Software Synthesis for Dynamic Data Flow Graph

by

Chabong Choi
Codesign and Parallel Processing Lab.
Computer Engineering Dept.
Seoul National University

ABSTRACT

Data flow graph is a useful computational model to describe the functionality of a digital system. To execute a data flow graph on a target system, it should be synthesized to the code to be compiled on the target system. Current research activities on software synthesis are mainly focused on Synchronous Data Flow (SDF) graph, a special case of data flow graph [Lee87a]. Even though the static property of the SDF graph provides efficient schedulability and inexpensive implementation, it limits the range of applications because it can not represent the control structure of the application.

On the other hand, Dynamic Data Flow (DDF) graph can express the control structure, such as conditionals and data dependent iterations. DDF graph is shown to be equivalent to Kahn Process Networks [Kah74]. This implies that the termination problem of DDF graph is dependent on the graph itself, but the memory bounding problem is dependent on the execution order, as well as the graph itself.

This paper synthesizes a C code for a DDF graph which includes not only the functions associated with the nodes of the graph, but also the run-time scheduler and the run-time buffer manager which can execute a DDF graph without a deadlock in a bounded capacity. In addition, this paper suggests a hierarchical implementation of DDF which enhances the efficiency of run-time scheduling by clustering SDF type nodes as a specific macro node.
Key Words: Data flow, Code synthesis, Scheduler, Simulation
1. INTRODUCTION .................................................................3
2. BACKGROUND .............................................................6
  2.1. PREVIOUS WORK: PTOLEMY ...........................................6
    2.1.1. Synchronous Data Flow ...........................................8
    2.1.2. Boolean-controlled Data Flow ...................................11
  2.2. MOTIVATION .............................................................13
3. DESIGNING DDF SCHEDULER ..........................................16
  3.1. PROPERTY OF DDF NODES ..........................................16
  3.2. DDF SCHEDULERS ......................................................18
    3.2.1. Minimax Scheduler ...............................................20
    3.2.2. Blocking Scheduler ...............................................21
  3.3. EVALUATION OF SCHEDULERS ......................................22
    3.3.1. Experiment ..........................................................24
4. SYNTHESIZING DDF GRAPH .............................................28
  4.1. REPRESENTING DATAFLOW NODES ..................................28
  4.2. REPRESENTING ARCS ..................................................30
  4.3. MANAGING BUFFERS ..................................................31
    4.3.1. Static Buffers ....................................................31
    4.3.2. Dynamic Buffers ................................................32
    4.3.3. Buffer Expansion ................................................33
  4.4. SYNTHESIZING STAR CODES .........................................35
    4.4.1. SDF Star Code Synthesis .......................................35
    4.4.2. DDF Star Macros ................................................36
  4.5. RUN-TIME-SCHEDULER ...............................................37
5. CLUSTERING SDF NODES ................................................39
6. EXPERIMENT .............................................................41
7. CONCLUSION .............................................................43
REFERENCES .................................................................44
APPENDIX .................................................................49
1. INTRODUCTION

The rapid technology improvement in microprocessors and integrated circuits recently makes various digital systems available in wide range of applications. As much interest has been concentrated on the multimedia application area, the core technologies for the audio and video signal processing have been developed quickly. Many small scale digital systems such as consumer products based on the simple processor core technology were able to be manually designed and verified by a few experienced experts. However, the increased complexity and the short life cycle of today’s systems increase the need of more aggressive methodology capable of designing systems quicker and more easily. Therefore, diverse approaches for the design automation from system specification to system synthesis have been actively tried [Buc94][Wol94].

Design procedure of a digital system consists of several stages such as describing the system specification and the algorithms, simulating, verifying the system, and implementing target hardware and software. Various CAD tools have been developed to automate or aid each stage[Ho88][Ste90]. Recently, the approaches that integrate various CAD tools into one framework are in progress, of which Ptolemy is a good example.

Ptolemy is an environment for simulation, prototyping and software synthesis of heterogeneous systems. It has been developed by Ptolemy group of U.C. Berkeley since 1990 [Buc94]. An algorithm verified in Ptolemy can be synthesized into firmware for specific microprocessors or into hardware description languages such as Silage [Gen90]. In Ptolemy, a DSP algorithm is represented in Synchronous Data Flow (SDF), a special case of data flow graph [Lee87a].

Data flow graph is a useful computational model to describe the functionality of a digital system [Ack82]. The nodes of the graph represent computations, and directed edges between nodes represent the data flow between computations. To execute a data flow graph on a target system, it should be translated to code to be
compiled on the target system. This problem is called “software synthesis.” Current research activities on software synthesis are mainly focused on SDF. In SDF, a node consumes or produces a fixed number of data samples, or tokens, for each execution. Therefore, the flow control is completely predictable at compile time. This static property of SDF provides efficient schedulability and inexpensive implementation, but limits the range of applications because it cannot represent the control structure of the application.

On the other hand, Dynamic Data Flow (DDF) graph can express the control structure of an application. A DDF node may consume or produce a varying number of tokens for each execution. Therefore, the control structures such as conditionals, data-dependent iterations can be supported. DDF is shown to be equivalent to Kahn Process Networks, where a number of concurrent processes communicate through unidirectional FIFOs with unbounded capacity [Kah74]. Writes to a FIFO are non-blocking and reads are blocking. In Kahn process networks, each process consists of repeated executions of a node in a certain execution rule (or firing rule). However, Kahn process networks do not imply how to schedule the processes. Thus, many execution rules may exist in DDF even though efficiency has not yet been investigated.

In software synthesis for DDF, the execution order of nodes is determined at run-time by a scheduler embedded into the synthesized code. Furthermore, a run-time buffer manager is required for efficient handling the buffers.

This paper synthesizes a C code for a DDF graph. The C code includes not only the functions associated with the nodes of the graph, but also the run-time scheduler and the run-time buffer manager. The scheduler and the buffer manager should be correct and efficient. A scheduler is correct if the following two conditions are satisfied; 1) if a DDF graph is deadlock-free, then the execution of the graph by the scheduler should not be deadlocked, 2) if a DDF graph can be executed with bounded memory, then the execution by the scheduler should be finished also with bounded memory. In addition, this paper suggests a hierarchical implementation of DDF that enhances the efficiency of run-time scheduling by grouping, or clustering,
SDF-type nodes as a specific macro node. In short, this paper synthesizes an application specific run-time kernel for dynamic data flow graph.

Chapter 2 describes the background knowledge related to data flow models and motivation for DDF synthesis, chapter 3 reviews two schedulers for DDF and selects one of the two in order to be embedded into the generated code, chapter 4 describes the structure of synthesized code and key issues related to dynamic scheduler and dynamic buffer management, chapter 5 suggests an optimization for the generated code by clustering SDF nodes into each macro node, and chapter 6 shows experiment and evaluation, and finally we conclude with some remaining works.
2. BACKGROUND

Software synthesis is a key step of the whole process of behavioral synthesis of digital systems as shown in figure 2.1. To design a digital system, system specification on functionality, performance and restriction is given. In the proposed approach, the functionality of the system is represented with data flow models. By simulating the data flow graphs, algorithm correctness and performance are monitored to refine the algorithm. The next step is to partition the data flow graphs into subsystems and to synthesize codes for those subsystems. For a hardware subsystem, we generate a VHDL code that will be an input program of hardware behavioral compiler. On the other hand, we will synthesize a C-code for a software subsystem, which is the main topic of this paper. Since our framework is based on Ptolemy, this section will shortly review the Ptolemy framework.

