mathbf{P} \ @ \ [\texttt{Machine} : \texttt{M}_\texttt{i}, \ \texttt{Time} : \texttt{prev t}]);$

$\texttt{tf}_{\texttt{temp}} = \mathbf{tf} \ @ \ [\texttt{Machine} : \texttt{M}_\texttt{i}, \texttt{STATE}_{\texttt{from}} : \texttt{state}_{\texttt{from}}];$

$\texttt{progressEvent}(\texttt{tf}_{\texttt{temp}}, \texttt{e}, \texttt{state}_{\texttt{from}}, \texttt{variable});$

- *progressEvent*$(tf_{temp}, e, state, var)$ *calls the function* *IsAdmissible*$(tf_{tran}, e, state, var)$ *for each transition in the transition stream* **tf**.

  $\texttt{progressEvent}(\texttt{tftemp}, \texttt{e}, \texttt{state}, \texttt{var}) =$

  $\texttt{IsAdmissible}(\texttt{first tftemp}, \texttt{e}, \texttt{state}, \texttt{var})$

  $\texttt{fby progressEvent}(\texttt{next tftemp}, \texttt{e}, \texttt{state}, \texttt{var});$

- *IsAdmissible*$(tf_{tran}, e, state, var)$ *checks if the event e is included in* **tf** $@$ $[Machine : M_i, STATE_{from} : state_{from}, TRAN : k]$. *It does the following:*

  1. *Check if the event e is the one triggering the state transition;*
  2. *Check if the guard condition is satisfied through the function evalc*$(\mathbf{var_g} \ @ \ [TRAN : k], variable, t)$;
  3. *Check if the time constraint tc is satisfied through the function clockCheck;*
  4. *If all the above conditions are satisfied, then the function IsAdmissible does the following:*

     (a) *the state in the stream* **P** *is changed to the state* $state_{to}$ *in the stream tf.*

     (b) *the variable in the stream* **P** *is changed to the variable* v_i *in the stream* $\mathbf{var_g}$.

     (c) *the action* var_a *is executed through the function eval*$(1^{st}.\mathbf{var_a} \ @ \ [TRAN : k], 2^{nd}.\mathbf{var_a} \ @ \ [TRAN : k], t)$.

(d) Clocks are also reset through the function clockReset.

5. *If not all the conditions stated in (1),(2), and (3) are satisfied, then the state and the variable in* $\mathbf{P}$ *@* $[Machine : M_i, Time : t]$ *is the same as* $\mathbf{P}$ *@* $[Machine : M_i, Time :$ *prev* $t]$.

$\mathtt{k} = \mathtt{tf}_{\mathtt{tran}}.\mathtt{TRAN};$

$\mathtt{if}(1^{\mathtt{st}}.\mathtt{tf}_{\mathtt{tran}}@[\mathtt{TRAN} : \mathtt{k}] == \mathtt{e})\&\&(\mathtt{evalc}(\mathbf{var_g}@[\mathtt{TRAN} : \mathtt{k}], \mathtt{variable}, \mathtt{t}))\&\&$

$\mathtt{clockCheck}(\mathbf{C}@[\mathtt{Time} : \mathtt{t}], \mathbf{tc}@[\mathtt{TRAN} : \mathtt{k}])$

$\mathtt{then}\ 1^{\mathtt{st}}.(\mathbf{P}@[\mathtt{Machine} : \mathtt{M_i}, \mathtt{Time} : \mathtt{t}]) = 2^{\mathtt{nd}}.(\mathtt{tf}_{\mathtt{tran}}@[\mathtt{TRAN} : \mathtt{k}])$

$\mathtt{fby}\ 2^{\mathtt{nd}}.(\mathbf{P}@[\mathtt{Machine} : \mathtt{M_i}, \mathtt{Time} : \mathtt{t}]) = 2^{\mathtt{nd}}.(\mathbf{var_g}@[\mathtt{TRAN} : \mathtt{k}])$

$\mathtt{fby}\ \mathtt{eval}(1^{\mathtt{st}}.\mathbf{var_a}\ @\ [\mathtt{TRAN} : \mathtt{k}], 2^{\mathtt{nd}}.\mathbf{var_a}\ @\ [\mathtt{TRAN} : \mathtt{k}], \mathtt{t})$

$\mathtt{fby}\ \mathtt{clockReset}(\mathbf{C}@[\mathtt{Time} : \mathtt{t}], \mathbf{rtc}\ @\ [\mathtt{TRAN} : \mathtt{k}]);$

$\mathtt{else}\ \mathbf{P}@[\mathtt{Machine} : \mathtt{M_i}, \mathtt{Time} : \mathtt{t}] = \mathbf{P}@[\mathtt{Machine} : \mathtt{M_i}, \mathtt{Time} : \mathtt{prev}\ \mathtt{t}]$

- *clockCheck(clockVar, timeConstraint): checks if the time constraints are satisfied for each clock.*

$\mathtt{if}(\mathtt{oneClockCheck}(\mathtt{first}\ \mathtt{clockVar}, \mathtt{first}\ \mathtt{timeConstraint})\&\&$

$\mathtt{clockCheck}(\mathtt{next}\ \mathtt{clockVar}, \mathtt{next}\ \mathtt{timeConstraint}))$

$\mathtt{then\ true\ else\ false}$

$\mathtt{where}$

$\mathtt{oneClockCheck}(\mathtt{clockvalue}, \mathtt{timec}) =$

$\mathtt{if}(1^{\mathtt{st}}.\mathtt{timec} \leq \mathtt{clockvalue} \leq 2^{\mathtt{nd}}.\mathtt{timec})\ \mathtt{then\ true\ else\ false}$

$\mathtt{end}$

- *clockReset*(*clockVar*, *timeConstraint*):

```
clockReset(clockVar, timeConstraint) = if(first timeConstraint)
    then first clockVar = 0
        fby clockReset(next clockVar, next timeConstraint)
    else clockReset(next clockVar, next timeConstraint)
```

In the GIPSY environment [44] Lucx programs may call external functions written in Java, the target language of our compiler. Hence, *IsAdmissible* and *Reset* functions may also be implemented in Java, based on the above definitions.

6.4 Railroad Crossing Problem

In this section we provide a specification of the generalized railroad crossing problem, an example studied in real-time systems community [30]. This is a real-time reactive system, in which train, gate, and controller objects communicate through messages. There is no data-intensive computation in the model. The version of the problem [30] is given a formal object-oriented design by Muthiayen [43]. Only discrete time is required to model this system.

6.4.1 Problem Statement

Several trains cross a gate controlled by a monitor. Trains may be running on several tracks, and hence cross the gate simultaneously. When a train approaches the gate, it sends a message to the corresponding controller, which then commands the gate to close. When the

last train crossing a gate leaves the crossing, the controller commands the gate to open. The safe operation of the controller depends on the satisfaction of certain timing constraints, so that the gate is closed before the first train enters the crossing, and the gate is opened after the last train leaves the crossing. The following time constraints are assumed [43].

1. [C1] A train enters the crossing within an interval of 2 to 4 time units after having indicated its presence to the controller.

2. [C2] The train informs the controller that it is leaving the crossing within 6 time units of sending the approaching message.

3. [C3] The controller instructs the gate to close within 1 time unit of receiving an approaching message from the first train entering the crossing, and starts monitoring the gate. The controller continues to monitor the closed gate if it receives an approaching message from another train.

4. [C4] The controller instructs the gate to open within 1 time unit of receiving a message from the last train to leave the crossing.

5. [C5] The gate must close within 1 time unit of receiving instructions from the controller.

6. [C6] The gate must open within an interval of 1 to 2 time units of receiving instructions from the controller.

6.4.2 Events and Streams for Problem Specification

In [43], a formal design of the railroad problem is given. It uses ESMs to formalize the behavior of train-gate-controller objects. The formal object-oriented model thus obtained is linked with PVS to formally verify the safety property in the modeled system. The ESMs from [43] are reproduced in Figure 15, Figure 16 and Figure 17.

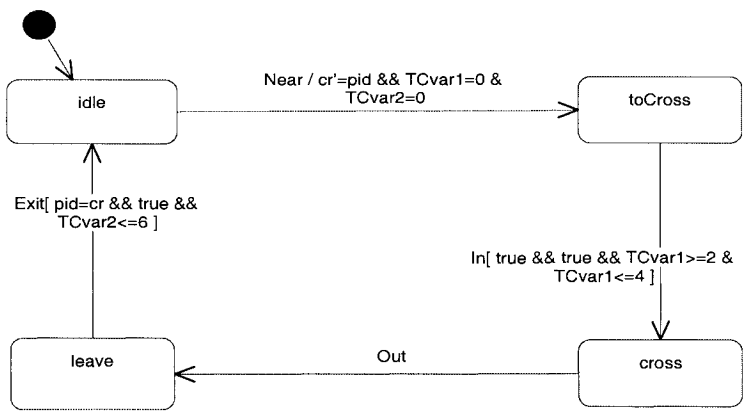

Figure 15: ESM of The Train

We use the approach outlined in Section 6.3 to formally represent the above design in Lucx. The Lucx program for a railroad crossing problem can be written down by simply instantiating the different streams in Section 6.3 with the details shown in the Figure 15, Figure 16 and Figure 17. So, we do not explicitly give the Lucx program. Instead, we give a formal proof in Lucx that the safety property is satisfied in a design which includes the above time constraints. Based on this proof, we claim that the constraints $[C1], \ldots, C[6]$ become verification conditions for the Lucx program. That is, for every instance of the railroad problem, determined by the trains and the times they send *Near*? events, if the

Lucx program satisfies the constraints $[C1], \ldots, C[6]$ then the program satisfies the safety property. As we show in Section 6.4.4 below, the safety property can itself be written down purely in terms of the times of occurrences of observable events in the system. So, we discuss below a Lucx specification of event streams and their constraints.

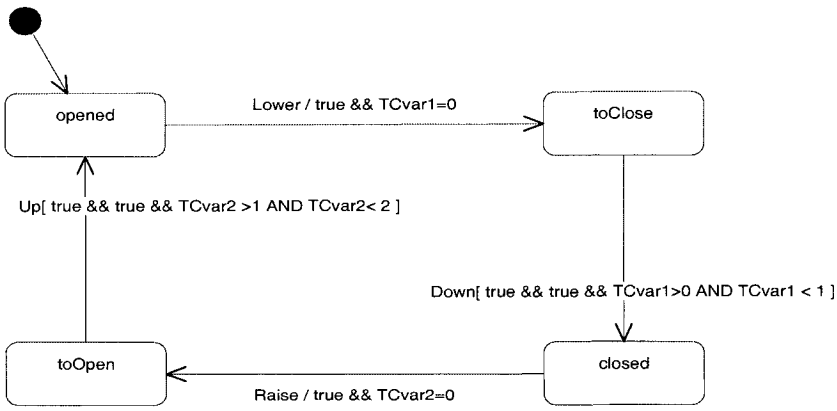

Figure 16: ESM of The Gate

We use the approach outlined in Section 6.3 to formally represent the above design in Lucx. The Lucx program for a railroad crossing problem can be written down by simply instantiating the different streams in Section 6.3 with the details shown in the Figure 15, Figure 16 and Figure 17. So, we do not explicitly give the Lucx program. Instead, we give a formal proof in Lucx that the safety property is satisfied in a design which includes the above time constraints. Based on this proof, we claim that the constraints $[C1], \ldots, C[6]$ become verification conditions for the Lucx program. That is, for every instance of the railroad problem, determined by the trains and the times they send *Near*? events, if the Lucx program satisfies the constraints $[C1], \ldots, C[6]$ then the program satisfies the safety

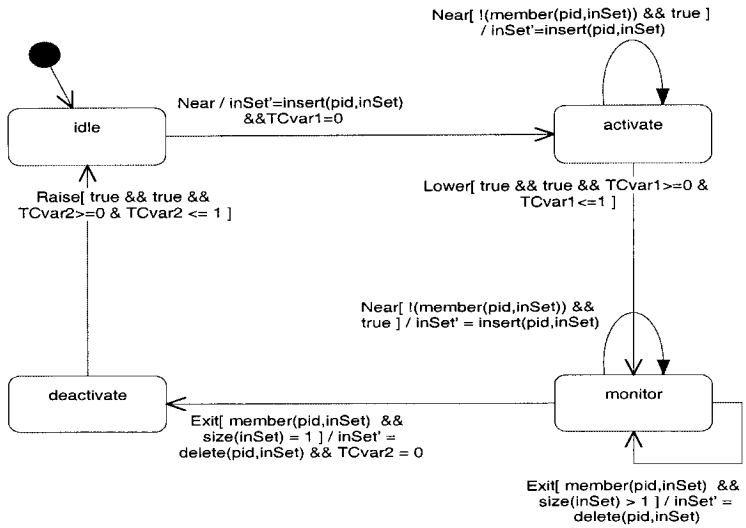

Figure 17: ESM of The Controller

property. As we show in Section 6.4.4 below, the safety property can itself be written down purely in terms of the times of occurrences of observable events in the system. So, we discuss below a Lucx specification of event streams and their constraints.

