Abstract Reduction Operation Models in the LIME Programming Model

Master Thesis

Mani Zandifar
Abstract Reduction Operation Models in the LIME Programming Model

Master Thesis by

Mani Zandifar
Abstract Reduction Operation Models in the LIME Programming Model

Author: Mani Zandifar
Student id: 1393715
Email: mzandifar@student.tudelft.nl

Abstract

Reduction operations are frequently used operations which reduce the size of an input by selecting or combining its element and producing that as the output. Reduction operations can be easily handled in sequential computers in $O(n)$ time. But in a parallel systems this time complexity can be reduced to $O(\log n)$ by using algorithms like tree-based reduction. However, this tempting decrease in the time complexity cannot be achieved without several platform specific tricks and optimizations.

In order to harness the power of parallel systems for reduction operations two major steps should be taken. First, a single standard model should be defined for such operations. Second, the model should be easily converted to different parallel platforms without any or with small changes to the underlying model. This document materializes the latter by defining a tool-chain called ‘LIME tool-chain’ and the former by defining dataflow models which can be fed into the tool-chain.
Although my name appears as the sole author of the thesis, in reality a number of people are responsible for the work. First and foremost, I would like to thank my thesis advisors, Prof. Henk Sips and Ir. Pjotr Kourzanov. They steered my research efforts to attack interesting and potentially solvable problems in parallel computing. I must thank Henk for his motivating suggestions, encouragements and supports during my MSc project. He let me develop my knowledge and taught me the process of good scientific research. Many thanks Pjotr, for your always positive attitude and for helping me to learn more and more in many topics ranging from programming styles to the way of thinking about scientific problems. Working with Henk and Pjotr has been very inspiring: their energy, dedication, creative ideas and excellent organization. I should also thank Alexander Amesfoort for his dedication and help in many stages of my work. I acknowledge Prof. Arjen de Vries and Dr. Johan Pouwelse for their participation in my MSc thesis committee. The outcomes of the last year of interaction with these people did not get reflected in the thesis, but I feel they deserve to be mentioned anyway. I could not have wished for better collaborators and coaches. Your contributions, detailed comments and insights have been of great value for me. I am also grateful for the opportunity to be part of the research group at NXP.

My warmest thanks go to my friends, very specially, Ghazale, Arman, Saman, Jeremy, Frank, Niels, Arash, Navid, Katayoun, Amir, Ata, Soroush, and Sheida for their cooperations, and for making me forget for brief periods of time about my work and my thesis.

I would like to express my gratitude and humility to my parents and my siblings. Even thousands of kilometers apart, they have been present through every step of my life, providing support in difficult times. They have been a constant source of inspiration, and this thesis is dedicated to them.

Finally, I would like to give my special thanks to Saeideh A Riseh whose patient love enabled me to complete this work.

Mani Zandifar
Delft, The Netherlands
29th June 2009
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>1 Dataflow Computational Models</td>
<td>4</td>
</tr>
<tr>
<td>1.1 Pure Dataflow Models</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Synchronous Dataflows (SDF)</td>
<td>8</td>
</tr>
<tr>
<td>1.3 SDF Relaxations</td>
<td>14</td>
</tr>
<tr>
<td>2 LIME - Less Is MorE</td>
<td>18</td>
</tr>
<tr>
<td>2.1 Definitions</td>
<td>19</td>
</tr>
<tr>
<td>2.2 Compilation Flow</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Analysis Models</td>
<td>22</td>
</tr>
<tr>
<td>2.4 Supported Backends</td>
<td>23</td>
</tr>
<tr>
<td>3 LIMEclipse - LIME GUI Plug-in for Eclipse</td>
<td>26</td>
</tr>
<tr>
<td>3.1 The Motivation for a Rich Graphical Editor</td>
<td>26</td>
</tr>
<tr>
<td>3.2 Graphical Frameworks</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Eclipse GEF - Graphical Editor Framework</td>
<td>32</td>
</tr>
<tr>
<td>3.4 LIMEclipse In Details</td>
<td>34</td>
</tr>
<tr>
<td>3.5 Future Works</td>
<td>39</td>
</tr>
<tr>
<td>4 Abstract Model for Associative and Commutative Reduction Operations</td>
<td>42</td>
</tr>
<tr>
<td>4.1 Reduction Operations</td>
<td>43</td>
</tr>
<tr>
<td>4.2 Reduction Operations Background</td>
<td>44</td>
</tr>
<tr>
<td>4.3 Reduction Models in LIME</td>
<td>48</td>
</tr>
<tr>
<td>4.4 The Collect/Reduce Model in Database Systems</td>
<td>54</td>
</tr>
<tr>
<td>Bibliography</td>
<td>58</td>
</tr>
<tr>
<td>4.5 LIMEclipse</td>
<td>60</td>
</tr>
<tr>
<td>4.6 MPI</td>
<td>62</td>
</tr>
</tbody>
</table>
List of Figures

1.1 A simple dataflow model example, which adds A and B, multiplies B by C and produces the output by reducing the latter from the former ................. 4
1.2 A nested component example create by LIMEclipse ....................... 6
1.3 A component which keeps history by re-feeding current state variable to itself .......................................................... 7
1.4 A merger actor which merges its inputs based on a control data [24]. ........ 7
1.5 Dataflow granularity optimization curve [24]. .............................. 8
1.6 A simple SDF example consisting of nodes with predefined rate for consumption and production .............................................................. 9
1.7 An SDF example .............................................................. 10
1.8 (a) An example SDF (b) PAPS example with J = 2 for graph shown in (a) (Fig. from [28]) ................................................................. 12
1.9 Precedence graph for Fig. 1.8(a) (example is from [28]) ......... 13
1.10 Static buffering example (example is from [29]) ....................... 13
1.11 The relation between dataflow categories ................................. 15
1.12 A sample node: (a) CSDF: Consumption and production rate for tokens change in a cycle (Fig. based on [22]) (b) SDF: a similar SDF graph of (a) ......................... 15
1.13 Two important BDF actors: Switch and Select. They behave differently based on the data which is received from the control unit arc ................................. 16
1.14 Two types of VRDF actors: Consumer and Producer. They consume and produce different number of tokens based on ‘size’ control input .............. 17
2.1 LIME is a middle-end capable of converting a single model to several parallel backend ................................................................. 19
2.2 A simple source-sink model in LIME: components are the large boxes with gray titlebar; input ports are green rectangles; output ports are orange ovals; arcs are solid lines; associations are dashed lines; component, input/output port, arc and association ........................................ 21
2.3 Compilation flow of LIME (Fig. from [27]) ................................. 22
3.1 (a) Simple diagram in mxGraph; (b) Simple diagram in JGraphPro .... 28
3.2 yEd a rich graph editor framework .................................................. 29
3.3 CoG graph editor ............................................................................... 29
3.4 ArgoUML - a UML diagram editor based on GEF standalone .............. 29
3.5 Piccolo - resizable box example ......................................................... 30
3.6 Piccolo - graph editing example ......................................................... 30
3.7 Circuit example in Eclipse GEF ............................................................ 30
3.8 GEF components and dependencies (Fig. from [4]) ............................ 32
3.9 GEF, Draw2D and SWT relations (Fig. from [4]) ............................... 33
3.10 Draw2d Tree(Fig. from [4]) ................................................................. 33
3.11 Draw2d Layout Managers(Fig. from [4]) ........................................... 33
3.12 Editparts as connectors between model and view(Fig. from [4]) ............ 34
3.13 Newer LIME GUI based on Eclipse GEF with all supported elements ... 35
3.14 Class diagram of model classes in LIMEclipse .................................. 36
3.15 Class diagram for view classes in LIMEclipse .................................. 36
3.16 Class diagram for controller classes in LIMEclipse ............................ 37

4.1 Reduction operation - converting an input to a smaller output. ............. 43
4.2 A binary reduction tree for 16 elements. Each node contains a value. Each two
edges in each stage sum up the values of two nodes. ................................. 43
4.3 The MPI reduction - reducing $m = 28$ (A-Z) arrays of size $n$ to an array of size $n$. 44
4.4 Tests results for timing reduction example using 1 to 64 nodes. The shortest
time is when 63 nodes are used 0.010s. .................................................. 46
4.5 Simple sequential model for reduction in LIME. The same component can be
drawn for the implicit parallel reduction model ........................................ 49
4.6 Instantiation occurs whenever there is a mismatch between consumption and
production rate on an edge. ........................................................................ 50
4.7 Single-stage explicit parallel reduction model using cloned components. .... 51
4.8 Explicit parallel reduction model using the full reduction tree. ................ 51
4.9 Explicit dynamic parallel tree-based reduction model. .......................... 53
4.10 An example for Command classes. AddCommand prevents a port to be added
to something else than LIMEComponent ................................................. 60
4.11 Properties of a model element shown in visual editor .......................... 61
4.12 How new properties can be defined for a model class ......................... 61
4.13 XMLFactory - connection between LIME and LIMEclipse ................. 62
Introduction

Almost every scientist is now aware of the power of parallel systems for computations. However, since many parallel systems exist nowadays, the selection among these systems for a specific application is almost a difficult task. Most of these parallel systems have their own architecture, interconnection network or memory model. In addition each of these systems perform for an application better than the others.

Suppose a scientist picks one of these systems and writes his application, but unfortunately, he finds out that his choice results in a slow application. Now, if he wants to migrate his application to another parallel system in order to achieve a better performance, he is obliged to write the same application from scratch. It means that not only he has to reinvent the wheel, but he has to learn how to work with the new tools to reinvent the wheel.

A way to overcome this problem is to provide a level of abstraction over the existing parallel systems. That is what many parallel languages have done so far. However, many of these languages are confined to specific architectures and cannot be migrated easily to different architectures. Therefore, yet another layer of abstraction is needed to solve this problem. This layer can be a program model that is translatable to different parallel languages. Using this abstraction allows a programmer to define his program once and then run it on different parallel systems. This allows comparing different parallel systems without rewriting an application. On the other hand, the abstraction lets a programmer to focus on his application instead of parallel systems tricky details.

However, the question is that what should be the model and how would it be converted to practical computer applications. A good answer for the first question is the well-known parallel programming model: ‘dataflow model’. Dataflow models have concrete theoretical background in scheduling and buffer limit determination. To address the second question a software tool-chain is needed which on one hand can create dataflow models and on the other hand can convert these models to different parallel languages. This tool-chain is called as the ‘LIME tool-chain’ in this thesis and is comprised of two parts; LIMEclipse which allows visualization and editing dataflow models and LIME [26] which converts dataflow models to different parallel languages. The LIME tool-chain is used to create new parallel applications and can easily model common operations in parallel systems.

The LIME tool-chain is used in this thesis to propose models for challenging parallel ope-
rations called: ‘reduction operations’. In order to narrow down the scope of the thesis, the research is confined to only associative and commutative reduction operations. Reduction operations are handled differently in different parallel architectures and different parallel languages. In most of the cases if these operations do not execute in specific timesteps their verification and schedulability becomes problematic. Therefore, proposing a single model for these operations as a dataflow model overcomes both problems and is hence addressed in this thesis.

In order to explain how reduction operations are treated in the LIME tool-chain, first the background needed and the tool-chain are explained in this document. In Chapter 1 dataflow models and their many variations are defined. Chapter 2 then defines one end of the LIME tool-chain, the ‘LIME programming model’. Chapter 3 introduces ‘LIMEclipse’, the other end of the tool-chain which is used to graphically define and edit LIME-based parallel applications. With the information about the tool-chain, Chapter 4 explains how associative and commutative reduction operations are modeled in the LIME tool-chain.
Chapter 1

Dataflow Computational Models

Advances in parallel system over the years have motivated many researchers to design numerous programming models and languages for parallel systems. But so far, many of these models and languages are confined to the characteristics of the systems they are supposed to operate on [16]. It means that any change in the underlying system causes them to change, consequently. This can be explained as they are in fact an abstraction over the underlying parallel system but not the inherent parallelism in an application itself. In other words, they are not abstract and generic enough.