2.1. PREVIOUS WORK: PTOLEMY

Ptolemy is an environment for simulation, prototyping and software synthesis of heterogeneous systems[Buc94], which has been developed by Ptolemy group of U.C. Berkeley. Ptolemy is designed to support many different computational models and to permit them to be interfaced cleanly. It uses modern object-oriented software technology to model each subsystem in a natural and efficient manner, and to integrate these subsystems into a whole.
Ptolemy uses a block diagram as an input program graph. As shown in figure 2.2, a block, *Block* object in Ptolemy terminology, describes a computational task that receives input data and generates output result through the standard interface called *Porthole* objects. An arc between blocks consists of two objects: *Geodesic* and *Plasmas*. Geodesic is a FIFO queue with unlimited size buffering data *Particles* transferred between blocks, and Plasmas are garbage collector.

Ptolemy provides an efficient way to manage the complexity of a large system with a hierarchy in the description. An atomic block is called a *Star* and a hierarchical block which contains stars or hierarchical blocks is called a *Galaxy*. The outermost block which contains the entire application is known as a *Universe*. The entity that determines the order of the execution of a universe, is a *Scheduler*. A *Target* also controls the execution of an application by determining suitable behavior for the application. The combination of a set of blocks, targets, and associated schedulers that conform to a common computational model is called a *Domain*.

Ptolemy is written in C++, an object-oriented programming language. Through the object-oriented abstraction mechanism of polymorphism, new domains including new computational models, new types of blocks, and new communication primitives among blocks, can readily be added to Ptolemy without any modifications to the
Ptolemy kernel or the previously implemented domains. Moreover, Ptolemy provides a seamless interface between heterogeneous domains.

There are many domains in Ptolemy, which may be classified into Simulation Domains and Code Generation Domains. In simulation domains, Ptolemy itself plays a role of simulation engine to execute the block diagram and produce the results. On the other hand, in code generation domains, Ptolemy produces codes, for example C-code or VHDL code, from the input graph, thus acts as a texture program generator from graphical description. When a star of code generation domain is executed (or fired), the `go()` method of the star adds ASCII text to a data structure defined in the Target, framing a source code. Scheduler controls the sequence in which each star fires, and thus, it determines where and how many times to place the code block generated by each star. A target in code generation domains is specific to the hardware that will run the generated code, while a code generation domain is specific to the type of language. For example, CGC (Code Generation in C) domain generates C code from the input graph while two targets may be defined in the CGC domains. One is for single processor and the other is for workstation clusters.

Ptolemy supports many computational models in simulation domains, but only SDF and BDF (Boolean-controlled Data Flow) models in code generation domains. It is because its clean algebraic structure of those models enables the generated code to have simple structure. Therefore, code generation capability and simulation capability are unbalanced in the current Ptolemy framework. Now, we briefly review the algebraic characteristics of SDF and BDF models.

2.1.1. Synchronous Data Flow

Data flow [Ack82] model has been proven to be an effective way representing functionality of a digital embedded system because it expresses algorithm structure intuitively and explicit parallelism. Under the data flow model, algorithms are described as directed graphs, as illustrated in figure 2.2(a), where the nodes represent computations and the arcs represent data paths. If data samples or tokens are available on the input arcs of a node, the node performs its computation or fires by
consuming input samples, and then produces samples on its output arcs. A node with no input arcs may fire at any time. Because only availability of data determines to perform the computation of a node, data flow is called to be data-driven [Tre81].

Synchronous data flow (SDF) is a special case of data flow, in which the number of data samples produced or consumed by each node on each invocation is specified priori [Lee87a], which means the number of tokens produced or consumed must be independent of the data. An SDF graph may be depicted as shown in figure 2.3. Nodes are numbered and arcs are also numbered (underscored). The number of tokens produced and consumed by firing each node is also expressed on each arc.

An SDF graph can be characterized by a topology matrix where a column is assigned to each node and a row to each arc. The \((i,j)\)th entry is the number of tokens produced by node \(j\) on arc \(i\) for each firing, and the number is negative if node \(j\) consumes tokens on arc \(i\), and the number is zero if there is no connection on arc \(i\) by node \(j\). The topology matrices of figure 2.3 (a) (b) (c) are, respectively:

\[
\Gamma_a = \begin{bmatrix} 1 & -2 & 0 \\ 0 & 1 & -1 \\ 1 & 0 & -2 \end{bmatrix} \quad \Gamma_b = \begin{bmatrix} 1 & -2 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -2 \end{bmatrix} \quad \Gamma_c = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}
\] (2.1)

\((a)\) A correctly constructed SDF \hspace{1cm} \((c)\) A deadlocked SDF
\((b)\) An inconsistent SDF \hspace{1cm} \((d)\) Avoiding deadlock

Figure 2.3.
Node invocations are represented as the following vectors;

\[ v(n) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T \text{ or } \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T \text{ or } \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T \]  \hspace{1cm} (2-2)

where a vector \( v(n) \) is defined that the \( i \)th entry is one if node \( i \) is invoked at time \( n \), and zero for each node not invoked. If we define a vector \( b(n) \) to contain the number of tokens in each arc at time \( n \), the change in the number of tokens caused by each invocation is given by;

\[ b(n + 1) = b(n) + \Gamma v(n) \]  \hspace{1cm} (2-3)

If we can find a non-zero vector \( q \) satisfying \( \Gamma q = o \), known as balance equation \cite{Lee87a}, where \( o \) is a vector full of zeros, then \( b(n+p) \) will be same as \( b(n) \) at every time period \( p \), that is, the total number of tokens on the graph is bounded. In figure 2.3(a), there is a solution;

\[ q = \begin{bmatrix} 2 & 1 & 1 \end{bmatrix}^T = 2 \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T + \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T + \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T \]  \hspace{1cm} (2-4)

Furthermore, \( q \) can be the following;

\[ q = \rho \begin{bmatrix} 2 & 1 & 1 \end{bmatrix}^T \]  \hspace{1cm} (2-5)

for any positive integer \( \rho \), which means \( q \) specifies the number of times to invoke each node in one period of schedule. Therefore, schedule to invoke node 1 twice and both node 2 and 3 once, does not generate any change of the whole tokens kept on all arcs at each iteration. We call \( q \) the repetition vector, we may also say that a SDF graph with non-zero repetition vector is consistent, and it has a cyclic schedule which returns the graph to its original state after every repetition.