The events *Lower*! and *Raise*! are sent by the controller to the gate. These events are constrained. The events *Near*? and *Exit*? are received by the controller from a train. They are unconstrained events. The gate closes using the event *Down* and opens using the event *Up*. The events *In* and *Out* are used by trains respectively to indicate that they are inside the crossing and outside the crossing respectively. Each event defines its own stream in Lucx.

Informally, a *period* is the duration in which several trains pass during one session of gate closing. A period starts when the gate starts closing (occurrence of *Down* event) and finishes when the gate opens again (occurrence of *Up* event). Since the duration for events *Down* and *Up* are not part of the specification we assume that the gate opens at the instant

Up occurs. Within a period, several trains may come, and hence the events *Near*, *In*, *Out*, and *Exit* may occur several times. However, within a period, the controller events and gate events occur just once. We represent the events by the following streams:

1. The streams *Lower* and *Raise* are shared representations for the synchronous occurrences of *Lower*!, *Lower*?, and *Raise*!, *Raise*?. Thus, $\overline{Lower_k}$ and $\overline{Raise_k}$ give the times of occurrences of the events Lower and Raise in the $k-th$ period.

2. The streams *Down* and *Up* represent the events Down and Up. That is, in the $k-th$ period, the events Down and Up occur at times $\overline{Down_k}$, and $\overline{Up_k}$.

3. We use a 3-dimensional stream σ to represent the events from trains, with the convention that the events *Near*, *In*, *Out*, and *Exit* are denoted by 1,2,3, and 4 respectively. The justification is that for each train in the $k-th$ period, these events are linearly ordered:

$$TIME(Near)(k) < TIME(In)(k) < TIME(Out)(k) < TIME(Exit)(k).$$

The stream σ has three effective dimensions, say *TRAIN*, *EVE*, and *PER* with tags $\mathbb{N}$ for *TRAIN* and *PER* and the set $\{1,2,3,4\}$ for *EVE*. The evaluation σ @ $[TRAIN : i, EVE : j, PER : k]$, denoted σ_{ijk}, is the time at which the event j occurred in $i-th$ train in the $k-th$ period. For instance, σ_{243} gives the time at which the event *Exit* occurred in the second train in the $3rd-period$. Notice that i increases with the arrival of a new train in the system. The 1-dimensional stream σ@$[TRAIN : i, EVE : 1]$ gives the times of arrivals of the ith train in all periods.

6.4.3 Lucx Specification

Abstracting the problem we come up with the following specification of the event occurrences. For every period, events used by the gate are linearly ordered:

$k \in \mathbb{N}, \overline{Lower_k} < \overline{Down_k} < \overline{Raise_k} < \overline{Up_k}$

Within a period k, the events of each train are linearly ordered:

$\sigma_{i1k} < \sigma i2k < \sigma i3k < \sigma_{i4k}$.

For every period k, the time constraints $C1, \ldots, C6$ can be formally specified in Lucx as follows:

[C1] σ @$[EVE : 1, PER : k] + 2 < \sigma$ @$[EVE : 2, PER : k] < \sigma$ @$[EVE : 1, PER : k] + 4$

[C2] σ @ $[EVE : 1, PER : k] < \sigma$ @ $[EVE : 4, PER : k] < \sigma$ @ $[EVE : 1, PER : k] + 6$

[C3] $\sigma_{11k} < \overline{Lower_k} < \sigma_{11k} + 1$

[C4] $last\_time(\sigma$ @$[EVE : 4, PER : k], \overline{Lower_{k+1}}) < \overline{Raise_k} < last\_time(\sigma$ @$[EVE : 4, PER : k], \overline{Lower_{k+1}}) + 1$

[C5] $\overline{Lower_k} < \overline{Down_k} < \overline{Lower_k} + 1$

[C6] $\overline{Raise_k} + 1 < \overline{Up_k} < \overline{Raise_k} + 2$

The specification is a set of assertions. They become the verification conditions for the Lucx program, once we verify that the satisfy property is a logical consequence of the above assertions.

6.4.4 Verification of Safety Property

Informally, a program that is consistent with the above requirements is *safe*, if in every period k the following property is satisfied by the program: *The gate closes before any*

train is in the crossing and opens only after the last train in the period has left the crossing.

Using our specification formalism, we formally rewrite the safety property as follows:

$$\overline{Down_k} < \sigma_{12k} < last\_time(\sigma \text{ @ } [EVE : 4, PER : k], \overline{Lower_{k+1}}) < \overline{Up_k} \qquad (S)$$

To prove that the safety property, we use the assertions in the specification and $A1$ below as axioms and show that the predicate (S) is a consequence of these axioms:

$$\sigma_{14k} \leq last\_time(\sigma \text{ @ } [EVE : 4, PER : k], \overline{Lower_{k+1}}) \qquad A1$$

The proof steps are as follows for any period k - the axioms used in deriving a step are shown at the end of each step:

[Step 1] $\overline{Down_k} < \overline{Lower_k} + 1$ [C5]

[Step 2] $\overline{Lower_k} + 1 < \sigma_{11k} + 2$ [C3]

[Step 3] $\sigma_{11k} + 2 < \sigma_{12k}$ [C1]

[Step 4] $\overline{Down_k} < \sigma_{12k}$ [Steps 1,2,3]

[Step 5] $\sigma_{12k} < \sigma_{14k}$ [C1,C2]

[Step 6] $\sigma_{14k} \leq last\_time(\sigma \text{ @ } [EVE : 4, PER : k], \overline{Lower_{k+1}})$ [A1]

[Step 7] $\sigma_{12k} < last\_time(\sigma \text{ @ } [EVE : 4, PER : k], \overline{Lower_{k+1}})$ [Steps 5,6]

[Step 8] $last\_time(\sigma \text{ @ } [EVE : 4, PER : k], \overline{Lower_{k+1}}) < \overline{Raise_k}$ [C4]

[Step 9] $last\_time(\sigma \text{ @ } [EVE : 4, PER : k], \overline{Lower_{k+1}}) + 1 < \overline{Raise_k} + 1$ [Step 8]

[Step 10] $\overline{Raise_k} + 1 < \overline{Up_k}$ [C6]

[Step 11] $last\_time(\sigma \text{ @ } [EVE : 4, PER : k], \overline{Lower_{k+1}}) < \overline{Up_k}$ [Steps 9,10]

[Step 12] $\overline{Down_k} < \sigma_{12k} < last\_time(\sigma \text{ @ } [EVE : 4, PER : k], \overline{Lower_{k+1}} < \overline{Up_k}$ [Steps 4,7,11]

We conclude that the ESMs for train, gate, and controller satisfy the constraints $C1, \ldots, C6, A1$

and hence the design satisfies the predicate (S). In order to prove that the Lucx program also satisfies the safety property we have to verify each stated axiom (constraint) in the Lucx program.

6.5 Summary

An implementation of the verified design of the timed system must faithfully conform to the design. The semantic gap between the language used for the formal design and the programming language that implements the design is a barrier for a formal proof of faithfulness of the implementation to the verified design. Our investigative effort in this research is motivated to overcome this barrier. We fixed the formal model of the timed system, and gave a representation of it in Lucx. The operational semantics of the formal model is implemented as Lucx functions. For the railroad crossing problem [30] we gave an abstract specification and stated the safety property in Lucx. Using Lucx representation, we gave a formal proof that the specified solution satisfies the safety property. In fact, this step can be automated in Lucx, because Lucid was demonstrated early on as program verification language [5]. It remains to formally prove in Lucx that the design satisfies the specification. This is one part of future work.

GIPSY [44] is an implementation platform for Lucid. As explained in Chapter 5, an important part of future work is to implement Lucx programs in the same framework. So we claim that there is a semantic continuity in our approach - specification, program development, and implementation are all integrated.

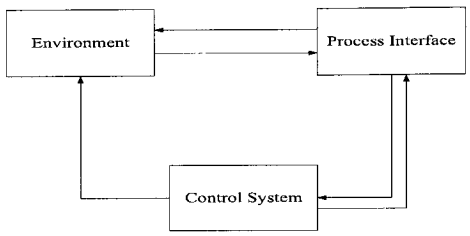

Figure 18: The Abstract Model for Hybrid Systems

The approach discussed in this chapter can be modified and applied to program hybrid systems. The main factors to research are the following:

1. The abstract model of a hybrid system must include the physical properties expressed as a set of mathematical equations. Usually these are algebraic and/or differential equations involving the continuous variables in the system. To be consistent with the OO approach, the mathematical equations can be encapsulated as an active object in the model, as shown in Figure 18. The advantages are:

    - The (system) controller model is separated from the physical model, the mathematical functions that characterize continuous change. This implies that the same controller specification can be reused either as a black box or a glass box for different physical models.

    - GIPSY programs can be written as hybrid programs allowing Java functions to be called by the Lucx part of the program. These Java functions can actually be the implementation of the mathematical functions in different states of a hybrid model. For numerical evaluation of mathematical functions there will

be considerable speed up in performance.

2. Paquet [46] has extended Lucid to handle *tensors* and has applied the extended Lucid (called TensorLucid) to physical problems. There is a need to look at these extensions or similar representations for handling continuous constraints arising in hybrid models.

3. The state transition semantics

$$\frac{s_i \wedge e \in \mathcal{E}(s_i) \wedge var_g(s_i) \wedge tc(t)}{s_i \xrightarrow{e} s_j \wedge var_a(s_j) \wedge rtc(t)}$$

must be changed to reflect the evaluation of predicates in the expressions $var_g(s_i)$ and $var_a(s_j)$ that involve continuous variables.

Chapter 7

Programming Agent Communication in Lucx

In this chapter, we illustrate the suitability of Lucx as an Agent Communication Language(ACL). The ACL that we introduce in this chapter uses context expressions in messages exchanged between communicating agents. KQML [25] *performatives* and FIPA [26] *communicative acts* form the basic structure for composing messages and communication between agents. Performatives as well as communicative acts are fixed in the respective languages, in the sense that they can not be dynamically changed. This restricts the communication ability between agents when they are faced with fast evolution of their environment. In this chapter, we show that context expressions in Lucx are good candidates for representing dynamically changing performatives. We define message structure and semantics of conversations in Lucx.

7.1 Agent Communication Languages

Software agents, according to Chen et al [12], are personalized, continuously running and semi-autonomous, driven by a set of beliefs, desires, and intentions (BDI). Agent technology is being standardized by FIPA [26] with the goal of seamlessly integrating their architectures and languages with various commercial application systems such as *network management*, *E-commerce*, and *mobile computing*. In such applications agents should have capabilities to exchange complex objects, their intentions, shared plans, specific strategies, business and security policies. An ACL must be declarative and have a small number of primitives that are necessary to construct the structures required for achieving the above capabilities. An ACL must support *interoperability* in an agent community while providing the freedom for an agent to hide or reveal its internal details to other agents. The two existing ACLs are *Knowledge Query and Manipulation Language* (KQML) [25] and the FIPA [26] communication language. The FIPA language includes the basic concepts of KQML, yet they have slightly different semantics. We summarize below the major points of contrasts between KQML and FIPA ACL, from the work of Labrou, Finin, and Peng [38].

KQML KQML is a high-level, message-oriented communication language used for protocol specification and information exchange between agents. A KQML message can be considerred as consisting of three layers:

- *The content layer* bears the actual content of the message in the agent's own representation language.

- *The communication layer* decribes the features of lower-level communication parameters, such as the identity of the sender and the receiver.