However, an abstract model like a dataflow model (Fig. 1.1) can define parallelism in an explicit way by showing potential parallel entities as components in a model (circles in Fig. 1.1), regardless of the underlying parallel system. Individual components in such a model are called to be potentially parallel entities since parallelization of all individual components can in some cases decrease the performance of the model. But this does not contradict the explicitness of the model. Even in those cases a group of components, together, can be considered as parallel entities which still depicts parallelism in visual representation of a model.

Briefly, dataflow models can depict run-time behavior of parallel systems in addition to the inherent parallelism in an application programmed for such systems [34]. Because of this advantage, dataflow models have been always a fruitful field for research in parallel
systems, both in software and hardware side and especially when the application can be divided into modules e.g. in DSP applications.

The expressiveness and usefulness of dataflow models in real world parallel applications was the reason for the author to do research on modeling reduction operations in variations of dataflow models. In order to understand those models precisely variations of dataflow models are explained in this chapter. First, pure dataflow models are described in Section 1.1. Then SDF (Synchronous Dataflow) is explained as a relaxation over pure dataflow model in Section 1.2. In Section 1.3 relaxations of SDF are described which are used in the final reduction models which are the main thread of this document; CSDF (Cyclo-Static Dataflow), BDF (Boolean Dataflow) and VRDF (Variable Rate Dataflow). This chapter creates the background which is needed for the rest of this document.

1.1 Pure Dataflow Models

Dataflow models are software models which are supposed to fairly depict the inherent parallelism of an application while showing the execution path. Dataflow models are defined as directed graphs like in Fig. 1.1 with actors (nodes) representing operators (single or multiple instructions) and arcs (edges) as data carriers. Typically, the operators in an actor are defined as code fragment in a host language like C and an actor executes the code fragment when it receives enough input data. Each execution of the code which is called invocation of the actor in dataflow terminology, consumes a set of incoming data and produces a set of outgoing data. The outgoing data travels through an outgoing arc or a set of outgoing arcs and is either the final output of the whole program or is an input to the other actors.

By defining actors as black boxes in dataflow models, they have gained importance in modular applications, especially in the cases which the same modules must be reused several times. As a blackbox, an actor must be without side-effects. Side-effect freedom is explained later in this section. But considering an actor as a blackbox, brings this idea to mind that an actor can consist of anything, even of nested actors. This idea is used in LIME and LIMEclipse (Chapters 2, 3) to define complex parallel applications. An example of nested actors is depicted using LIMEclipse graphical tool in Fig. 1.2.

Pure dataflow models are a type of dataflow models in which consumption and production by an actor are always one. They are also called SRDF (Single Rate Dataflow) or HDF (Homogeneous Dataflow). Since pure dataflow models contain the common properties of all dataflow model variations, their properties are explained in this section in more detail.

1.1.1 Rules for a Pure Dataflow Model

In general, two sets of rules exist for data flow models: execution rule and consistency rule. Execution rules determine when an actor is fireable (executable) and consistency rules ensure that a dataflow model works correctly and as expected.

Execution rules divide dataflow models into to broad categories: data-driven and demand-driven. In the former an actor is fireable when it has enough inputs on its incoming arcs.
But in the latter an actor is fireable only when its outputs are requested. The advantage of a data-driven model is that it avoids propagating a request up in the model, hence having a lower overhead in comparison to a demand-driven model. But on the other hand, the advantage of a demand-driven model is its lazy execution of actors which avoids executing unneeded actors. In practices both models are used widely when appropriate.

Consistency rules, instead, must be obeyed by all dataflow models and consist of two categories: model rules and arc rules. Model rules are actors-based and inter-actors relations and arc rules define arc-specific constraints.

Model rules can be defined as [16]:

- **visible dependencies** - no external or hidden rules are defined for dependencies between the actors. Visible data dependencies solely depicts dependencies between the actors.
- **no side-effects** - an actor does not modify its arguments (incoming data) or global variables. Instead, it works on a copy of them.
- **locality of effect** - changes on variables inside an actor are transparent to the other actors.
- **local error handling** - errors are handled internally in an actor, e.g. using a default value for an erroneous result.
- **single-assignment rule** - each variable is assigned a value only once in an actor.
- **history unawareness** - using history for data in an actor is explicitly shown in a model by having an arc from an actor to itself (Fig. 1.3).
A dataflow model obeying model rules has deterministic behavior. It always produces the same outputs for a given set of inputs [24].

In addition, arc rules prevent a model to have non-deterministic actions. Arc rules can be defined as the followings [24]:

- **restricted fork** - if an arc is forked then a copy of data is sent to each e.g. in Fig. 1.1 B is duplicated. This is equal to creating a dummy actor copying an input to several outputs.
- **no non-deterministic merge** - if two arcs merge then ordering of their data is not deterministic and is thus prohibited. Nevertheless, if forking is allowed and merging is not then forked arcs could never join again. In fact, this problem can be solved by defining *merger* actors. A merger actor is an actor with an extra input named *control data* which selects its outputs from its inputs based on the control data (Fig. 1.4)[24]. Apparently, in merger control input data is read before the data inputs.

Variations of pure dataflow models can be defined based on imposing changes on these rules.

### 1.1.2 Granularity

Typically, a single program can be modeled by several dataflow models, ranging from fine-grained parallelism (a thread for each instruction) to coarse-grained parallelism (the whole program in one model, like von Neumann model). But Johnston [24] showed that the optimal choice is a medium-grained approach (Fig. 1.5). This case is indeed a hybrid model of both extremes, “a dataflow with von Neumann extensions or vice versa” [24]. Such an optimal model is also used in the abstract model provided in Chapter 4.
Obviously, finding the best granularity highly depends on the hardware architecture and the software model being used and should be determined by several tests, but the essence of Johnston experiment is to avoid both extreme cases.

Figure 1.5: Dataflow granularity optimization curve [24].

1.1.3 Execution Schedule

To utilize the inherent parallelism of dataflow models, there should be a way to control the execution sequence, this is what so called as a schedule. Without a schedule correctness (no-deadlock and correct results) and performance of the system cannot be guaranteed. The task of a schedule is to determine which actors are fireable at each timestep and in which sequence they should run.

A schedule can be either dynamic or static. Dynamic scheduling means that a runtime scheduler determines fireable actors and assigns them to available processors at run-time [34]. Static scheduling means that the schedule must remain the same through the whole period of execution and it should be known before run-time. Therefore, static scheduling is done at compile-time.

Each scheduling has its own cons and pros. On one hand, dynamic scheduling is flexible to a system’s dynamic behavior, but it needs to poll at runtime to find fireable actors resulting in a rather large overhead. On the other hand, static scheduling has almost no overhead in runtime and is more analyzable, but it runs the same schedule for all periodically incoming data, hence being too rigid and is less expressive [34]. As a result, a scheduling method which could make a trade-off between the two approaches is the best choice for a dataflow model.

To find an optimal scheduling method, many researchers have introduced improvements over pure dataflow models in the literature. Among those Static Dataflow (SDF) [29] and Cyclo-Static Dataflow (CSDF) [34] which are described in the next sections have tried to make a trade-off between the two scheduling schemes. They are basically statically scheduled but in a more relaxed way.

1.2 Synchronous Dataflows (SDF)

Synchronous dataflow is a variation of pure dataflow in which the number of elements consumed from and produced on an arc is known at compile-time [24, 29](Fig. 1.6). This
is why they are also called MRDF (Multiple Rate DataFlow) in front of SRDF (Single Rate Dataflow). Consumption/production rates are independent of data [29] and remain constant in the execution of a model. Following this rule, an SDF ensures that after a number of invocations the buffers of its arcs align in the way they started at the first point, hence the term synchronous.

In addition, in an SDF, even communications between the processors are also known at compile-time, avoiding runtime control overhead. The only overhead is then the sequential controls for a code [29].

These properties make it possible for a complete SDF model to be statically schedulable [29]. Being statically schedulable, makes runtime behavior of a dataflow model predictable and analyzable. It also determines the upper bound for buffer sizes on the arcs of a model which is an essential requirement for practical applications. But in practice not always the whole model is an SDF. A model can instead contain SDF portions which are connected via asynchronous actors. In such cases one can statically schedule each SDF part and use another scheduling scheme for the rest of the graph, hence the term partial static scheduling [24].

But since in both full and partial SDFs static scheduling is needed for SDF parts, static scheduling and buffer size determination in SDFs are described in this section.

![Figure 1.6: A simple SDF example consisting of nodes with predefined rate for consumption and production](image)

### 1.2.1 SDF Scheduling

Finding a static schedule for an SDF is formalized by Lee in [28]. He names such a schedule as a periodic admissible parallel schedule (PAPS). But in order to make a PAPS for an SDF, one should first make a periodic admissible sequential schedule (PASS). The former is a schedule for a parallel machine and the latter is a schedule for a sequential machine. Both PAPS and PASS have two main properties. They are:

- **periodic** - the same schedule runs on an infinite data stream
- **admissible** - an actor is fired when it has enough data but not sooner

A few terms are needed to be defined before discussing about PASS and PAPS. First, a topology matrix is a matrix with actors as columns and arcs as rows which is similar to the incidence matrix for a directed graph. If an arc is connected to an actor it has a non-zero value indicating the number of elements produced (positive) or consumed (negative) by the actor on that arc. Since dataflow models are connected graphs, topology matrices have always $p$ columns and at least $p - 1$ rows in which $p$ stands for the number of actors in a
model. As an example, the matrix for the example given in Fig. 1.7 is:

\[
\Gamma = \begin{bmatrix}
a_1 & -b_1 & 0 \\
a_2 & 0 & -c_1 \\
0 & b_2 & -c_2 \\
\end{bmatrix}
\] (1.1)

Second, **buffers vector**, \( b(n) \), indicates the number of elements in all buffers in the system at time \( n \). Since each arc has one buffer such a vector has as many rows as the number of arcs in the graph.

Third, **invocation vector**, \( v(n) \), indicates which actor is activated at a certain moment. Obviously, this vector has as many rows as the number of actors in a dataflow model. If it is considered for a sequential schedule, it has only a single 1 for the actor which is invoked at time \( n \). For the example given in Fig. 1.7 with 3 actors, vector \( v \) can have one of the following values:

\[
v(n) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}
\] (1.2)

At each moment, sizes of the buffers are affected according to the actor being invoked and the number of tokens which the actor consumes and produces. This could be formulated using the above terms as:

\[
b(n+1) = b(n) + \Gamma v(n)
\] (1.3)
It can be proved by induction that for a tree-shaped SDF the above equation always holds [28]. Following the same induction it can be simply proved that for any SDF the following equation holds [28]:

\[ \text{rank}(\Gamma) \geq p - 1 \]  \hspace{1cm} (1.5)\]

However, as mentioned before the rank of a topology matrix cannot exceed \( p \). Therefore, it is either \( p \) or \( p - 1 \). To prove that it cannot be \( p \), buffers vector equation should be considered:

\[ b(n + 1) = b(n) + \Gamma v(n) \]  \hspace{1cm} (1.6)\]

This equation can be re-written as:

\[ b(n) = b(0) + \Gamma \sum_{i=0}^{n} v(i) \]  \hspace{1cm} (1.7)\]

in which \( n \) stands for the total number of actor invocations. Equation 1.7 can be simplified by writing the sigma as:

\[ q = \sum_{i=0}^{n} v(i) \]  \hspace{1cm} (1.8)\]

in which \( q \) means the number of invocations for each actor in the system in a complete cycle of the schedule. If then \( m \) is the number of cycles that the schedule is executed, the buffer size after \( m \) cycles is:

\[ b(m,n) = b(0) + m\Gamma q \]  \hspace{1cm} (1.9)\]

However, given that in practice, buffers sizes cannot be infinite, the second term should be zero:

\[ \Gamma q = 0 \]  \hspace{1cm} (1.10)\]

This means that at least one row in the topology matrix is linearly dependent on the other rows or in other words the rank of that topology matrix is less than \( p \). This proves that for a statically schedulable SDF, equation 1.4 should hold. In other words, if topology matrix for an SDF has a rank different than the above mentioned value, it will either result in a deadlock or its buffers sizes should be infinite for it to work.