Another example of SDF graph is illustrated in figure 2.3(b). We can know there is no non-zero solution for balance equation of the graph, i.e. \( \Gamma b q = 0 \), therefore,
the graph in figure 2.3(b) is inconsistent. Moreover, it cannot be performed in a cyclic schedule, which means the number of total tokens it requires to fire are not bounded.

Finally, consider an example of figure 2.3(c). Even though it has a non-zero solution for the balance equation, but it cannot be scheduled because of the lack of initial tokens; it is deadlocked. In order to avoid this problem, we may put an initial token, i.e. a delay, on arc 2, that is, \( b(0) = [0 \ 1]^T \) as shown in figure 2.3(d). In this case, \( b(n) \) in (2-3) will be \( [0 \ 1]^T \) after every repetition, instead of a zero vector.

In short, we can recognize that if there is a non-zero solution for the balance equation and there are enough initial tokens in any iteration of an SDF graph to avoid deadlock, the graph can be scheduled with bounded capacity for data tokens. The schedules for consistent SDF graphs are static so that they are generated at compile time. An SDF scheduler can take advantage of this static information to construct a schedule that it can be used repeatedly. This static property provides various benefits; ease of programming since the availability of data tokens need not be checked, a greater degree of compile-time syntax checking since inconsistencies are easily detected, run-time efficiency, and automatic parallel scheduling [Buc94]. SDF models of computation, however, has a serious limitation that it can not represent control structure such as conditionals and data dependent iterations.

2.1.2. Boolean-controlled Data Flow

Boolean-controlled dataflow (BDF) is an extension model of SDF which provides a limited set of data-dependent operations analogous to the if-then-else or case statement in C [Buc93][Buc94]. A BDF graph consists of SDF blocks and a couple of BDF nodes illustrated in figure 2.4.
A SWITCH node consumes one token from its input arc and reroutes it to its output arc according to a Boolean value from its another input arc, or control arc, illustrated in figure 2.4(a). A SELECT node is another BDF node used as a pair of SWITCH node, which reads a token from one of its input arcs according to a Boolean value from its control arc, and then produce it to the output arc, illustrated in figure 2.4(b). By using these two nodes, a graph with not only consistency but also data dependent behavior can be constructed, having IF-THEN-ELSE structure as shown in figure 2.5.

In figure 2.5, node 1 produces a token and this token is delivered to node 4 if node 7 produces TRUE, or to node 3 if node 7 produces FALSE. The Boolean value from node 7 is also transported to node 5. This graph performs node 3 or node 4 conditionally according to the value from node 7. The topology matrix for figure 2.5 can be constructed with a function of $p$, the vector of $p_i$ which is the long-term proportion of TRUE tokens in the Boolean stream $b_i$ which supplies the control inputs as shown in figure 2.4.
\[
\Gamma(p) = \begin{bmatrix}
1 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & (1 - p_1) & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & (p_2 - 1) & 0 & 0 \\
0 & p_1 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -p_2 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & -1 & 0 & 1
\end{bmatrix}
\] (2-6)

The balance equation for this case is given by the following:

\[
\Gamma(p) \cdot r(p) = o
\] (2-7)

If the balance equation has non-zero solutions regardless of values of \( p_i \), the graph is said to be strongly consistent. On the other hand, if solutions exist only for some values of \( p_i \), the graph is called weakly consistent, and such graphs are usually not correct. Because \( b_1 \) and \( b_2 \) in figure 2.5 are produced by the same node, \( p_1 \) and \( p_2 \) are also same, therefore, we may let \( p_b = p_1 = p_2 \). We can obtain the following solution for the above balance equation, using the techniques of [Lee91]:

\[
r(p) = k \begin{bmatrix} 1, 1, 1 - p_b, p_b, 1, 1, 1 \end{bmatrix}^T
\] (2-8)

If we regard \( p_b \) as the number of TRUE tokens, there are two firing sequences associated with two values of \( p_b, 0 \) and 1. Since both firing sequences require bounded memory, the example BDF graph has a bounded schedule. Also, based on value \( p_b \), we select one of these two firing sequences that are determined at compile-time.

In short, a BDF graph does not lose the merits for static scheduling like a SDF graph even though it provides data-dependent operations. Therefore, it may be scheduled at compile time with a bounded cyclic schedule, as well as executed in bounded memory.

2.2. MOTIVATION
In the previous sections, we review SDF and BDF which are implemented in Ptolemy and widely used in many designing areas. These computation models are useful and efficient, but have some restrictions.

A consistent SDF graph can be fully scheduled statically, or at compile-time. It is adequate for digital signal processing systems, in which typical DSP algorithms require relatively little run-time overhead. However, recent applications with multimedia functionality require complex decision-making at run-time, as a result, for the data-dependent operations the SDF is extended into the BDF with a few nodes having dynamic property, or control structures.

The BDF model works well in various applications, and it has been shown that the addition of only the SELECT node and the SWITCH node of figure 2.4 to the SDF model is sufficient to make it Turing complete [Buc93]. This means that it is not always possible to construct a bounded cyclic schedule for a BDF graph. In fact, it is not decidable whether or not an arbitrary BDF graph can be executed in bounded memory [Lee95]. Even though BDF graphs have the same expressive power as Turing machines, it is not convenient to represent “for” type data dependent iterations. Therefore, it is necessary to extend the BDF model further to define dynamic data flow (DDF) model of computation.

Dynamic data flow graph contains other dynamic nodes besides SWITCH and SELECT that may consume and produce a varying number of tokens for each execution. For example, REPEATER star in figure 2.6 produces the copies of the input token as many number of times as specified the integer control value. As a result, the number of execution of node F becomes data dependent.

![Figure 2.6. DDF REPEATER](image)
Recently, it is revealed that DDF model of computation is nothing but an execution model of Kahn Process Networks [Kah74] due to the breakdown of a process into a sequence of node firing. In Kahn process networks, concurrent processes communicate each other through infinite unidirectional FIFO channels. A firing of a DDF node corresponds to a different quantum of execution after a process is scheduled in. Like a Kahn process that suspends on blocking reads, a DDF node is not runnable until a specified arc has enough number of tokens. We summarize Kahn’s conditions in figure 2.7.

1. *Data tokens should be transported only through communication channels. Any shared variable is not allowed.*

2. *A node should be blocked when attempting to read empty input channel. It may not access input channel to check the data availability.*

3. *A node can not wait for data on more than one input channel at a time.*

*Figure 2.7. Kahn’s Conditions*

It is known that if execution rules of a program satisfy Kahn’s conditions, its termination is decided by the program itself, not by the execution order, that is, a bad scheduling decision could not turn a non-terminating program into a terminating one. The boundedness property, however, *does* depend on the execution order as well as the program’s definition. A process network is said to be *strictly bounded* if the number of data elements buffered on the communication channels remains bounded for all possible execution orders [Par95].

If the network is not strictly bounded, it is important to determine an execution order (or schedule) in which a data flow graph is performed with bounded memory. Unfortunately, the problem finding this schedule is equivalent to a well-known halting problem of Turing machines, it is not possible to determine it at compile-time. Instead,
we have to find out a run-time scheduler of the DDF graph which requires bounded memory.