- *The message layer*, also called performative, illustrates the interactions between agents.

A KQML message from agent Joe querying the price of a share of IBM stock is encoded as "ask-one" performative in Example 31 [38]. Example 32 illustrates STOCK-SERVER's reply to Joe's query [38].

Example 31 *:*

```
(ask − one
sender Joe
: content (PRICE IBM?price)
: receiver STOCK − SERVER
: reply − with IBM − STOCK
: language LPROLOG
: ontology NYSE − TICKS)
```

Example 32 *:*

```
(tell
: sender STOCK − SERVER
: content (PRICEIBM14)
: receiver Joe
: in − reply − to IBM − STOCK
: language LPROLOG
: ontology NYSE − TICKS)
```

KQML also provides a small number of performatives that the agents can use to define meta data. A semantics of KQML in a style similar to Hoare logic is given in [37]. That is, the semantics of KQML performatives is provided in terms of *preconditions*, *postconditions*, and *completion conditions*. Precondition Pre(A) indicates the necessary state for an agent A to send a performative, and Pre(B) is a precondition for the receiver agent B to accept it and successfully process it. Postconditions Post(A) describes the state of the sender agent A after sending the performative successfully. Similarly, the postcondition Post(B) describes the state of the receiver after receiving and processing the message but before a counterutterance is made. A completion condition, i.e. Completion, indicates the final state. Preconditions, postconditions, and completion conditions involve action descriptors,

such as $PROC(A, M)$ and $SENDMSG(A, B, M)$, and describe states of agents in a language of mental attitudes such as belief, knowledge, want, and intention. As an example [38], Figure 19 shows the semantics for *tell* performative. The semantics for *tell* suggests that an agent cannot offer unsolicited information to another agent.

Example 33 :
tell(A, B, X)
Pre(A) : BEL(A, X) $\wedge$ KNOW(A, WANT(B, KNOW(B, S)))
Pre(B) : INT(B, KNOW(B, S))
where S may be any of BEL(B, X), or $\neg$(BEL(B, X)).

Post(A) : KNOW(A, KNOW(B, BEL(A, X)))
Post(B) : KNOW(B, BEL(A, X))

Completion : KNOW(B, BEL(A, X))

Figure 19: KQML semantics for tell.

Building on pre- and postcondition semantics conversation policies have been devised for agent conversation. Conversation policies describe both the sequences of KQML performatives and the constraints and dependencies on the values of the reserved parameters of the performatives involved in the conversations.

KQML has a *predefined* set of *reserved performatives*. It is neither a minimal required set nor a closed set. That is, an agent may use only those primitives that it needs in a communication, and a community of agents may agree either to use the union of the sets of primitives required by each one of them or use some additional performatives with a consensus on the semantics and protocols for using them. In the latter case, it is not clear as to how the agents will construct the additional performatives and how a semantics can be dynamically worked out.

FIPA ACL The syntax of the FIPA ACL resembles KQML, however its semantics is formally given by a quantified multi-modal logic [65]. The communication primitives in FIPA ACL are called *communicative acts* (CA), yet they are the same as KQML primitives [26]. As an example, we give below the communication between an arbitrator agent with two other agents who are in dispute. The behaviour descriptions are given in [7].

1. Agent `a` contacts the arbitrator agent `m` requesting mediation between itself and agent `b`. The agents disagree on the location where an e-mail has to be delivered. Agent `a` wants to send the email to `cs@scu.edu` and agent `b` wishes to send the email to `ee@scu.edu`.

(**request**

:sender (agent-identifier :name a)

:receiver (set (agent-identifier :name m))

:content (action (agent-identifier: name a)

(I (send-email cs@scu.edu)

(arbitration-identifier 5)))

:language FIPA-SL)

2. The arbitrator agent `m` contacts agent `b` for its version of the dispute.

(**inform**

:sender (agent-identifier :name m)

:receiver (set (agent-identifier :name b))

:content

(inform

(send-report (arbitration-identifier 5)))

:language FIPA-SL)

3. Agent m receives agent b's version of dispute.

(**inform**

:sender (agent-identifier :name b)

:receiver (set (agent-identifier :name m))

:content

(action agent-identifier :name b)

(I (send-email ee@scu.edu))

:language FIPA-SL)

4. Agent m analyzes and replies with a suggestion to both the agents.

(**propose**

:sender (agent-identifier :name m)

:receiver (set (agent-identifier :name a))

:content

(arbitration-identifier 5

(suggestion-number 1

(action (send-email cs@scu.edu ee@scu.edu))))

:language FIPA-SL)

5. The suggestion is rejected by one of the agents.

(**reject proposal**

:sender (agent-identifier :name a)

:receiver (set (agent-identifier :name m))

:content

(arbitration-identifier 5

(suggestion-identifier 1))

:language FIPA-SL)

6. Both agents accept the suggestion.

(**accept proposal**

:sender (agent-identifier :name b)

:receiver (set (agent-identifier :name m))

:content

(arbitration-identifier 5

(suggestion-number 1))

:language FIPA-SL)

The semantics of FIPA ACL is given in the formal language SL, which provides the modal operators for beliefs (B), desires (D), intentions (persistent goals PG), and uncertain goals (U). Actions of objects, object descriptions, and propositions can be described in the language. Each formula in SL defines a constraint that the sender of the message must satisfy in order for the sender to conform to the FIPA ACL standard [65].

In order to achieve cooperation and interoperability, both KQML and FIPA ACL need to predefine a set of performatives, which is neither a minimal required set nor a closed one. This creates a big problem for maintaining and extending the agents to face the fast evolution of performatives. However, if we design the communication language from a

higher level and in a more abstract way in which the performatives become *first class objects*, we will be able to create additional performatives as context expressions.

This can be done in Lucx by encapsulating performatives in contexts. Meanwhile operators on contexts can be used to create new performatives from existing performatives. Informally, when an agent A sends a communicative act x to an agent B, we view x as a collection (may be a sequence) of objects, where each object is bound to some description on its interpretation, evaluation criteria, temporal properties, constraints, and any other information that can be encoded in the language. We view this collection as a context. In our approach, the name of a performative is considered as an expression, and the rest of the performative constitute a *context* which can be understood as a *communication context*; each field except the name in the message is a *micro context*. The communication context will be evaluated by the receiver. In some cases, the receiver may combine the communication context with its *local context* to generate a new context.

7.2 Message Structure and Evaluation in Lucx

The syntax of a message in Lucx is $\langle E, E' \rangle$, where E is the message name and E' is a context. The message name in a Communicative Act CA of FIPA ACL or the name of a KQML performative is captured in Lucx by E. In an implementation, E corresponds to a function. The context E' includes all the information that an agent wants to convey in an interaction to another agent. Thus, a query from an agent A to an agent B is of the form $\langle E_A, E'_A \rangle$. A response from agent B to agent A will be of the form $\langle E_B, E'_B \rangle$, where E'_B will

include the reference to the query for which this is a response, in addition to the contexts in which the response should be understood.

The operational semantics in Lucx is the basis for query evaluation. The query from agent A $\langle E_A, E'_A \rangle$ to agent B is evaluated as follows:

- agent B obtains the context $F_B = E'_A \oplus L_B$, where L_B is the local context for B.
- agent B evaluates $E_A @ F_B$
- agent B constructs the new context E'_B that includes the evaluated result and information suggesting the context in which it should be interpreted by agent A, and
- sends the response $\langle E_B, E'_B \rangle$ to agent A.

The query in Example 31 is represented in Lucx as the expression E_A @ E'_A shown in Example 34. The reply in Example 31 is represented in Lucx as the expression E_B @ E'_B shown in Example 35.

Example 34 :

$\mathrm{E_A}$ @ $\mathrm{E'_A}$
where
$\mathrm{E_A} = \text{"ask} - \text{one"};$
$\mathrm{E'_A} = \mathrm{E_1} \oplus \mathrm{E_2} \oplus \mathrm{E_3} \oplus \mathrm{E_4} \oplus \mathrm{E_5} \oplus \mathrm{E_6};$
$\mathrm{E_1} = [\mathtt{sender} : \mathtt{joe}];$
$\mathrm{E_2} = [\mathtt{content} : \mathrm{E_7}];$
$\mathrm{E_3} = [\mathtt{receiver} : \mathtt{STOCK} - \mathtt{SERVER}];$
$\mathrm{E_4} = [\mathtt{reply} - \mathtt{with} : \mathtt{IBM} - \mathtt{STOCK}];$
$\mathrm{E_5} = [\mathtt{language} : \mathtt{LPROLOG}];$
$\mathrm{E_6} = [\mathtt{ontology} : \mathtt{NYSE} - \mathtt{TICKS}];$
$\mathrm{E_7} = [\mathtt{PRICE} : \mathtt{IBM}];$
end

Example 35 :

$\mathrm{E_B}$ @ $\mathrm{E'_B}$
where
$\mathrm{E_B} = \text{"tell"};$
$\mathrm{E'_B} = \mathrm{E'_1} \oplus \mathrm{E'_2} \oplus \mathrm{E'_3} \oplus \mathrm{E'_4} \oplus \mathrm{E'_5} \oplus \mathrm{E'_6};$
$\mathrm{E'_1} = [\mathtt{sender} : \mathtt{STOCK} - \mathtt{SERVER}];$
$\mathrm{E'_2} = [\mathtt{content} : \mathrm{E'_7}];$
$\mathrm{E'_3} = [\mathtt{receiver} : \mathtt{joe}];$
$\mathrm{E'_4} = [\mathtt{in} - \mathtt{reply} - \mathtt{to} : \mathtt{IBM} - \mathtt{STOCK}];$
$\mathrm{E'_5} = [\mathtt{language} : \mathtt{LPROLOG}];$
$\mathrm{E'_6} = [\mathtt{ontology} : \mathtt{NYSE} - \mathtt{TICKS}];$
$\mathrm{E'_7} = [\mathtt{PRICE} : 14];$
end

The implementation will assure that the local context of B is sufficient to evaluate the query and respond to A within an acceptable time delay. This is an important issue because we

want the agents to be reactive (responds within acceptable time limits) while the eduction is allowed to continue. The choice operator helps in achieving such a goal. For example, the query in Example 36 gives the receiver, depending on its local context, choose either LPROLOG or STANDARD_PROLOG to ensure timeliness. The fields in the performative in Example 31 can not be dynamically changed in either FIPA or KQML. In our language, we form the context expression $E''_A = E_A \uparrow \{language\} \oplus [language : Java]$ to dynamically replace the language requirement and construct a new query. The meaning of the examples shown in this section can be clearly understood from the semantics of the context calculus presented in the Section 2.1.

Example 36 *:*

$E_A @ E'_A$
where
$E = "ask - one";$
$E'_A = E_1 \oplus E_2 \oplus E_3 \oplus E_4 \oplus E_5 \mid E_6 \oplus E_7;$
$E_1 = [sender : joe];$
$E_2 = [content : E_8];$
$E_3 = [receiver : STOCK - SERVER];$
$E_4 = [reply - with : IBM - STOCK];$
$E_5 = [language : LPROLOG];$
$E_6 = [language : STANDARD\_PROLOG];$
$E_7 = [ontology : NYSE - TICKS];$
$E_8 = [PRICE : IBM];$
end

7.3 Semantics of Conversation

We reviewed the semantics of KQML and FIPA ACL briefly in Section 7.1. Our approach to semantics is different from these two approaches. We specify the pre conditions and

post conditions for each query using contexts having four dimensions B(Belief), K(Know), W(Want), and I(Intention). The tags along the dimension B are predicates, and the respective tags along the dimensions K, W, and I are logic expressions. That is, each performative is bound to a context $c = [B : i_1, K : i_2, W : i_3, I : i_4]$ over the dimensions B, K, W, I, where c is suggested as the precondition to act upon the performative. If a dimension is not specified in c then it is equivalent to a "don't care" condition.