However, if an SDF is statically schedulable, one should be able to first, define a schedule and second, prove its validity. Obviously, a schedule exist when a \( q \) can be determined with which equation 1.10 holds. It is proved in [28] that such \( q \) is easily determinable. With such a \( q \), the length of a cycle of the schedule is:

\[ \text{period} = \sum_{i=0}^{p} q[i] \]  \hspace{1cm} (1.11)\]

The schedule itself is defined as \( \text{actor}_{j}(i) \) in which \( i \) is the index of an actor and \( j \) corresponds to timestep of the schedule.

Knowing a static schedule (PASS) for an SDF, the goal is then to determine a PAPS for is. PAPS is normally created by considering several cycles of the sequential schedule and
interleaving independent parts of the cycles in the timeline on different parallel entities [28]. This technique is the same technique which is typically used in ‘operational research’ field.

A PAPS is defined formally as \( \text{sched}(k) \) in which \( k \) is the index of a processor. \( \text{sched}(k) \) is a periodic schedule for processor \( k \). Based on [28] if \( \text{PASS} \) with the smallest period has a period \( P_s \) then the period for the parallel schedule, \( P_p \), is a multiple of this period:

\[
P_p \leq JP_s
\]  

(1.12)

The reason to choose a multiple of sequential period length is to minimize the average time

\[
P_p / J
\]

for \( P_p / J \) by interleaving sequences. To illustrate this an example SDF is given in Fig. 1.8(a) with runtime of 1 unit for actor 1, runtime of 2 units for actor 2 and runtime of 3 units for actor 3. A PAPS schedule for two processors is with \( J = 2 \) which is depicted in Fig.1.8(b).

In formal notation it is defined as:

\[
\text{sched}(1) = \{3,1,3\}
\]  

(1.13)

\[
\text{sched}(2) = \{1,1,2,1,2\}
\]  

(1.14)

Which has a length of 7 units. This schedule is determined by drawing precedence graph (Shown in Fig. 1.9) of two executions of the graph. Knowing this much about the scheduling procedure of SDFs up to this level, suffices for the purpose of this document. For further information on parallel scheduling of dataflow models, more information can be obtained from researchers done in [28, 17].

### 1.2.2 SDF Buffering Technique

In an SDF, if not enough tokens are on the arc to an actor, the actor cannot be invoked. Therefore, those tokens should remain on the arc until they are in sufficient quantity. This is why buffers are needed in SDF. Typically, if FIFO buffers are used, as mentioned in the previous part, minimum size of the buffer can be determined through static scheduling
process. But in fact, FIFO buffers are costly and not efficient. Therefore, it is wise to know how buffers can be allocated in a more efficient way.

Lee [29] uses shared memory as buffer container to propose FIFO queues with less overhead. In his buffering scheme each node uses fixed positions in the memory to read from and write to. This scheme is known as static buffering.

To clarify static buffering, consider the simple model in Fig. 1.10 in which the schedule is known to be \( \{A_1, A_2, B_1, A_3, B_2\} \). If \( A \)'s outgoing arc buffer size is 4, in the first cycle of the schedule, \( A_1 \) writes in positions 1 and 2, \( A_2 \) in 3 and 4 and \( A_3 \) again in 1 and 2. In the second cycle, \( A_1 \) should write in positions 3 and 4 which is different from its first cycle. It means that with buffer size 4, static buffering cannot be used for this example. But apparently, if buffer size is set to 6 then static buffering can be used. In that configuration \( A_1 \) always writes in positions 1 and 2, \( A_2 \) in 3 and 4, \( A_3 \) in 5 and 6.

The size of a buffer \((N)\), in order to do static buffering is determined by this equation: [29]:

\[
iq = KN
\]

In which \( i \) is the number of tokens produced/consumed in each invocation of the node (for
the above example it was 2), \( q \) is the number of invocations of an actor in a cycle of the schedule (for \( A \) above it was 3), \( N \) is the size of the buffer and \( K \) is some integer. This equation can be defined also as:

\[
 iq \mod N \equiv 0
\]

which trivially should be true for a static buffering scheme to work. If the first equation is re-written then:

\[
 N = \frac{iq}{K}
\]

And if the values of the above example are used:

\[
 N = \frac{6}{K}
\]

In which \( N \) should be an integer. Therefore, possible values for \( K \) can be 1, 2, 3 and 6. Determining the correct buffer size is however not easy. Lee [29] uses Gabriel (their implementation for SDF models) to ignore unacceptable values. In the above mentioned example, since node \( B \) can only read from the buffer after two invocations of \( A \), sizes 1, 2 and 3 cannot be the correct answer. Therefore, size 6 is the correct answer for the example.

Obviously, static buffering can be used in shared-memory model systems and results in better analyzability and debugging.

### 1.3 SDF Relaxations

In an SDF model each actor is assumed to consume and produce the same amount of tokens in all firings. This assumption restricts the expressiveness of a dataflow model. In practice many applications cannot be modeled with SDF models. SDF models do not cover the cases in which an actor is data or state dependent. A data-dependent actor produces a subset of its outputs based on the incoming data and a state-dependent one does that based on its internal state. For example, a program contains an if/then/else branch it cannot be modeled by an SDF model.

In order to overcome the shortcomings of SDF models two relaxations can be defined. The first relaxation is to use a sequence of consumption/production rate instead of single constant value. The second relaxation is to add asynchronicity to dataflow models. The first relaxation is called CSDF (Cyclo-Static Dataflow) and the second is called ADF (Asynchronous Dataflow). The relation of these two relaxations with SRDF and MRDF is depicted in Fig. 1.11. ADF is the most relaxed dataflow model and SRDF is the most restricted one.

In the rest of this section CSDF and variations of ARDF are explained.

#### 1.3.1 Cyclo Static Dataflows (CSDF)

Cyclo-static dataflows (CSDF) are a variation of SDFs [22] in which consumption and production rate of tokens cycle through sequences of number, hence the term ‘cyclo’. It is proved that they allow static scheduling in compile-time while providing cyclically changing behavior [34].
An example of a CSDF is depicted in Fig. 1.12(a) in which consumption and production rates are repeated in each cycle. In each cycle and in the first invocation the actor receives 1 element and produces 0 output. In the second invocation, it receives 2 and produces 1. In the last invocation of a cycle it consumes 1 and produces 0. This behavior is then repeated in all cycles. This CSDF model can be simulated as an SDF like in Fig. 1.12(b).

![Figure 1.11: The relation between dataflow categories](image)

**1.3.2 Asynchronous Dataflows (ADF)**

ADF models are a class of dataflow models in which asynchronous actors are also involved in a model. Asynchronous actors behave differently in different invocations based on the inputs they receive. This relaxation permits data and state dependent actors to be in a dataflow model. Since ADF models behave differently in each invocation, ADF require dynamic scheduling. As mentioned before, dynamic scheduling results in a large run-time overhead and might require unbounded buffers [34]. For this reason ADF models are referred in some contexts as Dynamic Dataflow (DDF) models.

ADF models have different variations and are the most practical variations of dataflow models. Among them two subcategories are used in this document for the proposed models in Chapter 4; BDF (Boolean Dataflow) and VRDF (Variable Rate Dataflows).

**1.3.2.1 Boolean Dataflows (BDF)**

BDF models like other ADF models behave differently in different invocations and should be dynamically scheduled. However, their difference in behavior is restricted to two beha-
viors, hence the term boolean. Two important examples of BDF actors are *Switch* and *Select* (Fig. 1.14). In the Switch (Fig. 1.13(a)), based on the control input, the actor sends data to only one of its outgoing arcs. In the Select (Fig. 1.13(b)), based on the control input, the actor receives data from only one of its incoming arcs.

Although static schedulability is not possible for BDF models, they are Turing complete and can cover all programs that can be modeled in any other way [33]. Nevertheless, since actors are behaved as blackboxes in dataflow models, automatic tools can not prove questions like consistency and scheduling in bounded time for BDF models. To overcome this problem CDDF (Cyclo-Dynamic Dataflow) is introduced in [33] which is a relaxation of CSDF by embedding BDF properties. In CDDF the knowledge inside actors is also available to automatic tools which brings about more analyzability of CDDF models. By using CDDF consistency of an asynchronous dataflow model can also be proved [33].

![Switch and Select actors](image)

(a) *Switch* actor - based on the value of control signal it sends output to one of its outgoing arcs
(b) *Select* actor - based on the value of control signal it consumes a token from one of its incoming arcs

Figure 1.13: Two important BDF actors: *Switch* and *Select*. They behave differently based on the data which is received from the control unit arc

### 1.3.2.2 Variable Rate Dataflows (VRDF)

In VRDF an actor like an actor in BDF receives a control input before receiving its real data inputs. However this input is used to determine how many elements should be consumed from an incoming edge (Fig. 1.14(a)) or should be produced on an outgoing edge (Fig. 1.14(b)). It is proved that if the range of control input values is bounded, buffer sizes can be determined [35].

**Summary**

Dataflow models are abstract programming models which are widely used in embedded parallel systems. On one hand, they can explicitly show parallelism in a model and on the other hand they are abstract enough to hide implementation details from a parallel program. This chapter explained different variations of dataflow models: pure dataflows, SDF, CSDF,
Figure 1.14: Two types of VRDF actors: Consumer and Producer. They consume and produce different number of tokens based on 'size' control input.

BDF, VRDF. These models are used to define a practical example of a parallel application in Chapter 4.
Chapter 2

LIME - Less Is MorE

For all software programs always two ends can be defined. One end is the front-end which
is typically a model used to visually represent an application and the other end is the back-
end which is a complex, non-abstract programming language used to run the application on
a specific hardware.

In parallel systems, front-end is usually modeled by models like Dataflow, Series-Parallel
graphs, KPN (Kahn Processor Networks), etc. and does not change frequently for a specific
application. Nevertheless, on the other end, parallel hardware are on their evolution path
and change almost rapidly. This change causes back-end (programming languages being
used) to change more frequently, resulting in a conflict between the rate of changes in the
two ends.

The incompatibility in change rate of the two ends urges the programmer to re-write the
same application or at least refine the same application to get a faster application for each
release of new hardware and programming language. This is indeed a tedious task which
slows down the progress in scientific applications greatly and must be avoided.

To overcome the incompatibility there should exist a middle-end between the two ends
which converts the same model to several ever-changing back-ends in a generic way. This
middle-end is then able to translate the same model to ever-changing back-ends regardless
of what the model is.

Such medium is provided by a programming model called LIME (Less Is MorE) [26].
LIME middle-end with its modeling environment LIMEclipse (Chapter 3) makes it possible
for the programmer to define an application once in a model and then convert it to several
backends (Fig. 2.1).

To define LIME in more detail, in Section 2.1 LIME syntax and elements are defined.
Section 2.2 describes compilation flow of LIME and the programs used in its tool-chains.
Then in Section 2.3, two approaches in LIME for parallel program analysis are described.
Finally, in Section 2.4, backends of LIME are mentioned briefly.
2.1 Definitions

LIME is a compiler which uses an application model and converts it to several parallel backends. Application model can be defined in a relaxed version of Dataflow that can be further converted to pure dataflows, SDF (Synchronous Dataflows), CSDF (Cyclo-static Dataflows) and SP (Series-Parallel) graphs for analysis purposes. The model can be defined by pure XML manually or by models created by LIMEclipse plug-in introduced in Chapter 3.

A LIME model (Fig. 2.2) consists of the following elements:

**Component** - also known as actors, computation kernels or lime are computation units which contain a program in ANSI C (ISO C99). Each component has return type, name, and can also nest other components. Components can be either a class or an instance. An instance component is an instance of a class component. Only instance components can be activated in the program.