Therefore, software synthesis of DDF model of computation is to generate the run-time scheduler code as well as computation code of nodes. To make this run-time scheduler be application-specific, we aim to avoid unnecessary overhead of general purpose run-time systems or operating systems.
3. DESIGNING DDF SCHEDULER

This chapter describes the property of DDF nodes in view of scheduler, and suggests two DDF schedulers satisfying non-termination and bounded memory issues in the previous chapter. We finally analyze the efficiency of both schedulers discussed to choose one as a run-time scheduler embedded in the generated codes.

3.1 PROPERTY OF DDF NODES

A DDF graph is a generalized computational model extended from the SDF graph [Lee87a] and the BDF graph [Buc93]. In this section, we will assume that all nodes are DDF, which implies that SDF nodes are special DDF nodes.

A consistent multi-rate graph in figure 3.1 can be executed forever without termination, but according to the firing order, it can be bounded or not.

![Figure 3.1. A multi-rate graph](image)

For example, firing \{ A, 100(B), 100(C) \} requires bounded memory, i.e., 100 tokens on each arc. But because node A can always fire, tokens accumulated on arc 1 may be unbounded, if only node A fires forever. Therefore, we may say the graph in figure 3.1 is bounded, but not strictly bounded as we mention in 2.2. How can a scheduler control the firing order of nodes not to exceed the capacity limit?

We define a node as enabled when enough number of tokens are available at its inputs to satisfy its firing conditions. Firing of an enabled node may be deferred when there are already sufficient tokens to satisfy the demand of its downstream node. Such node is called deferrable [Par95]. In figure 3.1, node A becomes deferrable after it fires once because its downstream node, B, satisfies the demand as long as tokens exist on the input arc. If we allow firing a deferrable node without restriction, the number of tokens on its output arc keeps increasing. If a DDF graph is bounded, we can assume there exists a finite bound \( b \) in which no node consumes or produces tokens.
more than $b$ tokens in a firing. Because the maximum tokens on an arc that could not satisfy the demand of its downstream node is $b-1$, the maximum $2b-1$ tokens could be accumulated on any arc satisfying the demand of its downstream node. Therefore, the number of tokens accumulated on each arc would never exceed $2b-1$ if deferrable nodes are never allowed to be fired. In figure 3.1, after node A fires and then node B fires, both node A and B become deferrable and node C becomes enabled, thus the graph may be executed in the order \{ A, 100(B,C) \}.

In addition to the above states of nodes, we define another state, `blocked`, in which a node can not be fired since it reaches the limit of arc capacity. We will discuss blocked state later in detail. In short, a node may have one of the following states:

- **Enabled**: Enough tokens are available on its input arcs to satisfy its firing conditions.
- **Deferrable**: It is enabled and at least one of its output arcs has sufficient tokens to satisfy the demand of its downstream node.
- **Blocked**: It is enabled and at least one of its output arcs has tokens equal to or more than an arc capacity.

*Figure 3.2. States of DDF node*

An EndCase node is shown in figure 3.4 which is a representative DDF node. It has the following firing conditions:

1. Wait for a token on control arc (arc c).
2. Read control arc c, and wait for a token on input arc i0 if value of c is 0, or on input arc i1 if value of c is 1.
3. Read a token on the specified input arc, and write it onto output arc o.

*Figure 3.3. Firing rules for EndCase*
The graph in figure 3.4 shows another case of deferrable state. While EndCase waits for control input, node A is deferrable even though no token exists on its output arc. If the control input is always 1, EndCase will not attempt to read a token from arc $i_0$. Firing A may produce redundant output tokens. To keep memory requirements low, the scheduler defer the execution of deferrable nodes as much as possible. However, it usually occurs when deferrable nodes should be executed. There is an example in figure 3.5.

Node A and B are both deferrable. Since the graph is non-terminating, we have to execute deferrable nodes. Therefore, the goal of scheduler is to determine the appropriate firing order based on the states of DDF nodes.

3.2 DDF SCHEDULERS

This section describes two scheduling policies satisfying the ultimate requirements of non-termination and boundedness [Par95]. If a DDF graph satisfies the Kahn’s condition in figure 2.7, a DDF scheduler should satisfy the followings:

- **Requirement 1 (Deadlock-Free):** The scheduler should implement a complete execution of the DDF graph, that is, if the graph is non-terminating, then it should be executed forever without terminating.

- **Requirement 2 (Bounded Memory):** The scheduler should execute the
DDF graph within a bounded number of tokens on each arc, if the graph is bounded. For an unbounded graph, the scheduler may implement only a partial execution to avoid exceeding the capacity limit.

Figure 3.6. Requirements for DDF scheduler

DDF schedulers can be classified as data driven or demand driven. A data driven scheduler fires a node as soon as sufficient data is available (eager evaluation) while a demand driven scheduler avoids the unnecessary production of tokens by deferring the firing of a node until its downstream node demands tokens (lazy evaluation). Every data driven scheduler satisfies requirement 1, but for a bounded graph, only some of the execution orders satisfy requirement 2 [Par95]. Thus to find a scheduler satisfying both requirements with efficiency is not easy. Before we explain two schedulers, we define some terms.

- **Default Iteration**: A period defined by a simple loop that is implemented by the scheduler to execute all ready-to-run nodes.
- **Effective Iteration**: A period in which the scheduler repeats default iterations until nodes with firingsPerIterations are fired as many times as specified firingsPerIterations.
- **firingsPerIteration**: A value by which times a node should be fired in every effective iteration.

Figure 3.7. Scheduling terminology

Unlike SDF and BDF, there is no clear definition of a cyclic schedule of a DDF graph, because we do not know how many nodes should be fired to preserve the same amount of tokens on each arc after every schedule cycle. Instead we define an effective iteration, or effective cycle, as a schedule period in which one or more nodes are required to be fired as many times as predefined values, that is, firingsPerIterations.
3.2.1. Minimax Scheduler

A data driven scheduler is suggested in [Par95]. This scheduler satisfies both requirements of figure 3.6. We display the algorithm of this minimax scheduler in figure 3.8. The stopTime defines the stopping condition of schedule or the number of effective iterations. Every iteration increases numFiring by 1, and if numFiring reaches stopTime, then the scheduler terminates. Each node is associated with three; firingsPerIteration, firings, and maxout. The maxout is set only for deferrable nodes and determined as the largest number of tokens accumulated on its output arcs at a scheduling point. A default iteration of the minimax scheduler is constructed by steps 3 to 7. Within a default iteration, the scheduler finds all enabled non-deferrable nodes to execute them one by one. Only if there is no enabled non-deferrable node, it executes a deferrable node holding the smallest maxout values. Since the scheduler does not permit a specific arc from being overflowed while others are not, it satisfies boundedness property [Par95].

1. Classify all nodes as enabled or deferrable. Set numFiring zero.

2. For each node that has non-zero firingsPerIteration,
   - Clear to zero firings of node.

3. For all nodes of graph,
   - Construct enabledNonDef list that contains enabled, non-deferrable nodes.
   - Determine an enabledDef node which has the smallest maxout among deferrable nodes.