We define a *dialogue* initiated by agent X with agent Y as a pair $\langle \alpha, \beta \rangle$, where α is sent from X to Y and β is the response from Y to X. The agent X constructs the special context $Pre_Y(M_X)$ for message M_X and sends the pair $\alpha = (Pre_Y(M_X), M_X)$ to Y. The agent Y evaluates its local state at $Pre_Y(M_X)$. The result of evaluation is a tuple $\langle b_1, k_1, w_1, i_1 \rangle$. The tuple corresponding to an empty Pre_Y is $\langle NONE, NONE, NONE, NONE \rangle$, interpreted as *true*. That is, the agent X has not indicated any preference as to when agent Y should evaluate the performative M_X. If at least one component of the tuple is not *NULL*, then the special context $Pre_Y(M_X)$ is said to be *satisfied* at some local state of Y. If all components of the tuple are *NULL* the context is not satisfiable at any local state of Y. If the outcome of evaluation is either *true* or *satisfied*, the agent Y will act upon the performative M_X. For instance, in Example 34, the agent X constructs the special context $E_7 = [I : i_4]$ and attaches it to the performative in a conversation with Y. The agent Y evaluates its local repository on its belief, desire, want and intentions, at the context E_7. The result of evaluation is the tuple $\langle NONE, NONE, NONE, PROC(Y, M) \rangle$, implying that the agent Y has the intention to process the message M.

The semantics of a dialogue initiated by X with Y is as follows:

1. Agent X creates a special context $Pre_X(M_X)$, the weakest precondition that enables to send a message to agent Y. When $Pre_X(M_X)$ is true in its local state, it constructs $Pre_Y(M_X)$, a precondition based on the information that it shares with agent Y.

2. Agent X sends $\alpha = (Pre_Y(M_X) \oplus M_X)$ to Y.

3. Agent Y disassembles it into the message part M and the special context $Pre_Y(M_X)$. This is done by computing $Pre_Y(M_X) = \alpha \downarrow \{B, K, W, I\}$, and $M_X = \alpha \uparrow \{B, K, W, I\}$.

4. Agent Y evaluates its local state at $Pre_Y(M_X)$.

5. If *satisfied* it does the following:

   (a) creates the post condition $Post_Y(M_X)$ that satisfies the task completion;

   (b) acts upon the message M_X;

   (c) composes the reply as a performative M_Y;

   (d) creates $Post_X(M_Y)$, the special context in which agent X should evaluate M_Y;

   (e) composes $\beta = (Post_X(M_Y), M_Y)$;

   (f) sends β to agent X.

6. If NOT *satisfied*, more than one semantics can be given:

   - [1.] Agent Y responds immediately to X: composes an "unable to act" performative, constructs the special context $\langle NONE, NONE, NONE, NONE \rangle$, and sends the pair to agent X.

- [2.] Agent Y delays the evaluation of M_X until the instant when the special context $Pre_Y(M_X)$ is either satisfied or not satisfied in its local state.
- [3.] Agent Y abandons the message if the special context $Pre_Y(M_X)$ is either satisfied or not satisfied within a certain amount of time.

The first semantics is preferred to ensure the deterministic progress in the system. Under the first semantics of dialogue we can define the semantics of a *conversation*. A sequence $\langle(\alpha_1, \beta_1);\ \ldots;\ (\alpha_k, \beta_k), \ldots, \rangle$ of dialogues is a conversation if for every i, $i \geq 1$, there exists at least one local state of X in which the postcondition $Post_X(M_Y)$ in β_i is satisfied. In the language, a conversation can be represented as *tuple streams*, where each tuple is a pair of contexts.

7.4 Summary

Lucx, as an ACL, has a number of advantages:

- In KQML and FIPA, performatives not defined in the language, can be agreed upon by the community of agents involved in a collaboration. That is, interoperability is possible. However, performatives are only static and not first class objects in the language. As a consequence, performatives can not be changed dynamically, nor can they be used as a vehicle to communicate local state information of agents. For a large application, this necessarily demands the construction of all the performatives required for communication in advance. This can be avoided in Lucx. In addition, we can define functions on contexts and they can be used as parameters in programs.

Thus, we have enhanced both *interoperability* and *flexibility* in agent communication.

- Lucx is declarative and has a formal semantics.

- *Multiple formats of communication* can be supported since intensional programming language deals with any kind of ordinary data type. Even the multimedia streams between agents become feasible.

Chapter 8

Constraint Programming in Lucx

Constraint programming has been successfully applied in several domains including operations research, business application, code optimisation, and molecular biology. Constraints may be regarded as a set of requirements to solve a problem, or a set of properties that a solution to the problem must satisfy. A constraint programming language must provide abstractions to represent constraints, a logic for reasoning with them, and constructs to express computations in the constraint solver. Constraint programming languages that are in use can be classified into three kinds:

- *[Modeling language]* They are mainly used for modeling and solving combinatorial and optimization problems. Examples include AMPL (A Modeling Language for Mathematical Programming) [13], GAMS (General Algebraic Modeling System), and OPL (Optimization Programming Language) [31].

- *[Imperative programming]* A constraint satisfaction problem is modeled as C++ classes. Constraint variables become objects manipulated by functions belonging

to the given class. The ILOG solver [33], currently being developed, is an example.

- *[Logic programming language]* Constraint variables are modeled as logical variables. The *unification* mechanism of logic programming is replaced by constraint solving and *substitution* is replaced by the constraint store [13].

In this chapter we discuss the suitability of *Lucx* as a constraint programming language by modeling a constraint as a set of *contexts*. The motivation comes from the merits of Lucx shown as follows:

- Lucx has the expressive power to represent and manipulate contexts, and hence constraints, in a declarative manner.
- Constraint reductions can be done using context calculus.
- Domain specific constraint solvers for optimization purposes can be implemented as Java programs in GIPSY, a platform under development introduced in Chapter 5.
- CSPs (Constraint Solving Problem) can be dynamically composed and decomposed in Lucx. Consequently the language is suitable as a Constraint Choice Language(CCL) for coordinated problem solving in a multi-agent system.

8.1 Contexts and Constraint Programming

In this section, we discuss the representation of CSP in Lucx, and give an example.

A *constraint CT* on a finite sequence $X := x_1, x_2, \ldots, x_k$, $0 < k \leq n$, is a subset of $D_1 \times D_2 \ldots \times D_k$, where $D_1, \ldots, D_n$ are value domains and $x_1 \in D_1, x_2 \in D_2, \ldots, x_k \in D_k$.

In this chapter we restrict to discrete domains. Each construct of the form $x_i \in D_i$ is called a *domain expression*. Formally, a CSP is represented as a tuple $\langle \mathcal{CT};\ \mathcal{DE} \rangle$, where $\mathcal{DE}$ is the domain expression $x_1 \in D_1, \ldots, x_n \in D_n$ and $\mathcal{CT}$ is a finite set of constraints, each on a subsequence of $x_1, \ldots, x_n$. Intuitively, a solution to a CSP is a sequence of legal values for all of its variables such that all its constraints are satisfied. More precisely, a n-tuple $\langle v_1, \ldots, v_n \rangle \in D_1 \times \ \ldots \ \times D_n$ is a solution to a constraint $CT \in \mathcal{CT}$ on the variables $x_{i1}, \ldots, x_{iw}$, $1 < w \leq n$, if the substitution $[v_{i1}/x_{i1}, \ldots, v_{iw}/x_{iw}]$ satisfies the constraint CT. An n-tuple $\langle v_1, \ldots, v_n \rangle \in D_1 \times \ldots \times D_n$ is a *solution* to $\langle\ \mathcal{CT};\ \mathcal{DE}\ \rangle$ if it satisfies every constraint $CT \in \mathcal{CT}$. If a CSP has a solution, it is called *consistent* otherwise it is called *inconsistent*. Hence, the complete set of solutions for a consistent CSP is

$$\mathcal{R}_n = \{\langle\ v_1, v_2, \ldots, v_n\ \rangle \mid \langle\ v_1, v_2, \ldots, v_n\ \rangle \textit{ is a solution to CSP}\}.$$

8.1.1 CSP Representation in Lucx

Each constraint $CT \in \mathcal{CT}$ can be represented by a *Box* notation, provided we introduce $\Delta(CT)$ as the set of dimensions $\{X_i \mid x_i \in D_i\}$ and $P(CT)$ is the constraint itself. For each domain expression DE associated with CT, we construct *Box*, where $\Delta(DE) = \Delta(CT)$ and $P(DE)$ is the DE itself. For example, corresponding to $CT = x \leq u$, $DE = (x, u \in \mathbb{N})$, we introduce the dimensions X corresponding to the variable x and U corresponding to the variable u. The corresponding *Box* expressions are B_1 and B_2, where $\Delta(B_1) = \Delta = \{X, U\}$, $P(B_1) = CT$, $\Delta(B_2) = \Delta = \{X, U\}$, $P(B_2) = DE$. It is a straightforward exercise to formalize this approach for all constraints in a CSP. We combine the *Box* expressions using the *Box* operators, thus getting *Box* expression for $\mathcal{CT}$ and $\mathcal{DE}$: Let $CT_1, CT_2, \ldots, CT_m \in \mathcal{CT}$,

and $X_1, \ldots, X_n$ be the dimensions introduced, X_k corresponding to the domain variable x_k, $0 < k \leq n$. Corresponding to the constraint $CT_i \in \mathcal{CT}$ on the variables $x_{i1}, \ldots, x_{iw}$, we define the *Box*es B_i and B'_i, where $\Delta(B_i) = \Delta_i$, $P(B_i) = CT_i$, $\Delta(B'_i) = \Delta_i$, $P(B'_i) = DE_i$, where $\Delta_i = \{X_{i1}, \ldots, X_{iw}\}$, and $DE_i = x_{i1} \in D_{i1} \wedge \ldots \wedge x_{iw} \in D_{iw}$. The representation of the CSP $\langle \mathcal{CT};\ \mathcal{DE} \rangle$, in Lucx is the pair $\langle E;\ B \rangle$ where

$E = B'_1 \boxtimes \ldots \boxtimes B'_m$

$B = B_1 \boxtimes \ldots \boxtimes B_m$

The justification for this representation is that the dimensions involved in all the *Box* expressions have to be taken together in obtaining the solution to CSP. We can interpret E as a function, defined as $E(x_1, x_2, \ldots, x_n) = \langle x_1, x_2, \ldots, x_n \rangle, x_1 \in D_1, \ldots, x_n \in D_n$.

From the above discussion and the semantics of evaluation of expressions in Lucx, we derive the following conclusions:

- the solution space to the CSP $\langle \mathcal{CT};\ \mathcal{DE} \rangle$ is $D_1 \times \ldots \times D_n$;
- the domain of the Lucx expression E is $D_1 \times \ldots \times D_n$;
- since every solution $\langle v_1, \ldots, v_n \rangle$ to the CSP satisfies all the constraints $CT \in \mathcal{CT}$, and B represents $\mathcal{CT}$ in Lucx, there exists some context $c \in B$, where $E@c = \langle v_1, \ldots, v_n \rangle$,
- the complete solution set is $E@B = \{E@c \mid c \in B\}$, which is equal to $\mathcal{R}_n$.

Example 37 illustrates the above procedure for a CSP.