**Ports** - ports are used in LIME to connect different components together. They can be either input or output port. Each port has a size attribute. Size of a port shows its consumption (if it is an input port) or production (if it is an output port) rate of tokens. A port (input or output) can have a set of attribute which are specified by ‘altered C modifiers’:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>With</th>
<th>Without</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>It is an input port</td>
<td>It is an output port</td>
</tr>
<tr>
<td>static</td>
<td>‘Port waits for ‘size’ elements before invocation (called blocking or SDF port)”</td>
<td>Consumes/Produces either R or 0 elements (called non-blocking or optional port)</td>
</tr>
<tr>
<td>restrict</td>
<td>All ports are non-overlapping</td>
<td>Ports overlap</td>
</tr>
<tr>
<td>volatile</td>
<td>Port data is shared</td>
<td>Port data is not shared</td>
</tr>
</tbody>
</table>

Table 2.1: Port modifiers

For example ‘const static restrict input[10]’ is an input SDF port with size 10 which its data does not overlap with other ports.

The above information about port modifiers is given for better understanding of the final
models. For more information about these properties one can refer to LIME documentations.

**Arcs** - ports can connect to each other via arcs. An arc can have different types. A few possible values are:

- **state** - feeds internal state of a component to itself
- **init_state** - is used by a constructor to feed a component buffer at time 0 (at start-up)
- **deinit_state** - is used by a destructor to free up the used memory at clean-up
- **fifo** - the buffer used on an arc is accessed as a FIFO buffer
- **init_fifo** - puts element in a fifo at start-up
- **deinit_fifo (Blackhole)** - consumes elements in a fifo buffer
- **fifo:scalar_operation** - is grouped with other fifo:scalar_operations arcs connected to a single port and asks the port to evaluate the reduction result of the elements in the group. The evaluation is done in a LIME native method and is invisible to the programmer. ‘scalar_operation’ on these arcs can be any reduction operation e.g. ‘+’, ‘*’ and can be even non-commutative e.g. ‘-’. However, it should be the same for all the arcs which are connected to the same port.

If an arc connects two components with different production and consumption rates, LIME instantiates as many of the one with the smaller size to match the bigger size. This is the main source of data level parallelism in LIME [27] e.g. in each invocation of ‘source’ in the example in Fig. 2.2 two instances of ‘copy’ are created.

**Associations** - associates a port (input or output) to a port of the same type and in a deeper level in a nested component.

A LIME model can be defined in GXF which is based on GXL (Graph Exchange Language) which is nowadays a rather standard XML representation for graphs. Models which are created by LIMEclipse GUI are also translated to this standard directly which provides access to the graphical and textual format at the same time.

In addition to the elements of a model a set of rules can be defined for a LIME model [27]:

- **fireability** - a component is fired (its corresponding C code is executed) whenever all its inputs are ready and enough space is available on the output buffers. This is indeed a stricter rule for an actor to be fireable than in a pure dataflow model.
- **consistency** - a deactivated component flushes its inputs and outputs, preventing further reads and writes before its next invocation
- **independence** - a fireable component fires regardless of the status of the other actors. This characteristic depicts locality of effect and transparency properties mentioned for a pure dataflow in Chapter 1. History unawareness allows instruction level parallelism within a component in addition to the global parallelism in the application.

Visualization software of LIME called LIMEclipse (Chapter 3) uses these definitions and rules to show LIME models. Based on these models an abstract reduction model is defined
in Chapter 4. In the next sections it is described how these models can be used to produce an efficient parallel code.

2.2 Compilation Flow

Compilation in LIME starts with a compiler-driver (currently implemented as a script) which consists of several processing stages. In a high-level view, it does the followings [27]:

- builds platform-specific shells
- creates Operating System configurations
- makes the binary file to run the program on a specific parallel backend

In a more precise definition ‘slimer’ compiles a LIME program in 5 steps [27]:

1. **Front-end Parsing** - using GCC, it creates a machine readable XML file based on the given C codes in the components and the GXL file. To avoid writing a new parser for C applications LIME uses `-fdump-translation-unit` switch in GCC.

2. **Middle-end (ME) static analysis and scheduling** - groups actors together to be assigned to different parallel tasks and schedules those tasks. Several methods are used for scheduling e.g. if the graph is an SDF, SDF scheduling methods (as defined in Chapter 1) can be used.
3. **Back-end code generation** - generates final parallel system specific codes (CUDA, Pthreads and NXP proprietary language so far) and produces initializer scripts and OS configurations.

4. **C tool-chain** - compiles computation kernels in components and performs optimizations if possible.

5. **Profiling and simulation** - performs simulation tests and provides feedback to the middle-end for better scheduling and grouping for the rest of the application.

This five steps can be seen in Fig. 2.3.

![Figure 2.3: Compilation flow of LIME (Fig. from [27])](image)

### 2.3 Analysis Models

There are two different approaches used in LIME for model analysis: *SDF* and *SP*. When an application model is an SDF, SDF scheduling is used. As mentioned in Chapter 1, SDFs can be statically scheduled. With static scheduling time constraints and buffer sizes can be determined and the behavior of a system can be easily predicted. In addition, deadlock-freedom, latency bounds and expected throughput can be guaranteed [26]. But SDF models are not always expressive enough. They assume that each component always produces and consumes the same number of tokens which is not the case in many practical applications. Typically, a component’s consumption and production rates vary based on the incoming data. In such cases SP analysis is used [26]. SP analysis creates series/parallel graphs for an application. SP graphs are special cases of *k*-trees. Such graphs can simplify scheduling and resource allocations to polynomial complexity. However, in a LIME dataflow model, each of these two analysis models are used when appropriate.

Analysis and scheduling are the key parts of LIME which can highly affect its performance and its success. Currently, scheduling is done either by proprietary NXP scheduler (currently co-owned by STE and NXP) or in runtime (for Pthreads back-end). By performing scheduling via LIME model, a programmer focuses only on his computation purposes not on hardware specific issues.
2.4 Supported Backends

Currently, LIME supports three backends: a proprietary NXP parallel programming language, Pthreads, and CUDA. Among these three, NXP parallel programming language is not open-source and is not described in this section.

Pthreads or ‘POSIX threads’ is a standard for threads which defines an API (library) for using and programming threads. Pthread is both used in Windows and Linux to exploit thread level parallelism explicitly. It is proved that Pthreads has better memory-to-CPU bandwidth than MPI shared-memory applications [13]. Pthreads is a good choice where different kernels in a program are doing different tasks (MIMD) but not the same task on different data (SIMD) [13].

LIME is tested for Pthreads code generation in [27] for a Digital Audio Broadcast application. The experiment shows that LIME can produce Pthreads code which work as efficiently as the manually written code while being easier to write the program, having smaller code size and being easier to maintain.

CUDA is a general purpose parallel computing architecture which is used over NVIDIA video cards or an emulated videocard. As described earlier, it is an extension to C and uses a set of new constructs and qualifiers for defining parallel applications. CUDA uses _device_ qualifier for kernel functions which is similar to components in LIME model [26]. In both models computation kernels produce/consume data tokens and transfer information between the kernels. The benefit of LIME over CUDA is that in CUDA separate codes must be defined for host (CPU) and device (videocard), and in LIME a single code is written by the programmer and the rest of the work is assigned to LIME. This is a level of abstraction over CUDA. LIME uses a proprietary scheduler called heracles to schedule computational kernels on a videocard of CUDA architecture.

Although, generating code from a model by LIME is optimized through its development, it is apparent that fully optimized code for a specific backend is most of the time not achieved through LIME code generator. But considering complexity reduction of code (it is reported for Pthreads in [26]) in most of the cases and increased maintainability can pay-off for this performance decrease.

Summary

LIME is a programming model which is used to create an abstraction layer over existing parallel programming libraries. Using a graphical model created by LIMEclipse (Chapter 3), LIME creates codes for several backends like Pthreads, CUDA and an NXP proprietary language from a single model. This makes a standard way of programming for parallel systems based on dataflow models. The benefits of using LIME are lower code complexity and relieving a programmer from tedious and platform specific tasks such as synchronizing the threads, thread creation, etc. while keeping expressiveness and parallelism in a model. This chapter discussed about LIME and its various properties. This knowledge is used in
the next chapters to first define a GUI for LIME and then to introduce practical examples of using LIME.
Chapter 3

LIMEclipse - LIME GUI Plug-in for Eclipse

As mentioned in the previous chapters, the main thread of this document is to provide abstract models (dataflow models) for parallel applications that can be mapped to different parallel back-ends. It was also noted that fulfilling this goal is only possible by defining a software tool-chain with two ends. One end was discussed in Chapter 2 which performs the mapping from application models to different parallel backends. The other end visualizes parallel application models in order to reduce the complexity of such models. This end is named as ‘LIMEclipse’.

LIMEclipse’s design is based on the requirements and the rules which are dictated by the LIME programming model. These requirements and rules were exploited by the author to evaluate the existing graphical frameworks based on their features. The evaluation led to choose the architecture of LIMEclipse and a graphical framework called GEF as the base of development.

In order to browse the evaluation phase and development process of LIMEclipse this chapter explains LIMEclipse in more details. First, in Section 3.1 the necessity and the requirements of a rich LIME (dataflow) model graphical editor are explained. Then in Section 3.2 the existing graphical frameworks are mentioned and compared with each other in order to find the best choice as the base of development. In Section 3.3 the chosen framework GEF is explained. Section 3.4 then describes the LIMEclipse plug-in and its various aspects in more details. Finally, Section 3.5, defines the future path of the LIMEclipse plug-in.

3.1 The Motivation for a Rich Graphical Editor

Before starting the development process of a rich graphical editor for LIME, it is wise to investigate why such a graphical editor is needed at all. In order to visualize the problem, consider drawing a dataflow model for a parallel application on a piece of paper. It is as simple as drawing boxes and connecting them together with lines. Even if a dataflow model is large, investigating the model and changing it on paper is as easy as deleting/adding
edges and boxes. However, when you try to define the same dataflow model based on textual representations e.g. in LIME way, the model becomes complex after adding just a few nodes. The reason is that you have to care about ‘rules in your models’, ‘valid hierarchies in your models’, etc. in a non-visual way. This complexity confines LIME usefulness in many practical applications, since in such cases keeping LIME model interactions valid would be a difficult task. This outcome repels people from adapting to LIME at all. Therefore, a graphical editor for LIME should overcome the problem by simplifying applications to the extent of drawing dataflow models on a paper. Such a graphical editor helps in investigating LIME usefulness for practical parallel applications and in testing theoretical dataflow models in practice.

However, making a graphical editor for LIME similar to what you draw on a piece of paper is an intricate software development. Such software should at least address three things. First, defining all the valid elements in the diagram and introducing them to the software (Model). Second, depicting the elements by visual shapes (View). Third, informing the software to impose LIME rules e.g. a port can only be added to a component, on the elements that are drawn and relating the visual shapes to the elements (Controller). These three compartments dictate that such a graphical editor must follow MVC (Model/View/Controller) pattern for its architectural backbone. Using MVC pattern shows the overview for the desired software and draws a guideline for the development. In addition, to clarify a more detailed requirements at least the following criteria are defined:

- **textual representation of LIME models** - since models are used by groups of people, the desired software is expected to provide textual representation of models. In that way, versioning systems like SVN, Mercurial, etc. can be used to provide team-work on a single model. The proposed standard is XML which is widely used and standardized.
- **ability to show and edit large graphs** - the desired software should be scalable and be able to show and edit large graphs
- **ability to automatically layouting a dataflow model** - the desired software should have the ability to position elements in the least crossing between the edges automatically.
- **support for direct compilation of LIME applications** - since LIME applications are C-based applications, it is expected that the desired software support compilation of C files inherently.
- **birdseye/fisheye for an application** - since LIME-based applications can be large in number of nodes, it is expected that the desired software have the ability of focusing on a single part of a diagram
- **platform independence** - the desired software should be runnable on most of the known operating systems.