4. Fire all nodes of enabledNonDef list. When firing a node, its adjacent nodes are classified as enabled or deferrable and its firings increases by 1.

5. If none fires at 4, fire enabledDef node.

6. If none fires at 4, 5, “Deadlock occurred.” And exit.

7. For each node that has non-zero firingsPerIteration,
If firingsPerIteration > firings, go to 3.

8. If there is any arc exceeding the limit, “Buffer overflows.” And exit.

9. Increase numFiring. If numFiring < stopTime then go to 2.

Figure 3.8. Algorithm for minimax scheduler

3.2.2. Blocking Scheduler

We propose another data driven scheduler in figure 3.9, blocking scheduler. The difference between the blocking and the minimax schedulers lies in the policy to execute deferrable nodes when there is no enabled non-deferrable nodes. For blocking scheduler, each node is categorized into the following types:

- **E**: enabled non-deferrable node.
- **D**: enabled, deferrable non-source node.
- **S**: enabled, deferrable source node.
- **B**: enabled, blocked node.

A source node is a node which has no input arcs. The blocked node is defined in figure 3.2. In the list of E-type nodes, enabled non-source nodes follow enabled source nodes. The blocking scheduler controls memory requirements by setting arc capacities with two kinds of bound values:

- **hardLimit**: a maximum number of tokens allowed on each arc.
- **softLimit**: a temporal maximum number of tokens allowed on each arc.
  
  (softLimitDelta: a value added to softLimit when increasing)

The hardLimit means the permanent limit of each arc related to system resources, while softLimit may increase up to the hardLimit, regulating the number of tokens to be produced.

A default iteration is formed by steps 3 to 7. Within a default iteration, the scheduler first fires all enabled non-deferrable nodes as the minimax scheduler does. In case there is no such node, it fires all enabled and deferrable non-source node instead.
of selecting one deferrable node of smallest maxout value. If buffer requirement of an arc reaches its softLimit, the node which produced tokens on the arc, is classified as blocked node. If there exists a bounded schedule whose largest buffer requirement is $b$, this blocking scheduler will find a bounded schedule by setting the softLimit to be $b$, which satisfies the boundedness property.

1. **Classify all nodes as enabled non-deferrable, deferrable non-source, deferrable source, or blocked.** Set numFiring zero.
2. **For each node that has non-zero firingsPerIteration,**
   - Clear to zero firings of node.
3. **For all nodes of graph,**
   - Construct E, D, S, B lists.
4. **Fire all nodes of E list. When firing a node, its adjacent nodes are classified as the case of 1 and its firings increases by 1.**
5. **If none fires at 4, fire all nodes of D list.**
6. **If none fires at 4, 5, fire all nodes of S list.**
7. **If none fires at 4, 5, 6,**
   7.1. **If B is empty, “Deadlock occurred.”**
   7.2. **If softLimit >= hardLimit, then “Buffer overflows.”**
   7.3. **Let softLimit = min ( softLimit + softLimitDelta, hardLimit )**
   7.4. **Classify all nodes of B list, go to 3.**
8. **For each node that has non-zero firingsPerIteration,**
   - If firingsPerIteration > firings, go to 3.
9. **Increase numFiring, if numFiring < stopTime then go to 2.**

Figure 3.9. Algorithm for blocking scheduler

3.3. EVALUATION OF SCHEDULERS
This section analyzes two DDF schedulers introduced in section 3.2. We will select one to embed it into the generated code as the run-time scheduler. The two schedulers are compared to each other in terms of performance and memory usage by simulation. First, we formulate the memory requirements, or the total number of tokens on all arcs.

In figure 3.10, let $e_i$ an arc of a node $v$, where $E$ is a set of arcs of $v$ and $|e_i^0|$ is the number of tokens on arc $e_i$ before firing, and $|e_i^1|$ is the number of tokens on arc $e_i$ after firing. And let $V_I$ be a set of nodes fired in an effective iteration, and $V_F$ be a set of nodes fired in a complete execution of the graph. When $v$ fires, the number of tokens in a system varies by the following amount:

$$\Delta v_m = \sum_{e_i \in E} |e_i|$$

This means that the variance of tokens caused by firing a node is same as the difference of tokens on each arc between before and after firing. Let (3-1) be

$$\Delta v_m = v_m^1 - v_m^0,$$

then total variance of tokens in an effective iteration is:

$$\sum_{v_m \in V_I} \Delta v_m,$$

where $\Delta v_m = v_m^1 - v_m^0$.

(3-2) is a function of $V_I$. To obtain the maximum buffer requirement, we take the maximum value of (3-2):

$$B_f = \max_{v_m \in V_I} \sum_{v_m \in V_I} \Delta v_m.$$
$B_f$ means a maximum buffer requirement during an effective iteration. The number of remaining tokens after a complete execution is $B_F = \sum_{v=1}^{T} \Delta V_m$. We prefer a smaller $B_f$ to larger one.

Another profiling factor is $V_F$, which means the number of total nodes for a complete execution satisfying the predefined stopping condition. We prefer smaller $V_F$ satisfying the stopping condition. Finally, the number of default iterations affects the system performance. The more default iterations are executed, the more scheduling should cost.

3.3.1. Experiment

A graph in figure 3.11 shows an example for a bounded DDF graph, where the Ramp node generates a sequence \{0,1,2,3,\ldots\}, the WaveForm Star generates a periodic sequence \{0,1,0,1,0,\ldots\}, the XMgraph displays result graph. The tokens produced by WaveForm is used as control inputs of EndCase and Case which reroute their inputs or outputs. The arc from Case to Endcase holds an initial delay to avoid deadlock. This graph works well and produces the same result by both schedulers.
Even though the results are same on both schedulers, the profiling factors are different each other. The graphs in figure 3.12 illustrate the profiling results as mentioned in section 3.3.

(a). Profiling result by minimax scheduler
(b). Profiling result by blocking scheduler

Figure 3.12.

In this example, the blocking scheduler shows better results than the minimax scheduler by a little amount. In figure 3.12, the blocking scheduler requires fewer buffer requirements by one on every effective iteration than the minimax scheduler. The blocking scheduler executes fewer default iterations (153 versus 244).

Let us examine another example which is not bounded. In general, the synthesized code will run on a system with limited resources. Even though a graph is not bounded, it may require fewer buffers or more buffers depending on the scheduler.

(a). An example of unbounded DDF graph
As the minimax scheduler fires stars continually the maximum buffer requirement of figure 3.13(b) increases continually, while the blocking scheduler requires fewer buffers till there are no more enabled nodes. We can see that both schedulers require the same amount of buffers at iteration 16, 31, the blocking scheduler postpones requesting more buffers by blocking nodes before all enabled nodes are blocked.

The above results show that the blocking scheduler can work efficiently in the buffer requirement and the number of fired stars, and the number of default iterations. Thus, we choose the blocking scheduler as a run-time scheduler for the synthesized code. It is worth noting that the examples used in the experiments are very practical if we assume that simple nodes are replaced with hierarchical nodes of SDF type. As will be explained later, all SDF type nodes are clustered to use the advantages of

Figure 3.13.