Example 37 *Problem Description: Consider the sequence of four variables x,y,z,u ranging*

over natural numbers and the following three constraints on them:

$$x^3 + y^3 + z^3 + u^3 = 100 \tag{1}$$

$$x < u \tag{2}$$

$$x + y = z \tag{3}$$

With respect to the constraint (1), we construct $\Delta_1 = \{X, Y, Z, U\}$, where X, Y, Z, U are respectively the dimensions associated with the variables x, y, z, and u. The predicate p_1 is $x^3 + y^3 + z^3 + u^3 = 100$. Since all the domains are $\mathbb{N}$, the corresponding domain expression is the predicate formula $p_1' = x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge z \in \mathbb{N} \wedge u \in \mathbb{N}$. Hence, corresponding to (1) we get the *Box*es shown in (4), and (7). Similarly, we construct the *Box*es for the other constraints.

$$CT_1 : B_1 = Box[X, Y, Z, U \mid X^3 + Y^3 + Z^3 + U^3 = 100$$

$$\wedge\ 0 \leq X \leq 4 \wedge 0 \leq Y \leq 4 \wedge 0 \leq Z \leq 4 \wedge 0 \leq U \leq 4] \tag{4}$$

$$CT_2 : B_2 = Box[X, U \mid\ X < U \wedge 0 \leq X \leq 4 \wedge 0 \leq U \leq 4] \tag{5}$$

$$CT_3 : B_3 = Box[X, Y, Z \mid\ X + Y = Z \wedge 0 \leq X \leq 4 \wedge 0 \leq Y \leq 4 \wedge 0 \leq Z \leq 4] \tag{6}$$

$$DE_1 : D_1 = Box[X, Y, Z, U \mid X, Y, Z, U \in \mathbb{N}] \tag{7}$$

$$DE_2 : D_2 = Box[X, U \mid\ X, U \in \mathbb{N}] \tag{8}$$

$$DE_3 : D_3 = Box[X, Y, Z \mid\ X, Y, Z \in \mathbb{N}] \tag{9}$$

For the given constraint CT the constructed expression in Lucx is

$$B = B_1 \boxtimes B_2 \boxtimes B_3, \textit{and} \tag{10}$$

corresponding to the domain expression DE the constructed expression in Lucx is

$$E = D_1 \boxtimes D_2 \boxtimes D_3 = [X, Y, Z, U \mid X, Y, Z, U \in \mathbb{N}]. \tag{11}$$

The CSP representation in Lucx is the pair of expressions:

$$\langle E;\ B \rangle \tag{12}$$

The complete solution to the CSP is given by

$$E@B = \{E@c \mid c \in B\} \tag{13}$$

8.1.2 Solving Constraint Problem in Lucx

Once the CSP is constructed in Lucx, solving the CSP in our approach is achieved through evaluating E at $Box\ B : E\ @\ B\ =\ \{E\ @\ c \mid c\ \in\ B\}$. Example 38 shows the Lucx program for solving the problem in Example 37. Because of the nature of dataflow language, Lucx program can be represented as a dataflow network.

Example 38 : The expression E has four dimensions (X, Y, Z, U) with tag values in $\mathbb{N}$. In the dataflow network, each domain of the expression E is expressed as an *input* node in the graph. Each constraint, a *Box* expression, is also expressed as an *input* node. The operator $\boxtimes$ is expressed as a *function* node. The solution of CSP is expressed as an *output* node. Figure 20 shows the dataflow network corresponding to Lucx program describing this CSP. Consequently, the output from the dataflow network in Figure 20 is the set of solutions for the constraint satisfaction problem in the lexicographic order defined as:

$\langle v_1, v_2, v_3, v_4 \rangle \leq \langle v'_1, v'_2, v'_3, v'_4 \rangle$

if $v_1 < v'_1$

or $v_1 = v'_1 \wedge v_2 < v'_2$

or $v_1 = v'_1 \wedge v_2 = v'_2 \wedge v_3 < v'_3$

or $v_1 = v'_1 \wedge v_2 = v'_2 \wedge v_3 = v'_3 \wedge v_4 < v'_4$

Figure 21 shows the Lucx program for the dataflow network. Notice that the *merge* function is necessary to prune the backtrack solution tree and produce only distinct solutions.

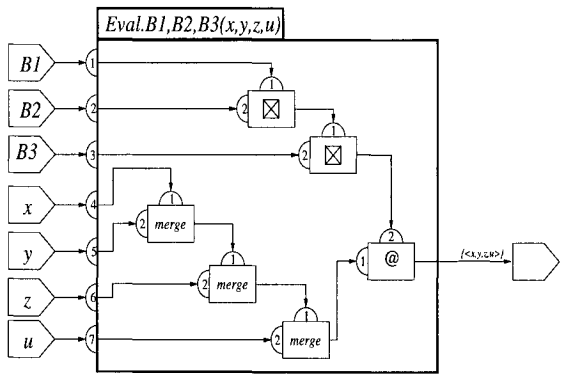

Figure 20: Dataflow Network for Example 52

```
Eval.B1, B2, B3 (x', y', z', u') = N
where
  N = merge ( merge( merge(x, y), z), u)
      @ B1 ⊠ B2 ⊠ B3;
  where
    merge(x, y) = if (x <= y) then x else y;
    B1 = Box [ X, Y, Z, U |  X^3 + Y^3 + Z^3 + U^3 = 100];
    B2 = Box [ X, U |  X < U];
    B3 = Box [ X, Y, Z |  X + Y = Z];
  end
end
```

Figure 21: Lucx Program for Example 52

In Example 38, evaluation takes place by generating successive demands for the appropriate values of `x',y',z',u'` in different contexts, until the final computation can be effected.

Using Theorem 2 (Page 58), Example 39 shows an alternate method to solve the CSP in Example 37.

Example 39 : We need to compute the righthand side expression in the following equation:

$$E@B_1 \boxtimes B_2 \boxtimes B_3 = E@B_1 \bowtie E@B_2 \bowtie E@B_3.$$

The programming steps are as follows:

- *Compute* E_2

  $$B_2 = \{[X:1,U:4],[X:2,U:4],[X:3,U:4],[X:1,U:3],[X:2,U:3],\\ [X:1,U:2]\}$$

  The solution set to $E@B_2$ is the relational expression:

  $$E_2 = \{\langle 1,y,z,4\rangle,\langle 2,y,z,4\rangle,\langle 3,y,z,4\rangle,\langle 1,y,z,3\rangle,\langle 2,y,z,3\rangle,\langle 1,y,z,2\rangle\}$$

  where each tuple is to be regarded as an expression over the domain $D_2 \times D_3$.

- *Compute* E_3

  $$B_3 = \{[X:1,Y:1,Z:2],[X:1,Y:2,Z:3],[X:2,Y:1,Z:3],[X:1,Y:3,Z:4],\\ [X:2,Y:2,Z:4],[X:3,Y:1,Z:4]\}$$

  The solution set to $E@B_3$ gives the relational expression:

  $$E_3 = \{\langle 1,1,2,u\rangle,\langle 1,2,3,u\rangle,\langle 2,1,3,u\rangle,\langle 1,3,4,u\rangle,\langle 2,2,4,u\rangle,\langle 3,1,4,u\rangle\}$$

  where each tuple is to be regarded as an expression over the domain D_4.

- *Compute Join $E_2 \bowtie E_3$ (Unify)*

  We use the result $E@B_2 \bowtie E@B_3 = E_2@B_3 \cap E_3@B_2$, and get the relation E_4:

  $$E_4 = \{\langle 1,1,2,4\rangle, \langle 1,2,3,4\rangle, \langle 1,3,4,4\rangle, \langle 2,1,3,4\rangle, \langle 2,2,4,4\rangle, \langle 3,1,4,4\rangle, \langle 1,1,2,3\rangle, \langle 1,2,3,3\rangle, \langle 1,3,4,3\rangle, \langle 2,1,3,3\rangle, \langle 2,2,4,3\rangle, \langle 1,1,2,2\rangle, \langle 1,2,3,2\rangle, \langle 1,3,4,2\rangle\}$$

  Observe that this calculation is equivalent to finding extensions to the IRs E_2 and E_3 so that their extensions have a non-empty intersection. Viewed differently, this amounts to finding *substitutions* to the unknowns in the IRs so that the resulting expressions are equal, which is a *unification* process.

- *Compute Join with E_1: $E_1 \bowtie E_2 \bowtie E_3$*

  Since E_4 is a constant relation, its evaluation over B_1 must produce another constant relation E_5 which is a sub-relation of E_4. The relation E_5 must equal $E_1 \bowtie E_4$, $E_1 = E@B_1$. It is easy to check that $E_5 = \{\langle 1,2,3,4\rangle, \langle 2,1,3,4\rangle\}$. The tuples in E_5 satisfy all the three constraints B_1, B_2, and B_3. Hence the complete solution to the CSP is $E_5 = \{\langle 1,2,3,4\rangle, \langle 2,1,3,4\rangle, \langle 1,3,4,2\rangle\}$.

8.2 Lucx as a Constraint Choice Language (CCL)

Many problems that arise in day-to-day life requires defining and making choices. Choices are usually expressed as constraints. Hence, solving these problems require natural representation of choices and efficient algorithms for solving constraints. Since, the choices are not in general fully known, they are not static. When problem solving takes place among anonymous participants, the world of possibilities keep changing until reaching a

consensus. That is, the domains associated with problem description, the domain values, and constraints may all dynamically change. Consequently, both the language of representation and the solution methods should be designed to handle such a dynamic evolution of constraints. Multi-agent systems is an emerging technology for solving constraint problems in a distributed environment. Agents require a communication language (ACL) to communicate with each other and a content language to express information on constraints, which is encapsulated as a field within *performatives* of ACL. FIPA Constraint Choice Language(CCL) is one such content language [27]. CCL is designed to support agent problem solving by providing explicit representations of choices and choice problems. In the previous chapter, we have shown the suitability of Lucx as an agent communication language. In this section, we show the use of Lucx as a CCL for coordinated problem solving in a multi-agent system.

We skip details on agent definitions and their collaboration, and illustrate how Lucx can be used by the agent community. It is sufficient if we agree that a software agent, hereafter referred only as agent, has sufficient knowledge and resources to perform actions attributed to it.

According to the requirements stated in [66], it should be possible in CCL to represent the sets of choices to be made, define operations that can be performed on the choices, declaratively state the relationships among choices, and introduce simple propositional statements. Lucx satisfies these requirements:

- Typically, the choice problem can be formulated as a CSP. In Lucx the CSP is representable as a pair of *Box* expressions, and the CSP solution is representable as a

stream of tuples;

- We have provided a context calculus for manipulating context expressions, and *Box* expression evaluation rules for reducing *Box* expressions. Consequently, the CCL *actions* for adding constraints, retracting domain values, and dynamically evaluating constraints can be programmed in Lucx as stream modifiers. That is, Lucx has the constructs and the mechanism for expressing and implementing actions.

In addition, Lucx has a formal semantics, whereas many of the CCLs that are in practice today [66] do not have strict semantics.

Below we show how Lucx can be used for agent-based problem solving in the following four aspects that are normally attributed to a CCL.

1. *Modeling*: Choice Problem is modeled as CSP in Lucx, say by an agent A;

2. *Information Gathering*: Agent A either sends the whole CSP to several other agents or has the knowledge to decompose the CSP into several sub-CSPs, where each sub-CSP is solvable by an agent; after decomposition it sends to those agents and gets their feedback; Lucx, as ACL, can be used here;

3. *Information Fusion*: Agent A incorporates the feedbacks from other agents using context calculus and *Box* operations.

4. *Problem Solving*: Agent A may run simple problem solving algorithm such as the General CSP solver, or send the CSP components to problem solving agents, get their solutions, and unify it.

Below, we give an overview of how Lucx can be used in these steps for the travel planning example taken from [66].

8.2.1 Travel Planning Example (TPE)

A general, but a partial, description of the problem is as follows: *Caroline would like to meet Liz in London for one of exhibition preview receptions at the Tate Gallery. These will be held at the beginning of October. Both Liz and Caroline have other appointments around that time, and will need to travel to London from their homes in Paris and New York.*

We suppose that there is an agent-based system making choices on when Liz and Caroline meet. Several agents assist each participant: a Personal Travel Assistant Agent (PTA) will communicate with Hotel Broker Agent(HBA), Air Travel Agent (ATA), and Diary Agent(DA). That is, the PTA for Caroline (PTAc) will get the hotel information from HBA, flight information from ATA, and meeting time from DA. After collecting and combining the information, it sends the information to Problem Solving Agent(PSA). The PSA will also receive the collected information from PTA for Liz(PTAl). The PSA computes the final solution and sends the solution to PTAs. As we remarked, the agents themselves are not important, only their usage of Lucx is important.

8.2.2 Problem Modeling

According to [66], a choice problem can be represented as a CSP. So we first define a choice problem as a CSP in Lucx as follows:

1. Each choice to be made is modeled as a *dimension* X_i, so $X = \{X_1, X_2, \ldots, X_m\}$ is

the set of m choices which need to be made to figure out the solution to the CSP.

2. For each choice, there is a domain which is the collection of all available options for the choice. So for each choice X_i, $1 \leq i \leq m$, D_{X_i} indicates the domain for X_i. Consequently, $Box_i = Box[X_i \mid X_i \in D_{X_i}]$ is constructed as a domain expression DE_i. All the domain expressions DEs consist of the domain expression E.

3. Constraints are relationships between choice which express valid or invalid combination. So for each constraint in the problem, CT is constructed as a Box shown in Section 8.1.1. Consequently, all the constraints CTs consist of the constraint set B.