Based on the MVC pattern and the above mentioned criteria the next sections explain different choices for the base framework for LIME graphical editor and why one is selected as the base of the development.
3.2 Graphical Frameworks

Nowadays, too many graphical frameworks exist which provide wide range of graphical features. Therefore, it is not wise to build a new framework from the scratch in order to satisfy the previously mentioned criteria for a LIME graphical editor. However, to select one of these frameworks as the core of LIME graphical editor, this must be taken into account that such a framework must be able to at least satisfy most of the criteria mentioned in the previous section. In addition, since the resulting graphical editor is going to be an open-source project, it is wise to start with an open-source graphical editor framework.

In this section, a number of such graphical frameworks are investigated and their features are compared with each other.

3.2.1 Existing Frameworks

mxGraph [11] from JGraph Ltd., is a non-free Javascript library which supports large graphs. It uses its own file formats and is not easily extensible 3.1(a). It does not have separate layers for model and view. JGraph Ltd. also has a Java-based editor called JGraph [7]. The free version is called JGraph and the non-free one is called JGraphPro 3.1(b). JGraph and JGraphPro both support ‘.dot’ files which are secondary standard for LIME model representations.

yEd [15] graph editor 3.2, a widely used graph representation tool is a framework from yWorks company with special features like layouting and graph analysis. yEd’s Java-based editor is a free application, but analysis and special layouting tools are packaged in another tool called yFiles which is not free. In spite of its wide range features, yEd is not used as
LIME GUI editor since it does not support C projects easily and is not open-source. It is indeed a graphical editor which can be used to draw diagrams and is not intended to be glued to other tools. But it supports all visual features that LIMEclipse needs.

CoG [6] is a simple light-weight graph editor 3.3 with minimal features. The only advantage of CoG is that it can export the diagrams to `.dot` which as mentioned before are the second standard for LIME models.

In addition to the above frameworks, there are three other frameworks which are more interesting; Piccolo [12], standalone GEF [3] and Eclipse GEF [4].

Piccolo is a structured 2D graphics framework. It is provided with a lot of built-in features. It is aimed to be used by Java and C# programs to create rich graphical editors.
Piccolo has many demos in which most of the features needed for LIME GUI can be seen (Fig. 3.5 and 3.6).

Standalone GEF is from an open-source community and is based on Java Swing. ArgoUML [1] (Fig. 3.4) a well-known open-source UML diagram editor is an example of GEF standalone framework. It also supports many of the features needed for LIME GUI and is a good candidate. Early developments of LIME GUI is done based on this GEF which can be obtained from [8].

Eclipse GEF, on the other hand, is based on Java SWT. Eclipse GEF is a project by a team different than the standalone version. Programs based on Eclipse GEF are in fact plug-ins for Eclipse RCP (Rich Client Platform). It means that a Eclipse GEF plug-in can be used with Eclipse C/C++ platform as well. In addition Eclipse GEF uses MVC pattern which separates model from visual representation. A good example of GEF Eclipse plug-in which covers a lot of features needed by LIME GUI is circuit example shown in Fig. 3.7.

With knowing all the possible candidates for LIME GUI, it is wise to compare their
features together. This comparison is done in the next section.

### 3.2.2 Features Comparison

Features provided by each of the frameworks mentioned above are included in the following tables. In Table 3.1 the editors are compared based on the file-types (standards) they support, the language they use and also the date of their last release. Release date was an important factor that could affect the choice among these frameworks for LIME GUI, since unsupported software could result in time-consuming and non-traceable bugs. Table 3.2 compares graphical features of the frameworks. Table 3.3 gives a conclusion about the editors benefits and proprietary usage info of the frameworks.

<table>
<thead>
<tr>
<th>Name</th>
<th>Supported File Formats</th>
<th>Export to</th>
<th>Based on</th>
<th>Last Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>mxGraph</td>
<td>xml</td>
<td>.pdf, .xml, .jpg, .png</td>
<td>Javascript</td>
<td>2007</td>
</tr>
<tr>
<td>JGraph</td>
<td>.xml, .ser (Its own Standard)</td>
<td>.xml, .ser</td>
<td>Java</td>
<td>20-May-08</td>
</tr>
<tr>
<td>JGraphPad Pro</td>
<td>.gxl, .csv, .dot (GraphViz), .xml</td>
<td>.dot(GraphViz), .svg, .pdf, .image, .gxl, .eps</td>
<td>Java</td>
<td>22-May-08</td>
</tr>
<tr>
<td>yEd</td>
<td>.gml, .graphml, .graphmlz, .tgf, .xgml, .ygf, .gxl, .xml</td>
<td>.images, .svg, .xml, .html, .gml, .graphml, .graphmlz, .xgml (GML/XML format), .ygf</td>
<td>Java and .Net</td>
<td>30-May-08</td>
</tr>
<tr>
<td>CoG</td>
<td>xml</td>
<td>.xml</td>
<td>Java</td>
<td>2-Apr-05</td>
</tr>
<tr>
<td>Piccolo</td>
<td>Not restricted, it is a graphical framework</td>
<td>Not restricted, it is a graphical framework</td>
<td>Java and .Net</td>
<td>27-Jul-08</td>
</tr>
<tr>
<td>GEF (Standalone)</td>
<td>Not restricted, it is a graphical framework</td>
<td>Not restricted, it is a graphical framework</td>
<td>Java</td>
<td>19-Jul-08</td>
</tr>
<tr>
<td>GEF Eclipse</td>
<td>Not restricted, it is a graphical framework</td>
<td>Not restricted, it is a graphical framework</td>
<td>Java</td>
<td>9-Jun-08</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of graph editors based on the file-types they support, the language they are based on and the last time that they have been updated.

<table>
<thead>
<tr>
<th>Name</th>
<th>Editor Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>mxGraph</td>
<td>+ + + + +</td>
</tr>
<tr>
<td>JGraph</td>
<td>+ + + + + + - - - -</td>
</tr>
<tr>
<td>JGraphPad Pro</td>
<td>+ + + + + + BE</td>
</tr>
<tr>
<td>yEd</td>
<td>+ + + + + + BE</td>
</tr>
<tr>
<td>CoG</td>
<td>+ + + + + BE</td>
</tr>
<tr>
<td>Piccolo</td>
<td>+ + + + + + BE</td>
</tr>
<tr>
<td>GEF (Standalone)</td>
<td>+ + + + + - BE</td>
</tr>
<tr>
<td>GEF Eclipse</td>
<td>+ + + + + BE</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison of graph editors based on their features as an editor

According to the above information, Eclipse GEF supports the widest range of features need by LIME GUI and it is highly supported by Eclipse community. On the other hand, it uses MVC pattern which separates model from view. Eclipse GEF can be used with Eclipse...
Table 3.3: Comparison of graph editors based-on their requirements and terms of use

<table>
<thead>
<tr>
<th>Editor</th>
<th>Free/Open-source</th>
<th>Requirements</th>
<th>Proprietary Application Usage</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>mxGraph</td>
<td>-</td>
<td>Java 5.0+, .Net 1.1+, PHP 5.0+</td>
<td>Only if purchased</td>
<td>Javascript library, member of JGraph Ltd, it is called as JGraph inside the browser, XML data</td>
</tr>
<tr>
<td>JGraph</td>
<td>+</td>
<td>Java 1.4</td>
<td>Free</td>
<td></td>
</tr>
<tr>
<td>JGraphPad Pro</td>
<td>-</td>
<td>Java 1.4</td>
<td>Free</td>
<td>JGraphPad Pro is a free version of JGraph</td>
</tr>
<tr>
<td>yEd Graph Ed-</td>
<td>+</td>
<td>Java 5.0+ and .Net 1.1+</td>
<td>Free</td>
<td>Java-based, it has a analyzer and tester which is not free, it has predefined layouts</td>
</tr>
<tr>
<td>CoG</td>
<td>+</td>
<td>Java 1.4+</td>
<td>Free</td>
<td>Simple to use and lightweight and works flawlessly, it has predefined layouts</td>
</tr>
<tr>
<td>Piccolo</td>
<td>+</td>
<td>Java 1.4+ and .Net 1.1+</td>
<td>Free</td>
<td>Ongoing project, full set of features, but features are not included as one application</td>
</tr>
<tr>
<td>GEF (Standalone)</td>
<td>+</td>
<td>Java 5.0+</td>
<td>Free</td>
<td>Argouml is an example of GEF as a standalone application and provides a wide range of features</td>
</tr>
<tr>
<td>GEF Eclipse</td>
<td>+</td>
<td>Java 5.0+</td>
<td>Free</td>
<td>It is configurable and a lot of examples and documents exist for that</td>
</tr>
</tbody>
</table>

C/C++ IDE. Therefore, it is chosen as the basis of LIME GUI development. Eclipse GEF is defined in the next section in more detail.

### 3.3 Eclipse GEF - Graphical Editor Framework

![GEF components and dependencies](image)

Eclipse GEF is a graphical editor framework which allows a programmer to create graphical editors with the least effort needed. It uses MVC pattern to separate model and visual representation. Model and view in GEF talk to each other through GEF framework. This makes modification to a graphical program easy.

Like other Eclipse libraries GEF relies on RCP (Rich Client Platform) libraries and Draw2D (Fig. 3.8). Draw2D is a lightweight layout and rendering library from Eclipse community which is used to visualize model elements in a GEF-based application. Draw2D itself uses SWT to display graphical elements (Fig. 3.9), like Swing uses AWT to display graphical shapes.

Draw2D uses a set of rules and concepts. First of all, it uses a tree to model elements in a diagram in which the root element is the diagram itself (Fig. 3.10). Draw2D determines
order of drawings in the tree by painting from left-to-right. Each node clips the visible area and hides anything outside the clipping area (e.g. in fig 3.10 box 6 is clipped).

Draw2D uses four Layout Managers to display the elements inside a node (a nested element). They are shown in Fig. 3.11. As an example, XYLayout is used in LIMEclipse to view nested components.

Draw2D then connects shapes in the diagram through GEF framework to the model used in an application (Fig. 3.12). When a change is requested in the model it is reflected in the visual representation via sending commands.

Editparts as is seen in the next section are the points in which a programmer defines extra actions when a change is occurred in the model or the view. Editparts are used in LIMEclipse to propagate changes on child elements.
Using Draw2d, GEF framework provides the following visual features [4]:

- **Tools like Selection, Creation, Connection and Marquee tools**
- **A Palette for displaying those tools**
- **Handles for resizing objects and bending connections**
- **Two types of GEF Viewers (Graphical and Tree)**
- **A Controller framework for mapping the business Model to a View**
  - Plug-in policies for mapping interactions with the view to changes in the model
  - Various implementations for showing feedback and adding selection handles
  - Various request types and tools or actions that send these requests to the controllers
- **Undo/Redo support via Commands and a CommandStack**

In addition, GEF makes it possible for a programmer to configure and extend GEF for new features or adding pre-developed features in other GEF-based application to his application. In fact, by creating new GEF projects, one can add graphical editors to Eclipse environments or in a more generic term to Eclipse RCP (Rich Client Platform). LIMEclipse, visual editor for LIME which is defined in the next section, describes the usage of GEF framework in more detail.

### 3.4 LIMEclipse In Details

LIMEclipse is an Eclipse plug-in used as the visual editor for LIME models. Like other Eclipse plug-ins, LIMEclipse is based on Rich Client Platform (RCP) plug-ins which are the minimal set of plug-ins needed to build a rich client application. In addition it is based on two other plug-ins by Eclipse; GEF (Graphical Editing Framework) and Draw2d [4] from Eclipse community.

LIMEclipse design is based on ‘Logic’ and ‘Schema’ examples by Eclipse community. Logic example (Fig. 3.7) is a circuit designer with **AND** and **OR** gates, **LED displays** and
circuits. It supports nested circuits and connection endpoints which are similar to components and ports in LIME. Schema example uses automatic layout for putting tables in a database schema in a good looking way on a page which is used for top-level layouting of the components in a LIME diagram.

In order to browse LIMEclipse software design, the rest of this section describes LIME in more detail. First, elements of MVC pattern used in LIMEclipse are defined: ‘model, view and controller’. Then the layouting algorithm for the components in a diagram is defined. After that connection point of LIMEclipse and LIME framework is described. Configuration and settings of LIMEclipse are then discussed next. Finally, extension points and configurability for the plug-in are defined in this section.