(b). Profiling result by minimax scheduler

(c). Profiling result by blocking scheduler
compile-time analysis of SDF schedulers. The top level DDF graph would be much simpler once clustered.
4. SYNTHESIZING DDF GRAPH

In code generation domain, a star defines the code segments corresponding to its functionality and stitch them into the generated code stream when fired. Ptolemy provides a set of predefined stars with star libraries while the user may define and add new star to the Ptolemy environment seamlessly. In case of SDF model of computation, the execution order of nodes is determined at compile-time. Thus, the generated code consists of a list of inlined code segments according to the scheduling order of nodes. Important issues for code generation include reduction of code size and data memory requirements. For DDF model of computation, however, nodes should be scheduled at run-time. Therefore, we manage nodes as separate functions and the synthesized run-time scheduler schedules these functions. Figure 4.1 shows the abstract structure of the generated code.

In this section, we will consider as major issues how to implement and how to embed a run-time scheduler efficiently into the generated code, how to represent nodes, arcs, and connections, and how to define scheduling information. Furthermore, the buffer management, i.e. when, how and who to expand the dynamic buffer, will be considered.

4.1. REPRESENTING DATA FLOW NODES

A node structure is illustrated in figure 4.2. Each node is represented as an entry of a node structure array. Since an array structure is simple and can be accessed directly with an index, it is better than a list structure for the efficient code. An
implication of the fixed array structure is that it does not allow run-time creation of nodes. Thus, currently, we do not support recursion constructs.

```c
struct DefStar {
    int (*func)();  // (S) pointer to node function
    int succ;      // (D) link for categorizing nodes
    short flag;    // (D) node state
    int iter;      // (S) firingsPerIteration value
    int firings;   // (D) firingsPerIteration counter
    int inArcNo;   // (S) number of input arcs
    int outArcNo;  // (S) number of output arcs
    int* iArcs;   // (S) pointer to input arc descriptor
    int* oArcs;   // (S) pointer to output arc descriptor
    int waitPort; // (SD) arc id for 'waitFor' method
    int waitTokens; // (SD) number of tokens for 'waitFor'
};
```

**Figure 4.2. Structure of a node (star)**

Each information of node structure may be **static** (S) or **dynamic** (D) depending on whether it is determined at compile time or run-time. There also exists information that is determined at compile time but may be changed at run-time (SD). The flag shows node state; *Enabled, Deferrable non-source, Deferrable source, Blocked*. Each input or output arc of a node is represented as an *arc descriptor* pointed by *iArcs* or *oArcs*, as follows:

```c
int CGCEndCase_ai[3] = { 0, 1, 2 };
int CGCEndCase_ao[1] = { 3 };
```

**Figure 4.3. Arc descriptors**

From figure 4.3 we know *CGCEndCase* node has 3 input arcs, 1 output arc, which are connected to arc 0, 1, 2 and arc 3 respectively. CGCEndCase node is a DDF node in *C code generation Domain* of Ptolemy which has the same functionality as EndCase node illustrated in figure 3.4. As showed in figure 3.3, EndCase node waits at first for
a token on control input arc, which is represented by assigning the index of control input arc to \textit{waitPort}. If the indices of \textit{i0}, \textit{i1}, and control arc are 0, 1, and 2 respectively in figure 3.4, then the \textit{waitPort} of CGCEndCase is initialized to 2 at first in the synthesized code and it will be changed to 0 if the first token read from the control arc is 0, or 1 if 1 is read from the control arc. If a node needs a fixed number of tokens from each input arc like SDF nodes, the \textit{waitPort} is set to -1.

4.2. REPRESENTING ARCS

An arc plays a role as a communication channel, or a buffer between a source node and a destination node. Two kinds of buffers are provided in order to implement arcs, \textit{static buffers} and \textit{dynamic buffers}. A dynamic buffer provides the storage of tokens whose size varies at run-time. Each arc in a graph is implemented by a dynamic buffer. When the code segment is defined for a star, it is convenient to assume that arcs have fixed size of buffers, which are realized by static buffers. As a result, tokens are copied from a dynamic buffer to a static buffer and vice versa before and after execution of star function codes. We will mention this more precisely in section 4.3.1.

In a SDF graph, the number of tokens consumed from each input arc and produced onto each output arc is a fixed value that is known at compile time [Lee87a], and various scheduling policies and buffer management schemes are developed in order to make buffering more efficient for SDF graphs [Bha92a][Bha92b]. Since a DDF graph should be synthesized with a buffer manager capable of handling dynamic property efficiently, the structure of a dynamic buffer is quite important. In this paper, we implement an arc as a circular buffer with the following structure:

```c
struct DefQ {
    void* star;    // (D) pointer to Q memory
    int type;      // (S) type of token, INT,DOUBLE,COMPLEX
    int size;      // (SD) Q size in tokens
    int mDel;      // (S) number of old tokens to be used
    int R;         // (D) index for reading Q entry
    int W;         // (D) index for writing Q entry
    int srcStar;   // (S) source star id
}
```
The `destStar` points to a memory position which holds data tokens after it is allocated at run-time. The `srcStar` and `destStar` indicate the source node and destination node of an arc. A source node produces as many tokens as `srcXfer` at a firing and a destination node consumes tokens by amount of `destXfer` at a firing. The `srcXfer` is referenced to check if a node is enabled. Each time when a token is put into the queue, the writing index `W` is advanced. Since the queue is circular, the pointer is wrapped around to the `start` pointer after reaching the end of the queue. The `mDel` (maximum delay) shows how many old tokens could be kept in the queue. Even though stars generally request only current tokens from the queue, there are some stars which use old tokens. We will show an example later. The `mDel` is not changed at run-time, it is static value.

4.3. MANAGING BUFFERS

Tokens produced by a source node are transported to a destination node through an arc queue. The arc queue can not directly interface with SDF stars, because the SDF star assumes static size of buffers when generating code segments. Thus, a static buffer provides an interface between a dynamic buffer and a SDF star.