4. The pair $\langle E;\ B \rangle$ is the constructed CSP in Lucx.

From the description of TPE, the choices corresponding domains and constraints are identified. For simplicity, we model only the following choices: Hotel information for Caroline(H_c), Hotel Information for Liz(H_l), Flight Information from NewYork to London for Caroline(F_{fromc}), Flight Information from Paris to London for Liz(F_{froml}), Flight Information from London to NewYork for Caroline(F_{toc}), Flight Information from London to Paris for Liz(F_{tol}), Meeting Time for Caroline(T_c) and for Liz(T_l). Thus the initial domains for the choices are as follows:

- D_{H_c} includes all the preferred hotels in London from Oct.1 to Oct.10 for Caroline to stay; For example, Caroline may like to stay at "Marriott", "Hilton", or "Sheraton". In this case, D_{H_c} is $\{Marriott, Hilton, Sheraton\}$.

- D_{H_l} includes all the preferred hotels in London from Oct.1 to Oct.10 for Liz;

- $D_{F_{fromc}}$ includes all the preferred flights from NewYork to London from Oct.1 to Oct.10 for Caroline;

- $D_{F_{froml}}$ includes all the preferred flights from Paris to London from Oct.1 to Oct.10 for Liz;

- D_{T_c} includes all the working time from Oct.1 to Oct.10 for Caroline;

- D_{T_l} includes all the working time from Oct.1 to Oct.10 for Liz;

- $D_{F_{toc}}$ includes all the preferred flights from London to NewYork on Oct.11 for Caroline;

- $D_{F_{tol}}$ includes all the preferred flights from London to Paris on Oct.11 for Liz;

The constraints which express valid combinations are time constraints in this example. In Lucx, any part of the information can be obtained partially by applying the context operator *projection* $\downarrow$. For example, the flight information for Caroline F_{fromc} can be written as an aggregation of micro-contexts $[T_{fl} : 10am]$ (departure time), $[T_{fa} : 13pm]$ (the arrival time), and $[F_n : AC32]$ (the flight number). That is,

$$F_{fromc} = [T_{fl} : 10am, T_{fa} : 13pm, F_n : AC32]$$

The different times related to Caroline's schedule are

- T_{1c} : the arrival time at London for Caroline;

- T_{2c} : the time at which Caroline begins her stay at the Hotel;

- T_{3c} : the time at which Caroline meets Liz;

- T_{4c} : the time when Caroline leaves the Hotel;

- T_{5c} : the departure time from London for Caroline;

The time constraints for both Caroline's and Liz's schedule are the following predicates:

$T_{1c} \leq T_{2c} \leq T_{3c} \leq T_{4c} \leq T_{5c}$ [For Caroline]

$T_{1l} \leq T_{2l} \leq T_{3l} \leq T_{4l} \leq T_{5l}$ [For Liz]

$T_{3c} = T_{3l}$ [Both for Caroline and Liz]

Box expressions for domain expressions and constraints are next computed. For example, $DE_1 = B'_1 = Box[H_c \mid H_c \in D_{H_c}]$; $DE_2 = B'_2 = Box[F_{fromc} \mid F_{fromc} \in D_{F_{fromc}}]$. The domain expressions are combined to get the domain expression E_{TPE}. Each constraint can be constructed as a *Box* expression. For example, $B_1 = Box[T_{1c}, T_{2c}, T_{3c}, T_{4c}, T_{5c} \mid T_{1c} \leq T_{2c} \wedge T_{2c} \leq T_{3c} \wedge T_{3c} \leq T_{4c} \wedge T_{4c} \leq T_{5c}]$, $B_2 = Box[T_{1l}, T_{2l}, T_{3l}, T_{4l}, T_{5l} \mid T_{1l} \leq T_{2l} \wedge T_{2l} \leq T_{3l} \wedge T_{3l} \leq T_{4l} \wedge T_{4l} \leq T_{5l}]$, $B_3 = Box[T_{3c}, T_{3l} \mid T_{3c} = T_{3l}]$ represent the time constraints respectively for Caroline, Liz, and both Caroline and Liz. From these representations we construct the constraint $B_{TPE} = B_1 \boxtimes B_2 \boxtimes B_3$. Thus the pair $\langle E_{TPE};\ B_{TPE} \rangle$ is constructed in Lucx as the CSP for TPE.

8.2.3 Information Gathering

After modeling the problem, PTAc communicates with other agents to *refine* the CSP model. The initial CSP representation may be decomposed into three parts, one corresponding to hotel choices, one corresponding to air travel, and another corresponding to meeting choices. In Lucx, such a decomposition is achieved by using projection ($\downarrow$) and/or

hiding ($\uparrow$) operators on *Box* expressions. Each sub-CSP may be sent to separate experts for requesting additional information to reduce the domain for each choice. In Lucx, agent communication is modeled as performatives. The content field for each performative is also represented as a context expression. Program in Figure 22 is a message from agent PTAc to agent HBA asking for hotel information for Caroline, and Program in Figure 23 is a reply from agent HBA to agent PTAc. The query from PTAc to HBA is evaluated as the expression E @ E'_A in Figure 22.

```
E @ E'_A
where
    E = "ask − one";
    E'_A = E_1 ⊕ E_2 ⊕ E_3 ⊕ E_4 ⊕ E_5;
    E_1 = [sender : PTAc];
    E_2 = [content : B'_1];
      where
      B'_1 = Box[H_c | H_c ∈ {Marriott, Hilton, Sheraton}];
      end
    E_3 = [receiver : HBA];
    E_4 = [reply − with : Hotel − Infor];
    E_5 = [language : Lucx];
end
```

Figure 22: Agent Communication Performative: Ask

The reply from HBA to PTAc is the expression E' @ E''_T, $E''_T = E'_1 \oplus E'_2 \oplus E'_3 \oplus E'_4 \oplus E'_5$ in Figure 23. (Suppose that the available hotels for Caroline are only "Marriott", and "Hilton".)

In addition, the specialized agents (HBA etc;) may also add constraints for the CSP, which can be easily done by using *Box* operator $\boxtimes$ in the performative. For example, HBA may ask that the guests should check in after 14pm. So the reply from HBA to PTAc in Figure 23 is changed as shown in Figure 24.

$E' \ @ \ E'_T$
where
$E' = \text{"tell"};$
$E''_T = E'_1 \oplus E'_2 \oplus E'_3 \oplus E'_4 \oplus E'_5;$
$E'_1 = [\text{sender} : \mathsf{HBA}];$
$E'_2 = [\text{content} : B''_1];$
where
$B''_1 = \text{Box}[H_c \mid H_c \in \{\text{Marriott}, \text{Hilton}\}];$
end
$E'_3 = [\text{receiver} : \mathsf{PTAc}];$
$E'_4 = [\text{in} - \text{reply} - \text{to} : \text{Hotel} - \text{Infor}];$
$E'_5 = [\text{language} : \text{Lucx}];$
end

Figure 23: Agent Communication Performative: Tell

$E' \ @ \ E'_T$
where
$E' = \text{"tell"};$
$E''_T = E'_1 \oplus E'_2 \oplus E'_3 \oplus E'_4 \oplus E'_5;$
$E'_1 = [\text{sender} : \mathsf{HBA}];$
$E'_2 = [\text{content} : B''_1 \boxtimes B_4];$
where
$B''_1 = \text{Box}[H_c \mid H_c \in \{\text{Marriott}, \text{Hilton}\}];$
$B_4 = \text{Box}[T_{2c} \mid T_{2c} \geq 14\text{pm}];$
end
$E'_3 = [\text{receiver} : \mathsf{PTAc}];$
$E'_4 = [\text{in} - \text{reply} - \text{to} : \text{Hotel} - \text{Infor}];$
$E'_5 = [\text{language} : \text{Lucx}];$
end

Figure 24: Agent Communication Performative: Tell - Add Constraint

8.2.4 Information Fusion

The querying agent, in this case agent PTAc, fuses the information received from other agents. In particular, the querying agent should fuse both the collected domain information and the constraints. This is achieved with the use of *Box* algebra mentioned in Section 4.1.1 (Page 53). If the whole CSP was sent by PTAc to other agents which have knowledge on the same domains, it will use the "intersection rule"($\boxdot$) or "union rule"($\boxplus$) to fuse the collected information. If the CSP was decomposed by PTAc, and sub-CSPs were communicated to different agents, it will use the "join rule" ($\boxtimes$) to combine the collected information. For the TPE, assume that PTAc decomposed the problem into hotel, travel, and meeting times and had constructed the original contexts:

$$B'_1 = Box[H_c \mid H_c \in D_{H_c}],$$

$$B'_2 = Box[F_{fromc} \mid F_{fromc} \in D_{F_{fromc}}],$$

$$B'_3 = Box[T_{3c} \mid T_{3c} \in D_{T_c}], and$$

$$E_{TPE_c} = B'_1 \boxtimes B'_2 \boxtimes B'_3.$$

In reply to the query, let the responses received be

$$B''_1 = Box[H_c \mid H_c \in \{Hilton, Marriott\}],$$

$$B''_2 = Box[F_{from_c} \mid F_{fromc} \in \{[T_{fl} : 10am, T_{fa} : 13pm, F_n : AC32], [T_{fl} : 16pm, T_{fa} : 19pm, F_n : AC38]\}], and$$

$$B''_3 = Box[T_{3c} \mid T_{3c} \in \{Oct.3 - 10am, Oct.6 - 14pm\}].$$

Agent PTAc computes the expression:

$$E'_{TPE_c} = B''_1 \boxtimes B''_2 \boxtimes B''_3$$

to fuse the domain information received from the agents. Similarly, E'_{TPE_l} is also constructed for Liz. The final domain set should be :

$$E'_{TPE} = E'_{TPE_c} \boxtimes E'_{TPE_l}.$$

Meanwhile, if the agent HBA has added a new constraint B_4, agent PTAc computes the expression $B'_{TPE} = B_{TPE} \boxtimes B_4$ to fuse the constraint received from agents HBA. The new CSP constructed for the TPE is $\langle E'_{TPE};\ B'_{TPE} \rangle$.

8.2.5 Problem Solving

$$
\begin{array}{l}
\mathrm{E}'' \ @ \ \mathrm{E}''_{\mathrm{A}} \\
\texttt{where} \\
\quad \mathrm{E}'' = \text{"}\texttt{ask} - \texttt{one}\text{"}; \\
\quad \mathrm{E}''_{\mathrm{A}} = \mathrm{E}''_1 \oplus \mathrm{E}''_2 \oplus \mathrm{E}''_3 \oplus \mathrm{E}''_4 \oplus \mathrm{E}''_5; \\
\quad \mathrm{E}''_1 = [\texttt{sender} : \mathsf{PTAc}]; \\
\quad \mathrm{E}''_2 = [\texttt{content} : \mathrm{B}'_1 \boxtimes \mathrm{B}'_2 \boxtimes \mathrm{B}'_3]; \\
\quad \texttt{where} \\
\quad \mathrm{B}'_1 = \mathrm{Box}[\mathrm{H}_{\mathrm{c}} \mid \mathrm{H}_{\mathrm{c}} \in \{\texttt{Marriott}, \texttt{Hilton}\}]; \\
\quad \mathrm{B}'_2 = \mathrm{Box}[\mathrm{F}_{\mathrm{from_c}} \mid \mathrm{F}_{\mathrm{fromc}} \in \{[\mathrm{T}_{\mathrm{fl}} : \texttt{10am}, \mathrm{T}_{\mathrm{fa}} : \texttt{13pm}, \mathrm{F}_{\mathrm{n}} : \texttt{AC32}], \\
\qquad [\mathrm{T}_{\mathrm{fl}} : \texttt{16pm}, \mathrm{T}_{\mathrm{fa}} : \texttt{19pm}, \mathrm{F}_{\mathrm{n}} : \texttt{AC38}]\}]; \\
\quad \mathrm{B}'_3 = \mathrm{Box}[\mathrm{T}_{3\mathrm{c}} \mid \mathrm{T}_{3\mathrm{c}} \in \{\texttt{Oct.3} - \texttt{10am}, \texttt{Oct.6} - \texttt{14pm}\}]; \\
\quad \texttt{end} \\
\quad \mathrm{E}''_3 = [\texttt{receiver} : \mathsf{PSA}]; \\
\quad \mathrm{E}''_4 = [\texttt{reply} - \texttt{with} : \texttt{Meeting} - \texttt{Time}]; \\
\quad \mathrm{E}''_5 = [\texttt{language} : \texttt{Lucx}]; \\
\texttt{end}
\end{array}
$$

Figure 25: Agent Communication Performative: Ask for PSA

Problem Solving Agent(PSA) receives the CSPs from different agents and provides a solution satisfying either all CSPs or at least one CSP. The PSA can either use a Domain Specific CSP Solver or a General CSP Solver. A general CSP Solver may run a backtrack algorithm to search for the full space. A Domain Specific CSP Solver will act as a unifier,

discussed in Section 8.1.2. For the simplified TPE, Problem Solving Agent can use some heuristics within a General CSP Solver. Such heuristics, which are meta constraints, may reduce *Box* expressions dramatically and converge to a feasible solution. As an example, if it is known that staying at Marriott hotel might enable one to agree for an early morning meeting in October, then the availability of that hotel should first be evaluated. If the hotel is not available for the requested dates, then without further search we can declare the CSP to be inconsistent. To seek a consistent solution, the solver may weaken certain constraints and recompute the *Box* expressions.