3.4.1 Model

![Figure 3.13: Newer LIME GUI based on Eclipse GEF with all supported elements](image)

Model classes in LIMEclipse are indeed elements which represent LIME models. They hold the properties of each element of a LIME diagram. Elements in a LIMEclipse diagram are (Fig. 3.13):

- **component class** - a gray box which holds information and structure of a component and is only visible in *class mode* of a diagram
- **component instance** - a gray box which is an instance of a component class. It is read-only and is only visible in *instance mode* of a diagram
- **input port** - a green rectangle which contains information about an input port, e.g. *port size*, *restrict*.
- **output port** - an orange oval which contains information about an output port, e.g. *port size*, *restrict*.
- **code fragment** - a note shape which can be used instead of C files for the components.
- **arc** - a solid directed edge which connects two ports together, providing data from the head to the tail
- **association** - a dashed bidirectional edge which associates two ports together
The classes which represent these elements form the class diagram which is depicted in 3.14. In the class diagram, abstract classes are shown in italic. In the diagram, LIME model elements have a common parent called *LimeSubpart*. LimeSubpart holds all the common information, e.g. name, location, size, etc. LimeSubpart is itself the children of LimeElement which keeps templates for functions used for property change propagation to the controller classes. This hierarchy uses *composite structural pattern* in which each ‘LIMEDiagram (LIMEComponent)’ can have a set of ‘LIMEElement’s.

As can be seen new model elements can easily be added to the diagram by deriving a new class from LIMESubpart. New model classes can be further coupled with new figures which would add new features to the diagram.

### 3.4.2 View - Figures in the Diagram
In LIMEclipse figures are grouped in almost the similar hierarchy as the model classes (Fig. 3.15) with having *Figure* class as the parent of all figures. *Figure* class is a class in Draw2D packages which provides common properties for all shapes. As mentioned before Draw2D uses a Figure instance to first draw the shape in its specified canvas. Then it starts in children tree of the figure and starts painting elements following LTR traverse in the tree.

This hierarchy easily allows drawing new figures. New figures can be created by extending *figure* class or a more concrete classes like *NodeFigure* class which is more LIMEclipse specific. Such new figures could further be coupled with model elements.

### 3.4.3 Controller and Policies Packages

Controller classes (Fig. 3.16) are the heart of LIMEclipse. They connect model and view classes together and reflect changes from one in another. If a property is changed in the model (e.g. name property of a model), they can reflect the change in the figure and repaint the shape (using *refreshVisuals()* method).

![Figure 3.16: Class diagram for controller classes in LIMEclipse](image)

### 3.4.4 Layouting

Layouting in LIMEclipse is based on layouting example of GEF ‘Schema Diagram Editor’ [2]. SchemaEditor uses predefined Draw2D layouting algorithms for fully connected graphs and puts tables in a way that relation arcs between those elements have minimum overlapping.

To perform layouting, first a graph should be created in Draw2D standard using *Node* and *Edge* classes. Then using Visitor pattern dummy nodes are created and nodes are clustered which makes layouting easier. The benefit of using Visitor pattern is that it separates an algorithm from a model upon which it operates. The clustered graph is then used by a Draw2D graph layouting algorithm to put each node in a possition with the minimal overlapping of the edges.
Layouting in LIMEclipse is performed in two consecutive phases. First, layouting is done for the top-level components in a diagram. It means that in the resulting graph, nodes are the top-level components and edges exist between two nodes when two components’ interior elements have an arc in between. The clustering heuristic in this phase is to cluster components which have the same name prefix. This is done in purpose to show cloned components (which have similar names) near together. Then layouting is performed for the clustered graph. Second, layouting is done recursively for the elements inside each component. In this phase the heuristic is to cluster input ports, output ports and nested components, separately. This heuristic puts input and output ports in most of the cases in the best layout. Obviously, better heuristics can be used to improve layouting of LIME diagrams which is also defined as future works in the end of this chapter.

### 3.4.5 Connection to LIME - XMLFactory

In the earlier releases of LIMEclipse the XML format was considered as a format which the diagram should be exported to. Nevertheless, from the moment that the plug-in was intended to be used with the Eclipse C/C++ IDE, it was decided that the XML format is to be the standard saving format for diagrams. The benefit of Eclipse C/C++ is that one can build and run his LIME codes while having the visual editor at the same time.

In the current release of LIMEclipse (v3.1.0), XMLFactory is responsible of converting XML files to LIMEclipse object models and vice versa. The former is done via `getDiagramForXML` and the latter by `getXMLForDiagram` methods. An XML file is used in LIME-based applications and a LIME diagram is used in the graphical editor.

XMLFactory is used by LIMEEditor to sync the XML and the visual representation of a diagram. Any change in one representation is reflected in the other view at the same time. This makes usage of both representations easier. The programmers who prefer using XML representation can use the textual representation and use the plug-in only to get an overview of their program.

Obviously, if XMLFactory generated files are not compatible with LIME, the plug-in would be useless. Therefore, possible common textual changes are pulled out from XMLFactory and are put in `system.properties`. For further information one can refer to these methods and the associated Javadocs or to the appendices.

XMLFactory uses a error-correcting approach. If for any reason an XML file cannot be read, the whole application would work but an empty diagram would be created. This would make the application more satisfactory and user friendly for the users.

### 3.4.6 Configurability and Extension Points

LIMEclipse is designed based on several software patterns like Factory, Visitor, Composite, etc. which makes it easily scalable, extensible and configurable. As a result of such patterns LIMEclipse provides configurability of the application at least in the following ways:
• New elements can be added to the diagram by defining corresponding model, figure and editpart
• New properties can be defined for an existing model element using PropertyDescriptors
• New graph layouting heuristics can be defined regardless of the model classes used in LIMEclipse models
• System settings can be defined externally without re-compiling the application using system.properties and messages.properties located in the root folder
• New XML hierarchies can be defined just by modifying XMLFactory class, hence not affecting the hierarchy in the existing models and figures
• New language bundles can be defined by creating corresponding messages.properties files

Having a fully-documented, scalable and configurable graphical editor, LIMEclipse can be configured to be used even in a completely different domain than LIME, easily. For such usages one can download this open-source project from [9].

3.5 Future Works

No software product exist which is 100% ideal. This asks from developers in a software community to keep their application alive by updating their programs and resolving the bugs which are found. LIMEclipse also is not an exception from this rule.

LIMEclipse in its current status, covers a wide range of features. In fact, to satisfy end-users by providing useable features, it might need to be improved and optimized. But in addition to improvements and optimizations on LIMEclipse, there are also a set of main concerns in LIMEclipse development that should be taken into account as future works:

• using better graph layouting - currently, layouting is done in several levels. This causes overlapping between the arcs and the associations in the diagram which should be avoided. One way to resolve this problem is to use better clustering and graph making heuristics. But another way is to use layouting frameworks like GraphViz [5] and fetching positioning information from such frameworks.
• using a more suitable layout manager for components - currently, XYLayout is used which sometimes crops the content pane based on the size of its contents.
• refactoring - refactoring should be done for all the packages in the source code. Sometimes legacy methods have remained that should be removed from the source code.
• creating update site - instead of downloading and putting the jar file inside plug-ins folder of Eclipse, plug-in should be put on an update site which would make its updating much easier.

Such improvements would make LIMEclipse a more trustable, user friendly, easy-to-use plugin. An up-to-date issue-list of changes and improvements in LIMEclipse can be seen on [9].
Summary

LIMEclipse is the graphical editor of LIME models. It is an Eclipse plug-in and it is based on RCP plug-ins, GEF and Draw2D. LIMEclipse uses several software patterns like Visitor, Composite, Command, etc. which helps it to be extensible and configurable.

This chapter introduced LIMEclipse and its various properties and development strategies. Using LIMEclipse, practical applications in LIME can be visually defined. This would ease creating new parallel applications by spending less time on integrating different parts of a program and by re-using predefined LIME models. The models introduced in the next chapter use LIMEclipse to visually depict a practical example of using LIME.
Chapter 4

Abstract Model for Associative and Commutative Reduction Operations

In computer terminology reduction operations are the operations which given an input array or stream, produce a smaller array or stream as the output. Reduction operations are frequently used in many parallel applications, e.g. in matrix multiplication. Therefore, any optimization on these operations results in significant speed-up of such applications.

However, any optimization on reduction operations sees two problems on its way. First, reduction operations are handled differently in many different parallel systems. Therefore, optimizations on these operations are confined to each individual parallel system and are not necessarily applicable to the others. This problem prevents improving reduction operations in a generic way. Second, their correctness is hard to verify, unless they run in lockstep. Nevertheless, since these operations are building blocks of many applications, if their correctness is questionable then a whole application’s correctness would be questionable.

A way to overcome both problems is to use LIME models (dataflow models) to model reduction operations. LIME models are convertible to many parallel systems. Therefore, imposing optimization in a reduction model is reflected in all those parallel systems. LIME models are in fact dataflow models which have a strong theoretical background. Dataflow models are side-effect free. Therefore, they produce the same result for the same input. The second concern is, hence, inherently addressed.

Implementing and testing reduction operation models brings three major outcomes for this thesis. First, it verifies the correctness of the LIME tool-chain in practical applications. Second, it resolves the problem of rewriting reduction operations for each different parallel back-end. Third, it results in the separation of an implementation (codes inside components) from the overall algorithm (reduction operation).

This chapter is outlined as follows. Section 4.1 describes the details of reduction operations. In Section 4.2, the ideas from MPI Reduction operation [10, 25], reduction in GPUs [18, 19, 31, 23], and Map/Reduce by Google [21] are described. These ideas serve as
inspirations for the final reduction models in this thesis. Section 4.3 using the existing knowledge about reduction, introduces static and dynamic LIME models for scalar operations. Finally, Section 4.4 explains how the Collect/Reduce model can be applied to non-scalar reduction operations as well.

4.1 Reduction Operations

Reduction operations are the operations that given an input stream or array produce a smaller output either by selecting from or by combining input elements (Fig. 4.1). In a sequential computer, such operations, like the other operations, are executed sequentially. It means that given \( n \) elements to reduce, it takes \( O(n) \) time for a sequential computer to calculate the result of a reduction operation. However, linear time complexity is in fact a bottleneck for a wide range of applications and must be avoided.

Figure 4.1: Reduction operation - converting an input to a smaller output.

A method of reducing the time complexity of reduction operations is to use a parallel system and to assign different portions of data to different processors. The most well-known method to do so, is to use tree-based reduction. A tree-based reduction consists of a set of stages (levels in the tree). In an \( m \)-ary tree, in each stage, \( m \) elements are assigned to a processor to compute a local reduction. In this manner in each stage input size is reduced to \( 1/m \) of its initial size. If the size of the input array is \( n \), after \( \log_m n \) stages, only a single output remains which is the final result of the operation. Therefore, in a tree-based reduction the time complexity is reduced from linear to logarithmic, \( O(\log_m n) \). For example for 16 elements in a binary reduction the reduction tree looks like Fig. 4.2.

Figure 4.2: A binary reduction tree for 16 elements. Each node contains a value. Each two edges in each stage sum up the values of two nodes.
Although tree-based reduction operations are widely used in parallel systems, in some cases they are treated with minor or major differences. In some cases additional restrictions are used for how a tree has to be structured [18, 32, 25] and in other cases the operations are investigated from a different angle e.g. Map/Reduce [21]. Such variations are described in the next section.

4.2 Reduction Operations Background

This section reviews reduction operations in three different domains. The first domain is MPI standard and its MPICH implementation which clearly shows the difficulties in the way of defining efficient reduction algorithms. The second is reduction in GPUs which clarifies how reduction operations are handled in data-intensive parallel systems like videocards with high data access rate. The last is Map/Reduce from Google which introduces a new way of looking at reduction operations.