4.3.1. Static Buffers

A static buffer is determined at code generation time and used at run-time without any change in size. In figure 4.5(a), node A produces one token and node B consumes 100 tokens at a firing. We define the number of transfer, briefly, `numXfer`, as the number of tokens produced or consumed by a firing of a node; `numXfer` of node A is 1, `numXfer` of node B is 100. For node B, the value of `numXfer-1` is same as

![Figure 4.5. A static buffer](http://library.snu.ac.kr)
$mDel$ (maximum delay) because it means that node B consumes one current token plus 99 old tokens. The graph in figure 4.5(b) has the same size of static buffer as (a) while it has a different static schedule. The firing sequence {A, B} is performed in (b) and {100A, B} in (a). One static buffer is allocated for a pair of the interconnected nodes. Since a static buffer is shared by the interconnected nodes, its size should be determined as the value enough to hold tokens required by both nodes as follows:

Static buffer size of an arc =

$$1 + \max (mDel \text{ of source porthole, } mDel \text{ of destination porthole})$$  \hfill (4-1)

4.3.2. Dynamic Buffers

A dynamic buffer provides interface between nodes. The initial size of a dynamic buffer should be carefully determined, furthermore, it should be dynamically changed at run-time. Therefore, we determine the initial size of each dynamic buffer at the code generation time, and allow it to be expanded at run-time when the number of tokens accumulated on the arc exceeds the buffer size, i.e. when the buffer overflows. The larger are the initial size and the expansion size of a queue, the more stable is the queue to change the size less likely. But the program would require more resources. Since we put more emphasis on resource savings, we determine the initial buffer size just enough to guarantee that a single firing of the source node does not overwrite unread initial tokens and the buffer size is at least the number of tokens accessed by a single firing of the destination node. We can formulate the initial size of a dynamic buffer as follows:

Initial dynamic buffer size of an arc (when old tokens are not used) =

$$(s+d-1) \times G + 1 + \text{Del}, \quad \text{(if } s \text{ or } d \text{ is 1})$$

$$\max ((s+d-1) \times G + 1, \text{Del} + 1), \quad \text{(if } s \text{ nor } d \text{ is not 1})$$  \hfill (4-2, 4-3)

Initial dynamic buffer size of an arc (when old tokens are used) =

$$S + (\text{destination } mDel+1) + \text{Del}, \quad \text{(if old tokens are used})$$  \hfill (4-4)

where \( G \) is Greatest Common Divisor of \( S,D \) (\( S=s\times G \), \( D=d\times G \)) and,

\( S \): the number of tokens produced by firing source node
D: the number of tokens consumed by firing destination node

Del: delays on arc or the number of initial tokens on arc

The cases to determine the buffer size are shown in figure 4.6. If the delay value is larger than 0, writing index $W$ is initialized by the delay value by putting as many 0 values on the dynamic buffer. It is experimentally recognized that almost all nodes, especially SDF nodes, can be executed with the predetermined buffer size by equations (4-2)(4-3)(4-4).

4.3.3. Buffer Expansion

When the initial dynamic buffer size determined by (4-2)(4-3)(4-4) is not enough to hold live tokens, the buffer expands. When the buffer expands, all tokens to be accessed by the destination node in the buffer should be preserved.

Figure 4.7 depicts two cases of buffer expansion situations. We assume the buffer size is 8, tokens fill the buffer except an empty slot, $e$. The buffer overflow condition occurs when a new token is put in the empty slot and the write pointer advanced to the position of a live token. The below figures show the buffer contents after buffer expands. All old tokens, which are indicated by negative values, are kept after buffer

\[
\begin{array}{cccc}
S & D & del & initial dynamic buffer \\
2 &  3 &  1 &  5 \\
2 & 10 &  1 & 12 \\
11 & 13 &  1 & 24 \\
\end{array}
\]
expands in both cases. For buffer expansion, it is important to determine when and how to expand the buffer. There are two functions implementing the buffer expansion, putQp() and resize();

| 1. get pointer p of current writing index W. |
|   ( p = start + W * type ) |
| 2. advance W (let this be W’, and original value is in W). |
| 3. if ((size + R + W’) % size = mDel) |
|   3.1. resize Q. |
|   3.2. recalculate pointer p of W (original value). |
| 4. return p |

**Figure 4.8. Algorithm for putQp()**

The function putQp() calculates the point on which a new token will be inserted. If the buffer overflow is detected, function resize() is invoked as follows;

| 1. realloc (start, (size + sz) * type) |
| 2. copy (size - W’) entries from W’ to end position of Q |
| 3. new read point R’ = R if R < W’ (case 1) |
|   R’ = R + sz if R >= W’ (case 2) |
| 4. writing point W’ is not changed |

**Figure 4.9. Algorithm for resize()**

where, sz is an expansion size, which may be determined arbitrary. An expansion of buffer may exceed the softLimit so that a node may be executed at most once even though it potentially exceeds the buffer limit. The resize() function is implemented by using realloc(), memcpy() provided with ANSI C, because these are compatible for almost all systems. The realloc() function avoids unnecessary data moves to allocate free memory and memcpy() can perform data transfer more efficiently. To use the
memcpy() function the destination point of data move should not be overlapped with
the source point of data move. Therefore, we make the increasing size (sz) same as
current buffer size(size), as shown in figure 4.10.

Figure 4.10. Guaranteeing no overlap between source and destination

4.4. SYNTHESIZING STAR CODES

The CGC domain of Ptolemy currently supports code generation for only SDF
stars. This paper extends the CGC domain to support DDF stars, by adding a new
target called ddf-CGC target without modifying the existent code generation facility of
SDF stars.

4.4.1. SDF Star Code Synthesis

In the generated code by ddf-CGC target, an SDF star defines a function by
wrapping around the inlined code segment which is generated by an existing CGC
target in Ptolemy. Moreover, codes are added to transfer tokens between static
buffers and dynamic buffers outside the inlined code segment as shown in figure 4.11
with an example of ADD star.

```c
int Add() {
    input_0 = getDQ(star[s].iArcs[0]); // 1
    input_1 = getDQ(star[s].iArcs[1]); // 2
    /* Add Star */ // 3
    output_0 = input_0 + input_1; // 4
}
*putDQp(star[s].oArcs[0]) = output_0; // 6
```
Lines 3, 4, 5 are inlined code segment in which static buffers are named by expanding macros defined in the star code block. That is, input_0, input_1 and output_0 are static buffers, and iArcs, oArcs indicate dynamic buffers. Lines 1, 2 and line 6 are generated by ddf-CGC target, in which getDQ() function copies a token from the specified dynamic buffer and putDQp() stores a token to the current writing position of the specified dynamic buffer.

4.4.2. DDF Star Macros

Unlike SDF stars, the number of tokens consumed or produced by DDF stars can not be measured at code generation time. Since it is not feasible to use static buffers inside the star code segment, we define the new macros for handling input and output arcs of DDF stars. First, we define a new class called CGCDDFStar derived from CGCStar. DDF stars may use all macros of CGC star as a base class of DDF stars as well as new macros representing getQ, putQp, waitFor functions as follows:

\[
\begin{align*}
$get(port) & \Rightarrow \quad get$(type)Q($(arcid)); \\
$get(port,index) & \Rightarrow \quad get$(type)Q(star[s].$(baseport)[$(port0)+index]); \\
$put(port) & \Rightarrow \quad put$(type)Qp($(arcid)); \\
$put(port,index) & \Rightarrow \quad put$(type)Qp(star[s].$(baseport)[$(port0)+index]); \\
$waitFor(port) & \Rightarrow \quad star[s].waitPort = $(arcid); star[s].waitTokens = 1; \\
$waitFor(port,no) & \Rightarrow \quad star[s].waitPort = $(arcid); \\
\quad (single port) & \quad star[s].waitTokens = no; \\
$waitFor(port,no) & \Rightarrow \quad star[s].waitPort = star[s].$(baseport)[$(port0)+no]; \\
\quad (multiple port) & \quad star[s].waitTokens = 1; \\
$waitFor(port,index,no) & \Rightarrow \quad star[s].waitPort = star[s].$(baseport)[$(port0)+index]); \\
\quad star[s].waitTokens = no;
\end{align*}
\]

where, $(type) = \{ I, D, C \}$, selected by type of port, (integer, double, complex)  
$(baseport) = \{ iArcs, oArcs \}$, selected by base of port, (input or output)  
$(arcid)$ is the index of port in arc array
$\text{(port0) is the index of port in arc descriptor array}$