The query from PTAc to PSA is evaluated as the expression $E'' \ @ \ E''_A$ in Figure 25.

The reply from PSA to PTAc is the expression $E''' \ @ \ E'''_T, E'''_T = E'_1 \oplus E'_2 \oplus E'_3 \oplus E'_4 \oplus E'_5$ in Figure 26.

$\mathtt{E'''} \ @ \ \mathtt{E'''_T}$
where
$\mathtt{E'''} = \text{"tell"};$
$\mathtt{E'''_T} = \mathtt{E'''_1} \oplus \mathtt{E'''_2} \oplus \mathtt{E'''_3} \oplus \mathtt{E'''_4} \oplus \mathtt{E'''_5};$
$\mathtt{E'''_1} = [\mathtt{sender} : \mathsf{PSA}];$
$\mathtt{E'''_2} = [\mathtt{content} : \mathtt{B''_1} \boxtimes \mathtt{B''_2} \boxtimes \mathtt{B''_3}];$
where
$\mathtt{B''_1} = \mathtt{Box}[\mathtt{H_c} \mid \mathtt{H_c} \in \{\mathtt{Marriott}\}];$
$\mathtt{B''_2} = \mathtt{Box}[\mathtt{F_{from_c}} \mid \mathtt{F_{from_c}} \in \{[\mathtt{T_{fl}} : \mathtt{10am}, \mathtt{T_{fa}} : \mathtt{13pm}, \mathtt{F_n} : \mathtt{AC32}]\}];$
$\mathtt{B''_3} = \mathtt{Box}[\mathtt{T_{3c}} \mid \mathtt{T_{3c}} \in \{\mathtt{Oct.3 - 10am}\}];$
end
$\mathtt{E'''_3} = [\mathtt{receiver} : \mathsf{PTAc}];$
$\mathtt{E'''_4} = [\mathtt{in - reply - to} : \mathtt{Meeting - Time}];$
$\mathtt{E'''_5} = [\mathtt{language} : \mathtt{Lucx}];$
end

Figure 26: Agent Communication Performative: Tell from PSA to PTAc

The figure 27 shows the scenario of message passing sequence diagram for TPE. The message m_1 corresponds to the performative shown in Figure 22. Similarly, m_2 corresponds

to Figure 23, m_7 corresponds to Figure 25, and m_8 corresponds to Figure 26.

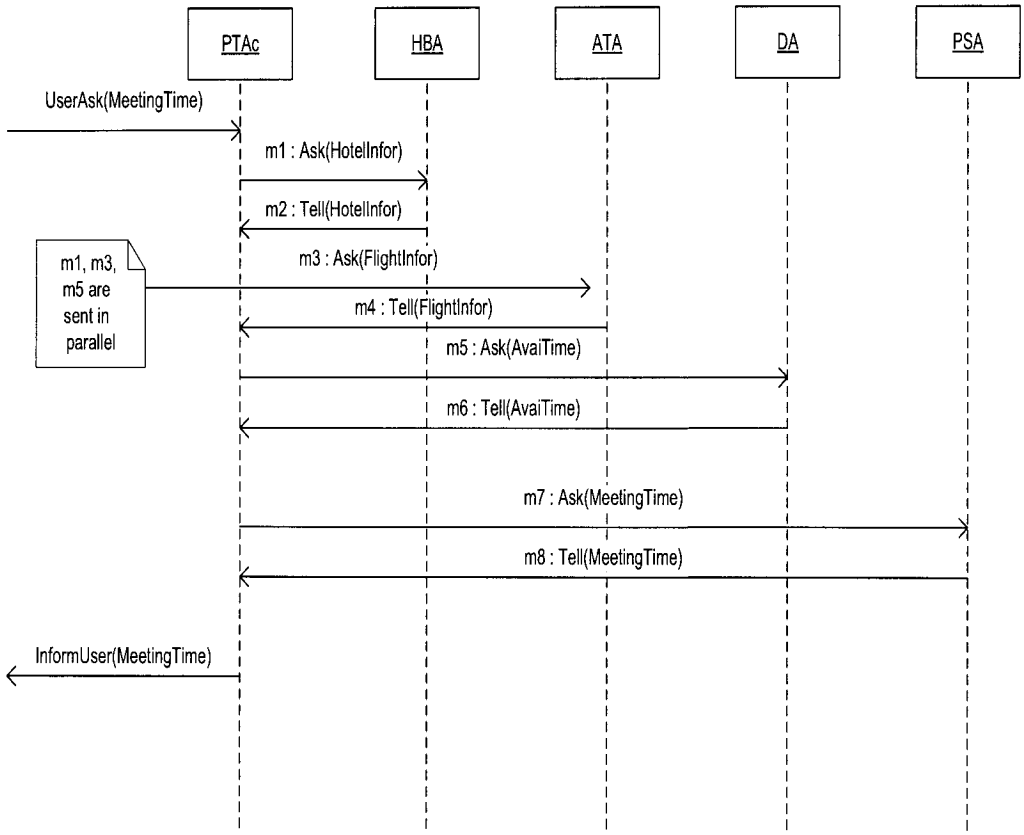

Figure 27: Message Passing Sequence Diagram for TPE

8.3 Summary

We have discussed how Lucx may be used as a constraint programming language. One of the distinct merits of Lucx is its ability to dynamically compose/decompose CSPs, and consequently it is advantageous to use Lucx as CCL in agent-based systems where agents' knowledge base changes dynamically. The following is the list of future research goals.

- For a problem described in a natural language it is not an easy task to formalize it and represent it in Lucx by the application developers. Even for a simplified TPE the number of variables, domains, domain values, and constraints are too many to comprehend and enumerate. A goal is to develop a graphical representation for CSP modeling and a user interface to incrementally modify the representation. The representation must allow composition from sub-CSPs and decomposition of a CSP into sub-CSPs. We plan to have an automatic translator of the graphical representation of the CSP into Lucx representation. The goal is to interactively create CSP representation in Lucx and automate its solution.

- Different constraint solvers have been announced, in particular, the ILog system [33], which is implemented in C++. An interesting work would be to use ILog in the background with the Lucx program translated from the graphical representation providing a high-level CSP representation.

- Lucid, initially designed for program verification, is founded on Intensional logic. The reasoning ability in Lucx is derived from this logical basis. Since contexts are explicit in Lucx, we can formulate proof rules for constraint reduction, which in turn will be a basis for formal reasoning with contexts in Lucx. Such a reasoning system can be exported to agent-based systems for verifying the correctness of protocols for agent conversations.

Chapter 9

Conclusion and Future Work

The notion of context is the cornerstone of the intensional programming paradigm. The previous versions of Lucid were merely using the notion of context of evaluation. They provided a single operator for navigation along a dimension of the evaluation context, but did not provide a mechanism to represent and manipulate contexts as first class objects. A major contribution of this thesis is the introduction of context as a first class object and semantics for the language Lucx : Lucid extended with context.

Contexts in Lucx have the following features:

- Contexts can be dynamically modified through operators defined for contexts. New contexts can be dynamically created from those defined in a program.
- Context calculus provides compilation rules for calculating a context from a context expression, and evaluation rules for expressions over context expressions.
- In previous version of Lucid, dimensions can be named explicitly. Hence stream can

be extracted only along one dimension; Similarly, in Lucx, by making context as first class objects, evaluation can be arbitrarily done in different contexts. Moreover, being a first class object, context exists independently in the language. That is, one context may be used to evaluate different expressions, at the same time an expression can also be evaluated at different contexts. In particular, subexpressions of an expression may be evaluated independently at different contexts.

- Lucx allows the definition (infinite) streams of contexts, thus offering the mechanism to program non-terminating continuous systems, such as timed systems.

By introducing contexts as first class objects in the language, the expressive power of the language is increased by an order of magnitude. It allows the definition of aggregate contexts, which are a key feature to achieve efficiency of evaluation through granularization of the manipulated data. It also allows to use the paradigm for agent communication by allowing the sharing and manipulation of multidimensional contextual information among agents. It allows the use of the paradigm for programming timed system as well.

In addition to the future works mentioned in the summary sections of thesis chapters, the following topics are to be investigated in future:

Implementing the Lucx compiler It is feasible to integrate Lucx into the GIPSY platform using the approaches shown in Chapter 5. This way, the programs written in Lucx can eventually run on the GIPSY. On the other hand, the generality of the GIPSY design will also get validated. Meanwhile, GIPSY programs can be written as hybrid programs allowing Java functions to be called by the Lucx part of the program. Hence as an investigation

of how to divide the programs into Lucx and Java parts to get coarse-granular execution and speed up the performance may also be interesting.

Reasoning and Formal Verification Lucid is founded on Intensional logic. The reasoning ability in Lucx is derived from this logical basis. Since contexts are explicit in Lucx, we can formulate proof rules for constraint reduction, which in turn will be a basis for formal reasoning with contexts in Lucx.

Lucid was originally invented as a program verification language. Naturally we can expect Lucx, a conservative extension of Lucid, to serve as a programming language verification. It is desirable to integrate Lucx programs with a reasoning system built on intensional logic. This way we will have a natural formal verifier for programs written in Lucx.

Context-aware Systems The context theory developed in this thesis will be applied to a formal development of context-aware computing applications. *Context-awareness*, first introduced by Schilit [54] has been generically characterized by Pascoe [52] as the ability of a system sensing, interpreting, and adapting to different contexts. Context-awareness was mainly studied for improving performances in user interface design. Today there is a wide body of literature on pervasive computing, ubiquitous computing systems where adaptation to context awareness seems essential. A common example of context-aware application is the *Anti-brake Locking System* (ABS) available in most of the modern day cars [36].

Context-awareness, as the ABS example suggests, consists of two parts: *context construction* and *context adaption*. The contribution of the context theory to such applications

will be mainly in formalizing the notion of context and providing a representation and manipulation of contexts for dynamic allocation of services and resources [62]. The environmental contexts can be converted into micro_contexts. Applying the $\oplus$ operation to micro_contexts, simple contexts representing contextual situations can be *constructed.* The stream of constructed contexts are steadily fed into the context adaptation process, which will determine for each context in the stream the necessary action to be taken, and how the context itself has to be modified externally. This can be achieved by introducing context-dependent functions, discussed in Section 4.2(Page 58), at different adapter modes. If the behavior of the adapter system could be described by an ESM, which is already embedded in Lucx, it would become feasible to program context-aware applications in Lucx. Combining the features of Lucx as a programming verification language and at the same time, a development of context-aware applications, we can have a verification-driven design methodology in Lucx.

Bibliography

[1] V.S. Alagar, A. Achuthan, D. Muthiayen. TROMLAB: *A Software Development Environment for Real-Time Reactive Systems.* (first version 1996, revised 2001), Technical Report, Department of Computer Science, Concordia University, Montreal, Canada.

[2] J.R.Abrial. *The B-Book: Assigning Programs to Meanings.* C.U.P., 1996.

[3] R.Alur, C.Courcoubetis, N.Halbwachs, T.A.Henzinger, P.H.Ho, X.Nicollin, A.Olivero, J.Sifakis, S. Bornot. *The Algorithmic Analysis of Hybrid Systems.* Theoretical Computer Science, 138:3-34, 1995.