4.2.1 MPI Reduction Implementation in MPICH

MPI is a standard used for programming parallel systems with clusters (or a cluster) of processors. The standard introduces a set of function prototypes and concepts which are then implemented by an implementation of the MPI standard. In the following we describe the ones which are needed for the rest of this thesis:

- **communication** - processors in MPI communicate with each other via message passing
- **communication group** - a set of processors in MPI should register themselves in a group before they can communicate with each other (default: MPI_COMM_WORLD)
- **memory model** - processors in MPI can use both distributed and shared memory models
- **reduction function prototype** - reduction operations in MPI are handled by MPI_Reduce method with a set of predefined reduction operations
- **reduction goal** - reducing $m$ arrays of ‘arrays of size $n$’ and producing a single array of size $n$ (Fig. 4.3). This is a generalization for reduction as it is defined in other parallel systems.

![Figure 4.3: The MPI reduction - reducing $m = 28$ (A-Z) arrays of size $n$ to an array of size $n$.](image)
MPICH is an implementation of the MPI standard that is widely used. In MPICH, reduction function prototype is implemented as a tree-based reduction algorithm. However, depending on the size of the input arrays \( n \), reduction is performed in different ways. The reason is that for large array sizes, it is not efficient to send all elements in the input arrays up in the tree.

To formulate different approaches in MPICH reduction method, first, a few terms should be defined:

- \( \alpha \) - initialization time for a message regardless of message size
- \( \beta \) - transfer time per byte
- \( \gamma \) - local reduction cost
- \( m \) - size of each array
- \( p \) - number of processors (arrays of size \( m \)) involved in the operation

Based on these terms, the cost model for small-size arrays can be defined as [32]:

\[
\lceil \log p \rceil (\alpha + m\beta + mn\gamma)
\]

(4.1)

This model can be described as follows. In each step, each two processors send their \( m \)-sized arrays to a semi-root processor (\( \alpha + m\beta \) term in the above time complexity). Then the semi-root processor computes local reductions by performing the reduction operation element-wise on the received \( m \)-size arrays (\( mn\gamma \) term). Finally, finding the final result using this procedure takes \( \lceil \log p \rceil \) steps to finish.

This cost model is good for small array sizes (being logarithmic), but it is mentioned in [32] that using another approach for larger size messages (array sizes larger than 2KB) is preferable. The approach for large-size arrays is to perform a \textit{Reduce-Scatter} operation followed by a \textit{Gather} operation in the root [32]. \textit{Reduce-Scatter} is similar to \textit{Scatter} operation, but in \textit{Reduce-Scatter}, reduction results of the first elements of all arrays is stored in processor 1, the second in the second processor and the \( p \)th in the \( p \)th processor. Then \textit{Gather} is called in the root which reads first element from the first processor, second from the second, until \( p \)th from the \( p \)th. This trick results in the following cost model [32]:

\[
2\log p\alpha + 2\frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma
\]

(4.2)

which for large arrays is smaller than the cost model mentioned in Equation 4.1. Thakur [32] reported that on Myrinet clusters this algorithm works twice as fast as the small-size array reduction implementation.

Based on this information, it can be easily observed that even in a single back-end, reduction operations should be configurable based on specific criteria. Therefore, the models provided later in this chapter should essentially consider configurability as well.

### 4.2.2 Reduction in GPUs

The computer chip of a videocard a.k.a. GPU is a parallel computing device that has gained much attention in the last few years. This is because of the GPU’s cost per GFLOPS. Like
in other parallel systems, many researchers have invested time on implementing reduction operations in GPUs efficiently. However, the specialty of a GPU is that it uses a fast shared-memory for communication between its embedded processors. The fast shared-memory in a GPU reduces communication costs to only the access time to memory locations. It means that communication overhead is minimized to almost zero. For this reason, it is expected that a GPU performs reduction operations for small arrays in a shorter time in comparison to a cluster of processors using MPI.

In order to confirm this assumption, an experiment is done by the author to justify this assumption. The experiment’s goal is to compare the time spent on sum reduction operation on an array of 33554432 integer elements between two different parallel systems. The first parallel system is a GPU - ‘NVIDIA GeForce 8800 GTX’, with two optimized (semi- and full-optimized) reduction codes provided in NVIDIA CUDA SDK. The second parallel system is a single-cluster of processors in DAS3 supercomputers in TUDelft connected via Ethernet with a similar optimized reduction code to NVIDIA CUDA SDK code (Refer to 4.6.1). The GPU codes are tested 50 times and their average times are computed. The single-cluster is tested with different number of processors involved; 1 to 64 nodes (Fig. 4.4). Using more than 64 nodes resulted in less decrease in time and is not depicted in Fig. 4.4. The best time achieved for the single-cluster example is for 63 and is compared with the time computed for GPU codes in Table 4.1. As can be seen in the table the time spent by the semi-optimized GPU code is the same as the single-cluster code while the full-optimized GPU code shows one-fifth of the time spent by the other two.

<table>
<thead>
<tr>
<th></th>
<th>CUDA semi-optimized code</th>
<th>CUDA full-optimized code</th>
<th>MPI code</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (ms)</td>
<td>10</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of the time spent for reduction operations by two CUDA codes on a GPU and an MPI code on a single-cluster of 63 processors

Based on the remarkable results of the above experiment and the relatively low price of a CUDA supported videocard, it is worthwhile to check GPU research on reduction operations in order to use their ideas and methods in the reduction models in this thesis. For this reason, reduction implementation in Brook [18], Accelerator [19], and CUDA [14] are
described in this chapter.

In Brook [18] reduction is defined for streams of data. A reduction operation receiving a single input stream produces an output stream which is either a smaller stream of the same type or a single element of the same type. This generalization in Brook is used for simplifying matrix multiplication [18].

Based on the ratio between the size of input and output streams (say \( n \)), Brook determines how many elements in the input should be combined to form a single element in the output. This means that a reduction operation takes \( O(\log n) \) time. Brook uses a local optimization by reading up to 8 elements in a single pass which can be called a *local reduction*. This optimization results in fewer steps e.g. for 8 elements it takes \( O(\log_8 n) \) time. Buck [18] showed that this method can compute sum of \( 2^{20} \) elements significantly faster in a GPU than a CPU. The same trick can be used in the model proposed in this chapter in Section 4.3.

In Accelerator [19] it is only mentioned that reduction is supported by using data-parallel arrays. Based on their report they support *Sum* (\( \sum \)), *Product* (\( \prod \)), *MaxVal*, *MinVal*, *All* (\( \land \), similar to And) and *Any* (\( \lor \), similar to Or). However, it is not mentioned how these operations are implemented.

CUDA provides reduction operation in its well-known SDK. This reduction program is tuned for CUDA architectures. The CUDA SDK optimizes reduction in 7 gradual steps (7 different kernels) which fairly depicts the applicable optimization tricks in the CUDA architecture. As mentioned above using CUDA, a videocard can reduce an array of size 33554432 integer elements using its kernel 7 (the most optimized one) in just a few seconds. In order to do so CUDA uses the following tricks:

- **interleaved addressing** - Combining non-adjacent nodes
- **loop unrolling** - reducing register usage
- **coalesced memory accesses** - accessing adjacent memory locations together
- **avoiding unnecessary synchronizations** - avoiding synchronization between threads of a warp
- **local reduction** - computing sum of more than two elements in each step

There has been no concrete research comparing different implementations of reduction operation in different GPU languages and frameworks. However, the knowledge gained from investigating individual implementations of reduction operations in GPUs dictates that configurability in a reduction model is inevitable and must be included in a flexible reduction model. Therefore, this thesis introduces such configurable models later in Section 4.3.

### 4.2.3 Map/Reduce Model

Map/Reduce model is a programming model proposed by Google [21] for distributed computing which has the goal to perform more generic reduction operations on extremely large
datasets. The concept is based on two functions \textit{Map} and \textit{Reduce}. \textit{Map} is a function like the Lisp \textit{Map} function. It applies an operation over all the elements in a list. \textit{Reduce} is the function which reduces \textit{Map} results to smaller amount of data. In fact, the Collect/Reduce model proposed later in this chapter inspires its separation of algorithm and implementation from the Map/Reduce model.

In Google, Map/Reduce is used as a template for applications like counting the occurrences of a word in documents which are spread on different servers. In a Map/Reduce application the \textit{Map} function produces a set of intermediate key/value pairs. The \textit{Reduce} function then combines key/value pairs which have the same key and produces the final results. In this model an engine uses these two functions to operate on data. This lets the programmer to be unaware of the underlying architecture while being able to define his own reduction algorithm.

Google has benchmarked applications using this model like \textit{Grep} and \textit{sort} and showed that this model performs efficiently while easing programming for the programmers as well. For this reason, many researchers have tried to map this concept to different parallel systems and applications [30, 20].

Ranger [30] evaluated Map/Reduce model on multi-core and multiprocessor systems. Introducing Phoenix which is an implementation of Map/Reduce model for shared-memory systems, Ranger simplifies programming of such systems by relieving the programmer from thread creation, task scheduling, data partitioning and fault tolerance [30]. He showed that word count, reverse index, matrix multiply, string match, and linear regression gain a significant speed-up using Map/Reduce model.

Kruijf [20] uses the same technique as Phoenix and maps Map/Reduce on Cell architecture. He states that the abstraction over the parallel architecture to the programmer for reduction operation eases programming significantly on Cell architecture. But in addition the model preserves the efficiency of a code as its completely hand-written equivalent code.

As a result, Map/Reduce can bring an abstraction for reduction operations on large datasets to a programmer. The abstraction on one hand eases programming and on the other hand makes it more computation oriented. Almost all the tedious, tricky tasks can become invisible to a programmer by using Map/Reduce abstraction. Therefore, the idea of separation of concept from the implementation is also used in the Collect/Reduce model proposed in the next section.

### 4.3 Reduction Models in LIME

Reduction in LIME can be modeled in four essential ways:

- sequential reduction
- implicit parallel reduction
- explicit parallel reduction
- explicit dynamic parallel reduction
The goal of all these models is to reduce an array of size $n$ to a single value. Each model, however, uses a different approach in order to improve the shortcomings of one another. This section describes these models in more detail. In all the models, providing the initial array to a model is not shown for the sake of simplicity. Concepts needed from LIME in these models are defined whenever it is appropriate.

### 4.3.1 The Sequential Model

In the sequential model a component receives an input of size $n$ and produces 1 element as the output (Fig. 4.5). In this case a component is used which performs a reduction operation on all the elements it receives sequentially. This model is only introduced to show all the possible cases that can be modeled in LIME. Therefore, this is the model with the worst time complexity, $O(n)$.

![Figure 4.5: Simple sequential model for reduction in LIME. The same component can be drawn for the implicit parallel reduction model](image)

### 4.3.2 The Implicit Parallel Model

The same component shape which is used in the sequential model (Fig. 4.5) can be used to denote an implicit parallel model. However, this component does not have any code inside. Nevertheless, annotating this component as a reduction component informs LIME to handle the input of this component by its own internal algorithm. This model is equal to what `MPI_Reduce` does. The implementation of this component is thus invisible to the user, hence the term implicit.

### 4.3.3 The Explicit Parallel Model

In an explicit reduction model, the algorithm of reduction is explicitly shown. Therefore, the programmer can define reduction in several ways. To further investigate how an explicit model can be defined, first, a few basic component-related definitions in LIME must be described:

- **activation** - a component is activated for different cycles in a schedule. Therefore, different activations can only run sequentially.
- **cloning** - a component is cloned by copying a component with all its internal ports and internal component, and internal associations. In LIMEclipse cloning is done by explicitly redrawing a component and copying its associated code and configurations. All clones of a component behave like separate components and are activated in parallel.

- **instantiation** - a clone can be instantiated when an arc between two components has mismatching consumption and production rates. For example if an arc receives elements from an output port of size 100 from one component and transfers those elements to another component to an input port of size 10 (Fig. 4.6), LIME instantiates 10 components of the second component to match 100 to 10. Like cloned components, different instances run in parallel. If there are two arcs between two components with mismatching consumption and production rates then their least common multiple is used to find the number of instances to be created. Instantiation is similar to cloning with two differences. First, instantiation is implicit. Second, instances are immutable.