**Figure 4.12. DDF star macros**

The EndCase in figure 3.4 can be described using DDF macros as shown in figure 4.13 and the generated code is shown in figure 4.14.

```c
static int readyToGo = FALSE;
static int c;
if (!readyToGo) {
    c = $get(control);
    if (c < 0) {
        printf("\nEndCase: control value is negative");
        exit(1);
    }
    if (c >= NO_OF_INPUT_PORTS - 1) {
        printf("\nEndCase: control value is too large");
        exit(1);
    }
    $waitFor(input,c);
    readyToGo = TRUE;
} else {
    $put(output) = $get(input,c);
    $waitFor(control);
    readyToGo = FALSE;
}
```

**Figure 4.13. EndCase star description**

```c
{ /* star tt.EndCase.input-21 (class CGCEndCase) */
    static int readyToGo = FALSE;
    static int c;
    if (!readyToGo) {
        c = getIQ(11);
        if (c < 0) {
            printf("\nEndCase: control value is negative");
            exit(1);
        }
        if (c >= NO_OF_INPUT_PORTS - 1) {
            printf("\nEndCase: control value is too large");
            exit(1);
        }
        star[s].waitPort = star[s].iArcs[1+c]; star[s].waitTokens = 1;
        readyToGo = TRUE;
    } else {
        *putDQp(star[s].oArcs[0]) = getDQ(star[s].iArcs[1+c]);
        star[s].waitPort = 11; star[s].waitTokens = 1;
        readyToGo = FALSE;
    }
```
4.5. RUN-TIME SCHEDULER

The run-time scheduler is implemented with the blocking scheduler verified by simulation as discussed in section 3.2.2. Most data structures which the run-time scheduler accesses are implemented in array, such as star array, arc array, arc descriptor array for accelerating execution speed. Specially, since it needs to categorize stars on every default iteration, the efficient categorization method is devised. The algorithm and data structure in figure 4.15 demonstrate a way of handling categorization.

1. initialize categorization header: \( E = D = S = B = -1 \).
2. \( i = \) number of stars - 1.
3. for star \( i \) of star array.
   
   3.1. if flag = Enabled, \( \text{succ} = E, E = i \).
   
   3.2. if flag = Deferrable non-source, \( \text{succ} = D, D = i \).
   
   3.3. if flag = Deferrable source, \( \text{succ} = S, S = i \).
   
   3.4. if flag = Blocked, \( \text{succ} = B, B = i \).
4. decrease \( i \), go to 3 if \( i \geq 0 \).

Figure 4.15. Star categorization algorithm

<table>
<thead>
<tr>
<th></th>
<th>Stars</th>
<th>succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>0</td>
<td>.....</td>
</tr>
<tr>
<td>Deferrable</td>
<td>1</td>
<td>.....</td>
</tr>
<tr>
<td>Source</td>
<td>2</td>
<td>.....</td>
</tr>
<tr>
<td>Blocked</td>
<td>3</td>
<td>.....</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>.....</td>
</tr>
</tbody>
</table>

Figure 4.16. An example of categorization
For only one pass, all stars are scanned backward and 4 categories are built. This implementation avoids performance degradation by using fixed links instead of list structure incurring much overhead for memory allocation.
5. CLUSTERING SDF NODES

It is very inefficient that the run-time scheduler executes each SDF node one by one because each node function should be invoked and data tokens should be transferred explicitly whenever a node is executed. If adjacent SDF nodes are grouped or clustered, they can be scheduled more efficiently by static schedulers. A clustered node behaves as an atomic SDF star with respect to the outside dynamic scheduler while the function code schedules the inside SDF stars statically. Attention should be taken when clustering SDF nodes in order not to make the graph deadlocked. To make a cluster between two SDF stars, it confirms that there is no delay between two stars and there is no alternate path between them. Hiding a delay may deadlock the graph. If there is an alternate path, clustering also forms a deadlocked graph as shown in figure 5.1. An example graph in figure 5.2 is clustered to the graph in figure 5.3.

Figure 5.1. Clustering makes the graph deadlocked.
Figure 5.2. A graph before clustering

Figure 5.3. A graph after clustering
6. EXPERIMENT

An example in figure 6.1 was executed by the implementation of this paper. It models a baud-rate timing recovery in a modem using an approximate minimum mean square error (MMSE) method. An amplitude-shift keyed (ASK) signal is used as source signal and the baud-rate samples are generated in figure 6.2, which are getting stable without error gradually. This example was presented in [Buc91].

Figure 6.1 shows a hierarchical view of Ptolemy. Graph A shows top level of this example which includes three galaxies; graph B, C, D. Only graph A contains DDF
stars, i.e., two Cases and one EndCase stars. Even though almost all stars are SDF, this example cannot be synthesized in the manner of SDF schedule. However, the clustering method compensates for loss of efficiency in DDF model. It works well and produces the same result as one from the simulation domain. The generated codes before and after clustering produce the same results. The code list generated by clustering is attached at appendix.

Figure 6.2. The results by figure 6.1.
7. CONCLUSION

Dynamic data flow model is investigated as a flexible, useful computational model to extend the Synchronous data flow model and the Boolean-controlled data flow model. The blocking scheduler is proposed as a DDF scheduler for the run-time system as well as for simulation environment. For synthesis of DDF graphs, the run-time scheduler is built with the efficient dynamic buffer manager and the suitable data structures for dynamic environments, and these are implemented in Ptolemy by adding a new target which automatically produces the application specific run-time system and by adding new macros to create fully dynamic data flow stars. The generated codes are confirmed by experiments.

Therefore, any graph representing an application by using SDF stars and DDF stars can be translated into C code. Plenty of SDF stars already built for DSP applications can be used for complex applications more flexibly with versatile DDF stars capable of representing control structures. DDF synthesis has made Ptolemy balanced between the simulation domain and the code generation domain, which means that any DDF application can be translated into C code after it is verified in the DDF simulation domain. Furthermore, DDF synthesis gives easier and more efficient way in not only designing but also debugging an application by reusing the verified functional blocks.

Even though the run-time kernel has been built, it requires fine tuning. Currently, the code block of an SDF star may appear many times in a clustered function of the generated code, thus, the size of code generated by clustering is larger than one by no clustering. This inefficiency can be overcome by using more efficient SDF scheduler such as loop schedulers in [Bha92a]. In addition, more efficient method to allocate buffers should be researched further. Since static buffers are currently used as temporary storage in a function block, the code size could be reduced explicitly if they are shared between function blocks.
REFERENCES