[4] Edward A. Ashcroft, William W. Wadge. *Lucid - A Formal System for Writing and Proving Programs.* SIAM J. Comput. 5(3): 336-354, 1976.

[5] Edward A. Ashcroft, William W. Wadge. *Lucid, a Nonprocedural Language with Iteration.* Commun. ACM 20(7): 519-526 (1977)

[6] E. Ashcroft, A. Faustini, R. Jagannathan, W. Wadge. *Multidimensional, Declarative Programming.* Oxford University Press, London, 1995.

[7] V.S. Alagar, J. Holliday, P.V. Thiyagarajan, B. Zhou. *Agent Architecture for E-Commerce Transactions.* Technical Report, Department of Computer Engineering, Santa Clara University, Santa Clara, CA 95053, U.S.A.

[8] Vangalur S. Alagar, Greetha Ramanathan. Functional Specification and Proof of Correctness for Time Dependent Behaviour of Reactive Systems. Formal Asp. Comput. 3(3): 253-283 (1991)

[9] Vasu Alagar, Olga Ormandjieva, Kaiyu Wan, Mao Zheng *Ensuring Service Availability for Media Handling in Mobile Ad-hoc Networks* Proceedings of AMT 2005 conference, Page 133-136. May 19-21, 2005, Kagawa International Conference Hall, Takamatsu, Kagawa, Japan

[10] V. S. Alagar, J. Paquet, K. Wan. *Intensional Programming for Agent Communication.* DALT'04. Lecture Notes in Computer Science, Springer-Verlag, Vol. 3476, Page 239-255. ISBN: 3-540-26172-9.

[11] V.S.Alagar, Joey Paquet, Kaiyu Wan. *Contexts in Intensional Programming.* Technical Report, Department of Computer Science, Concordia University, Montreal, Canada, April 2004.

[12] V.S.Alagar, J.Holliday, P.V.Thiyagarajan, B.Zhou. Agent Types and Their Formal Descriptions. Technical Report, Department of Computer Engineering, Santa Clara University, Santa Clara, CA, U.S.A., May 2002.

[13] Krzysztof R.Apt. *Principles of Constraint Programming* Cambridge University Press, 2003

[14] V.S. Alagar, K.Wan. *An E-Commerce Architecture for Secure Transactions* Proceedings of IADIS International Conference e-Commerce 2004, Lisbon, Portugal, Dec.2004

[15] S. Bornot, J. Sifakis. *On the Composition of Hybrid Systems.* In *International NATO School on "Verification of Digital and Hybrid systems"*, LNCS, Springer Verlag,1997.

[16] R. Carnap. *Meaning and Necessity.* University of Chicago Press, 1989. second edition.

[17] P.Caspi, D,Pilaud, N.Halbwachs, J.A.Plaice. *LUSTRE: A declarative language for programming synchronous systems.* P.O.P.L. 1987

[18] A.K.Dey. *Understanding and Using Context.* Personal and Ubiquitous Computing Journal 5(1).pp.4-7.2001.

[19] Theodosis Dimitrakos, Juan Bicarregui, Brian Matthews, T. S. E. Maibaum *Compositional Structuring in the B-Method: A Logical Viewpoint of the Static Context* Lecture Notes In Computer Science; Vol. 1878, Proceedings of the First International Conference of B and Z Users on Formal Specification and Development in Z and B, Pages: 107 - 126, Year of Publication: 2000, ISBN:3-540-67944-8

[20] P. Degano and C. Priami. *Enhanced operational semantics: A tool for describing and analyzing concurrent systems.* ACM Computing Surveys, 33(2):135176, 2001.

[21] A.K. Dey, D. Salber, G.D. Abowd. *A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-aware Applications*, anchor article of a special issue on Human Computer Interaction, Vol.16, (2001).

[22] D.Dowty, R.Wall, and S.Peters. *Introduction to Montague Semantics.* Reidel Publishing Company, 1981.

[23] W.Du and W.W.Wadge. *A 3D Spreadsheet Based on Intensional Logic.* IEEE Software, Page 78-19, July 1990.

[24] H.-D. Ebbinghaus, J. Flum and W. Thomas, *Mathematical Logic* Springer-Verlag, 1984.

[25] Tim Finin, Richard Fritzson, Don McKay, Robin McEntire. *KQML as an Agent Communication Language.* Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94), ACM Press, November 1994.

[26] FIPA Semantic Language Specification. *FIPA Specification repository*, FIPA-specification identifier XC00008G, September 2000 Foundation for Intelligent Physical Agents, Geneva, Switzerland.

[27] FIPA CCL Content Language Specification. FIPA TC C, Document Number: XC00009B www.fipa.org/specs/fipa00009/XC00009B.html,2001/08/10

[28] *Larch: Languages and Tools for Formal Specification.* John V. Guttag and James J. Horning, editors, Springer-Verlag, 1993.

[29] R. V. Guha. *Contexts: A Formalization and Some Applications.* Ph.d thesis, Stanford University, February 10,1995.

[30] C.Heitmeyer and N.Lynch. *The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems* In Proceedings of the 15th IEEE Real-time Systems Symposium, RTSS'94, page 120-131, San Juan, Puerto Rico, Dec 1994.

[31] P. Van Hentenryck. The OPL Optimization Programming Language The MIT Press, 1999.

[32] T.A. Henzinger, H. Wong-Toi. *Using HyTECH to SynthesiZe Control Parameters for a Steam Boiler.* In *Steam boiler control specification problem,*: Ed. J.R. Abrial, Lecture Notes in Computer Science, Vol.1165, pp. 265-282, 1996.

[33] Ilog White Paper. Available at `http//www.ilog.com/products/optimization/products.vfm`, 2003.

[34] Vincent Jayawardene. *An intensional engine on L4* BE Thesis, University of NSW, Sydney 2052, Australia, 1999

[35] R. Jagannathan, C. Dodd, and I. Agi. *GLU: A High-Level System for Granular Data-Parallel Programming.* Concurrency: Practice and Experience, vol. 9,1997.

[36] Cheverst, K., N. Davies, K. Mitchell and C. Efstratiou. *Using Context as a Crystal Ball: Rewards and Pitfalls.* Personal Technologies Journal, Vol. 3 No5, pp. 8-11, 2001.

[37] Y. Labrou, T. Finin. *Semantics for an Agent Communication Language.* Agent Theories, Architectures, and Languages IV, M. Woodridge, J.P. Muller, and M. Tambe, eds., Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 1998.

[38] Y. Labrou, T. Finin, and Y. Peng. *Agent Communication Languages: The Current Landscape.* IEEE Journal on Intelligent Agents, Amrch/April 1999, pp. 45-52.

[39] J.McCarthy. *Geneality in aritificial intelligece.* Communication of ACM, 1987.

[40] J.McCarthy. *Notes on formalizing context.* In Proceedings of the Thirteeth International Joint Conferece on Artificial Intelligence, 1993.

[41] Peter D. Mosses *The varieties of programming language semantics and their uses.* 4th International Andrei Ershov Memorial Conference on Perspective of System Informatics, PSI01, LNCS 2244. Berlin: Springer-Verlag, 2001. 165-190.

[42] H. Marchand, E.Rutten, M. Le Borgne, M. Samman. *Formal verification of programs specified with signal: application to a power transformer station controller.* Science of Computer Programming 41 (2001) 85-104.

[43] D. Muthiayen. *Real-Time Reactive System Development -A Formal Approach Based on UML and PVS.* Phd. Thesis, Department of Computer Science, Concordia University, Montreal, Canada, January 2000

[44] J. Paquet, P. Kropf. *The GIPSY Architecture.* DCW 2000, 144-153.

[45] Joey Paquet and John Plaice. *Dimensions and functions as values.* Proceedings of the Eleventh International Symposium on Lucid and Intensional Programming, Sun Microsystems, Palo Alto, California, USA, may 1998.

[46] Joey Paquet. *Intensional Scientific Programming* Ph.D. Thesis, Département d'Informatique, Universite Laval, Quebec, Canada, 1999

[47] Joey Paquet and John Plaice. *The Intensional Relation* In Intensional Programming I, pages 214–227, World Scientific, Singapore, 1996.

[48] John Plaice. *RLucid, a general real-time dataflow language.* Formal Techniques in Real- Time and Fault-Tolerant Systems, pages 363-374, Berlin, 1992.

[49] G. D. Plotkin. *A structural approach to operational semantics.* Lecture Notes DAIMI FN19, Dept. of Computer Science, Univ. of Aarhus, 1981.

[50] John Plaice and Joey Paquet. *Introduction to Intensional Programming.* In Intensional Programming I, pages 1-14. World Scientific, Singapore, 1996

[51] John Plaice, William Wadge. *A new approach to version control.* IEEE Transactions on Software Engineering, 3(19):268-276, 1993.

[52] J. Pascoe, N. Ryan and D. Morse. *Issues in developing context-aware computing.* Proc. Intl. Symposium on Handheld and Ubiquitous Computing, LNCS 1707, Springer, 1999, 208-221

[53] P.Rondogiammis, William Wadge. *Intensional Programming Languages.* Proceedings of the First Panhellenic Conference on New Information Technologies, 1998

[54] B. Schilit, M. Theimer. *Context-aware Computing Applications.* In Proceedings of the 1st International Workshop on Mobile Computing Systems Applications, pp. 85-90, 1994.

[55] P.Swoboda. *Practical Languages for Intensional Programming.* MSc Thesis, Computer Science, University of Victoria, Canada (1999).

[56] J. Springintveld, F. Vaandrager, P. Dargenio. *Testing Timed Automata.* Theoretical Computer Science, vol. 254, pp.225-257,2001.

[57] S.Tao. *Indexical Attriute Grammars.* PhD thesis, Department of Computer Science, University of Victoria, Canada, 1994.

[58] R.Thomason, editor. *Formal Philosophy, Selected Papers of R.Montague.* Yale University Press, 1974.

[59] K. Wan, V.S. Alagar, J. Paquet. *Real Time Reactive Programming Enriched with Context.* ICTAC2004, Lecture Notes in Computer Science, Springer-Verlag, Vol. 3407, Page 387-402. ISBN 3-540-25304-1.

[60] K. Wan, V.S. Alagar, J. Paquet. *Lucx : Lucid enriched with Context.* PLC'05 - The 2005 International Conference on Programming Languages and Compilers, Page. 48-54. CSREA Press, ISBN 1-932415-75-0.

[61] K. Wan, V.S. Alagar. *An Intensional Programming Approach to Multi-agent Coordination in a Distributed Network of Agents.* Declarative Agent Languages and Technologies III, Third InternationalWorkshop, DALT 2005, Lecture Notes in Computer Science, Springer-Verlag, Vol. 3904, Page 205-222, ISBN 3-540-33106-9.

[62] K. Wan, V. S. Alagar, J.Paquet *An architecture for developing Context-Aware Systems* The Second International Workshop on Modeling and Retrieval of Context (MRC2005), Lecture Notes in Computer Science, Springer-Verlag, Vol. 3946, Page 48-61.

[63] Kaiyu Wan, V.S. Alagar, J. Paquet. *A Context theory for Intensional Programming.* Context Representation and Reasoning CRR-05, Proc. of Context Representation and Reasoning (CRR-05), Satellite Workshop of The Fifth International and Interdisciplinary Conference on Modelling and Using Context (CONTEXT-05). CEUR Workshop Proceedings, ISSN 1613-0073, online CEUR-WS.org/Vol-136.

[64] William W. Wadge, Gord Brown, Monica M. C. Schraefel, Taner Yildirim. *Intensional HTML.* PODDP 1998: 128-139.

[65] Michael Wooldridge. *Verifiable Semantics for Agent Communication Languages.* Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS'98).

[66] S. Willmott, M. Calisti, B. Faltings, S. Macho-Gonzalez, O. Belakhdar, M. Torrens. *CCL: Expressions of Choice in Agent Communication.* The Fourth International Conference on MultiAgent Systems (ICMAS-2000), Held July 7-12, 2000, Boston MA,

USA. pages 325-332.

[67] W.W.Wadge, E.A.Ashcroft. *Lucid, the dataflow programming language.* Academic Press, 1985

[68] William W.Wadge. *Intensional Markup Language.* DCW 2000: 82-89

[69] William W.Wadge. http://i.csc.uvic.ca/home/hei/hei.ise