![Figure 4.6: Instantiation occurs whenever there is a mismatch between consumption and production rate on an edge.](image)

Using **cloning** a simplistic one stage parallel reduction model with four parallel elements can be defined as depicted in Fig. 4.7. As mentioned before in Section 2.1, fifo:+ arcs in this model ask the ‘out’ port of the ‘Reduce_0 component’ to compute the reduction result of the elements on these arcs using a LIME native evaluation method. Although the model in Fig. 4.7 uses fifo:+ arcs, this model can be converted to other scalar reduction operations by changing ‘+’ to any other supported ‘scalar operation’ in LIME.

However, in order to make the model more generic, it is preferred to eliminate fifo:scalar_operation arcs completely. One way to do this is to use **instantiation** instead. Using instantiation, the resulting model can be cascaded to a multi-stage reduction model as depicted in Fig. 4.8. In this model each stage in the tree is depicted by a ‘reduction_phase’ component. The first stage receives 8 elements and using 4 instances of ‘reduction_phase1_0’ produces 4 elements. The second stage receives 4 elements and using 2 instances of ‘reduction_phase2_0’ produces 2 elements. Finally, the last two elements are reduced using a single instance of ‘reduction_phase3_0’. Apparently, one can extend these models to form the complete tree of reduction by drawing all the stages in a tree of reduction. Although, these models have the
benefit of explicitly depicting the parallel algorithm being used, they are not easily configurable and are not used as the final reduction model proposed in this thesis.

4.3.4 Dynamic Abstract Model - Collect/Reduce

In addition to the above mentioned models for reduction operations, yet another model can be defined which uses both tree reduction and the idea behind Map/Reduce. The idea behind
Map/Reduce is to confine the programmer to the algorithm specific code and to make the
rest of the workflow hidden from him. The model proposed in this part inspires its idea from
Map/Reduce and defines two similar concepts named as Reduce and Collect. The Reduce
component performs reduction based on the algorithm specified by a programmer and the
Collect component synchronizes processors in the tree by keeping the intermediate values
in different stages of a reduction tree. The critical role of the Collect component lets it to
perform more than synchronization. For example, it can perform another implementation
of a reduction operation or can cope with different implementations of a parallel back-end.
Configurability of the Reduce and the Collect component is explained later in this part.

However, before explaining the model, the following terms are explained:

- **VRDF models** - VRDF (See Section 1.3) is a variation of dataflow models in which
  a component receives the number of items to be consumed or to be produced as a
  separate data input. Obviously, this data input should be consumed by a component
  before any other data inputs. In many contexts this data input is called a control data.

- **BDF models** - BDF (See Section 1.3) is a variation of dataflow models in which a
  component sends/receives either 1 (or any other positive number) or zero output on
  its outgoing edge based on the data it receives. In many contexts this data input is
  called a control data.

- **α or reduction factor** - a constant number which is used in the Collect/Reduce. α
  is the number of elements which their reduction result is computed using a programmer’s implementation for the Reduce Component.

Like the other two previous models, a tree-based reduction scheme is also used in this
model (Fig. 4.9). It means that this model is also based on stages in a tree. In each stage a
local reduction component receives two inputs. The first input is the size of the input array
which is considered to be constant α in this model (it can be variable in modifications of this
model). The second input is the input array to be reduced. The Local Reduction component
computes the reduction result of of α input elements and send the result out as its output.
The output of the Local Reduction component is fed to the Collect component. Since many
instances of the Local Reduction are activated in each stage, the Collect component receives
more than a single data element on its incoming edge (an explicit merge).

The Collect component, on the other hand, is based on VRDF. It determines the number
of elements it should receive in each stage from the Size Computer component. The Size Computer component simply computes intermediate outputs size in the tree for each stage
by dividing size of the previous stage by α. The Collect component waits until it receives
the number of elements that is dictated by the Size Computer component.

Since the Collect component is based on BDF, when it receives only one element it sends
that single element on its Print output port. Otherwise, the Collect component sends the
incoming data to the input port of the Local Reduction component and forms the next stage
of the tree. The mismatching rate of production and consumption rates on the arc from the
Collect component to the Local Reduction component comprises the parallelism point of
the model.

Obviously, optimizations can be easily added to this model. For example if sequential
reduction of p elements is more efficient than feeding it back to the Local Reduction com-
ponent, it can be easily programmed in the Collect component. Or if the reduction operation should be performed differently in each stage, this can be dictated by adding an additional arc to the Local Reduction component and selecting among different implementations based on the incoming data on the arc.

In contrast to the other three reduction models, this model is easily configurable. Therefore, it is introduced as the final reduction model in this thesis. This abstract and configurable model makes tuning of tree-based reduction operations on a special back-end a lot easier. It shows parallelism explicitly on one hand and on the other hand hides back-end specific issues from a programmer using LIME programming model.

Since the Collect/Reduce model is a generic reduction model, it can be used for different commutative scalar reduction operations e.g. produce, maximum, minimum as well. Even non-scalar operations like database reduction queries can be addressed by this model which is explained in the next section.

Like the Map/Reduce model the Collect/Reduce model is a template for reduction. Therefore, the model is independent from a specific parallel back-end or implementation. However, the Collect/Reduce model and the Map/Reduce model have two major differences. First, the Collect/Reduce model uses only ‘the Reduce’ from the Map/Reduce model and is not assumed to do anything about the ‘Map’. Second, the Collect/Reduce model is based
on dataflow models which is more common in Embedded Systems. Therefore, mapping the Collect/Reduce model to the systems outside the Embedded Systems domain is a tricky task. However, in order to show the flexibility of the Collect/Reduce model, an example of a reduction operation in Database Systems is explained in the next section.

4.4 The Collect/Reduce Model in Database Systems

Although, LIME models are not supposed to support database models, dataflow models can be used to model database queries as well. In order to clarify this statement, this section describe the Collect/Reduce model in a database reduction query. The query to be described in Collect/Reduce scheme is:

```
SELECT min(price) AS minprice FROM fruits GROUP BY type
```

This query finds the minimum price of each group of fruits e.g. apples, oranges, etc. in a table of fruits of different types. There are two issues that must be addressed in this query. First, fruits must be grouped by their types. This is the initialization part of the query. Second, the minimum price of each group should be computed to finish the query. The first issue is handled by using parallel initializers. Each initializer receives ‘price’ column of all the fruits of a single type. For the second issue, each group of fruit is fed to an instances of `Collect` component along with the number of elements in the group. Then each pair of `Collect` and `Local Reduce` computes the minimum of its group in a tree-based reduction scheme (No more database related queries). Finally, the results of all reduction trees are fed to the instances of the `Printer` components which completes the execution.

This application shows how the model can be used for different domains else than Embedded Systems. However, for deriving a conclusion of the applicability of the Collect/Reduce model for more generic reduction models, experiments should be carried out on demand.

Summary

Reduction operations are the operations which receiving an input, reduce its size by either selecting or combining its elements. They can be designed and implemented in many ways in several parallel systems. In order to simplify their various designs this chapter introduced generic LIME models based on dataflow models. Dataflow models have the benefit that they can be mapped to different parallel system using the LIME tool-chain. This fact frees programmers from taking care of system-specific tricks to implement their reduction algorithms.
Conclusions and Future Works

Conclusions

Nowadays, parallel systems are favorable choices for computation intensive applications. Nevertheless, their rapid changes in hardware and software prevent people from investing too much in them. As mentioned in this document a way to hide these changes from end-users is to use software models like dataflow models instead of concrete software programs. By using dataflow models one can harness a parallel system’s power by only focusing on his algorithmic or computation issues.

The ‘LIME tool-chain’ which is introduced in this document uses dataflow models in order to model parallel applications. It uses the LIMEclipse Eclipse plug-in front-end in order to draw dataflow models in its various formats; SDF, CSDF, VRDF, BDF, etc., and converts those models to several parallel systems back-ends such as Pthreads and CUDA using LIME middle-end. In simple words, the LIME tool-chain prevents “reinventing the wheel” by preventing programmers from rewriting a single applications for different platforms.

In order to prove the usefulness of the LIME tool-chain, abstract models for common operations in parallel systems such as reduction operations were introduced in this document. Such models are easily configurable and explicitly parallel which makes them reusable. Reusable models would create a rich model library for the LIME tool-chain which adds to the acceptance of LIME as a standard parallel model by parallel programmers. Using a parallel programming tool-chain like LIME eases programming parallel applications and expedites advances in computational science.

Future Works

Future works for this research can be separated in three different categories:

1. Improving and adding new features to the LIMEclipse plug-in
2. Improving and adding new features and new parallel back-ends to LIME
3. Defining more LIME models for common parallel operations
LIMEclipse can be improved in its several aspects. First, it should adapt new features in LIME. Second, it should provide better layouting heuristics to layout the diagram in a more optimal and eye-appealing way. Third, it should be made generic to be able to accept different domain models without affecting the code. It is possible using reflection by feeding XML files as domain models. It would make LIMEclipse as a DSL for graphical editors. Finally, it should be tested by several practical applications in LIME to prove its usefulness.

The LIME programming model can be improved by handling common parallel operations, natively. This could make LIME as a programming model with a rich library. Such improvements would add to acceptability of LIME by programmers in parallel systems. Second, it can be converted to a more flexible programming model by providing hybrid back-ends. A hybrid back-end can then convert different components in LIME diagrams to different back-end. Such improvements can make hybrid reduction operations possible. For example, in a hybrid reduction operation local reduction could be performed in CUDA and the global reduction would be performed in MPI. Third, it can define its domain models by XML files which can be further added to the future’s domain specific LIMEclipse.

Several new LIME models can also be defined for common parallel operations in the same way as it is done in this document. Such models can introduce LIME to programmers’ community with concrete examples which with a tuned LIME would produce efficient code equal to manually written codes.
4.5 LIMEclipse

4.5.1 LIMEclipse Command Models

In addition to the model elements in LIMEclipse, a set of rules can be defined for these elements. These rules are domain rules which should be obeyed by them in the diagram e.g. a port should be added only to a component. This sort of rules are model-related and are defined by extending Command classes. An example for such rules is AddCommand which checks if an element can be added to a container (LIMEDiagram or Component). Each command class has a canExecute method which is used by GEF framework to check if the command should be prevented. canExecute of AddCommand can be defined as:

```java
@Override
public boolean canExecute() {
    if (!child instanceof InputPort || child instanceof OutputPort ||
        child instanceof LimeCodeFragment) &&
        !parent instanceof LimeComponent) {
        return false;
    } return true;
}
```

Figure 4.10: An example for Command classes. AddCommand prevents a port to be added to something else than LIMEComponent.

4.5.2 LIMEclipse Model Properties

Model classes also provide properties which can be shown in properties view of Eclipse environment. This makes their settings configurable as shown in Fig. 4.11. In order to inform GEF to show these properties in the visual editor one should define Property Descriptors for member variables of a model class. As an example, sourcecode for LIMEComponent’s return type and type properties are shown in 4.12.
4.5.3 LIMEclipse Model Properties

XMLFactory is the connection point of LIMEclipse to LIME. This Factory class performs all XML related operations and converts LIMEclipse models to XML files and vice versa. Class description of XMLFactory is depicted below:
4.6 MPI

4.6.1 MPI Code to Test Reduction Operation

MPI code for implementing reduction similar to CUDA SDK reduction code:

```c
start_timer();
//_______________________________________________________________
// Starting point for reduction
//_______________________________________________________________
int i;
for (i = 0; i < local_data_array_size; i++)
{
    local_sum += local_data[i];
}
//_______________________________________________________________
//_______________________________________________________________
// do final global reduction
//_______________________________________________________________
MPI_Reduce(&local_sum, result, 1, MPI_INT, MPI_SUM, root_rank, MPI_COMM_WORLD);
//_______________________________________________________________
//_______________________________________________________________
stop_timer();
```