DATAFLOW SYNTHESIS AND VERIFICATION FOR PARALLEL OBJECT-ORIENTED PROGRAMMING LANGUAGES

SHUANG WU

Dataflow Synthesis and Verification for Parallel Object-Oriented Programming Languages

by Shuang Wu

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Engineering

Faculty of Engineering and Applied Science

January 2011

St. John's Newfoundland

Abstract

The HARPO project aims to develop a methodology to generate and verify hardware configurations from a high level object-oriented programming language. Specifically, the compiler of a high-level object-oriented programming language, HARPO/L (standing for HARdware Parallel Objects Language), outputs hardware configurations that are mappable to a coarse-grained reconfigurable architecture (CGRA) system.

This thesis develops a data flow synthesis method, which is a critical component in the middle-module of the HARPO/L compiler. This method is extendable to most other high-level parallel object-oriented programming languages.

In addition, this thesis proposes an automatic verification system for HARPO/L. An algorithm to compute weakest liberal precondition of parallel compositions, which fills the gap between verification of programming languages with parallel compositions and state-of-art automatic verification approaches, is introduced. This algorithm also helps verifying the absence of data race and the absence of deadlock, and has good interplay with grainless semantics.

Acknowledgements

I am heartily thankful to my supervisor, Dr. Theodore S. Norvell, whose encouragement and guidance from the initial to the final level enabled me to develop an understanding of computing and programming theories.

I am also grateful to all my instructors, colleagues and classmates in Memorial University of Newfoundland, who supported me in many respects during my study, my research and the completion of this thesis.

Finally, I dedicate this thesis to my parents who have given me the opportunity of an education in Canada and support throughout my life.

Contents

List of Figures

Chapter 1

Introduction

1.1 High Level Programming on CGRA

Traditional computing descriptions are classified into two kinds, structural descriptions (or hardware descriptions) such as Application Specified Integrated Circuit (ASIC) and behavioural descriptions (or software descriptions) expressible in high-level programming languages, which are performed on microprocessors. ASICs are designed for particular computations, so they have benefits on efficiency, but meanwhile they have very poor flexibility. Microprocessors are much more flexible: different combinations of the instructions can complete different computational tasks without any modification to the hardware. However, the loading of instructions from the memory and the decoding of the instructions bring great overhead, so the efficiency of microprocessors is much lower.[1]

Reconfigurable computing attempts to be a compromising solution with higher efficiency than software and higher flexibility than hardware. In the reconfigurable devices, such as field-programmable gate arrays (FPGAs), the computations are performed by an array of computational logic blocks (CLBs). The CLBs' functionality is programmable through configuration bits. The CLBs are connected by interconnection resources whose routing switches are also programmable.[2]

In reconfigurable computing, the efficiency is closely related to the granularity (the ratio of the amount of computation to the amount of communication). A fine-grained system has a larger amount of smaller primitive computations and requires more communication, while a coarse-grained system's primitive computations are larger so that it requires less communication. Because of the huge amount of communications between CLBs through interconnections and routing switches, FPGA is a fine-grained architecture whose efficiency is relatively low due to the overhead brought by routing. Contrasted with FPGA and other fine-grained systems, coarse-grained reconfigurable architectures (CGRAs) use computational function blocks (CFBs) to build reconfigurable datapath units (rDPUs) to perform coarse-grained computations.[3]

For instance, as a typical CGRA, the RaPiD architecture (Fig 1.1) [4] is composed of ALUs, multipliers, registers, RAM blocks and other functional units (FUs). All the FUs are attached to a programmable bus, and perform a pipeline-style communication with each other through registers. The instructions from an input stream are decoded and flow through the datapath, and the data and intermediate results are locally stored in registers and small RAMs which are close to their destination FUs. As a result, this coarse-grained architecture allows very small communication overhead and thus can obtain higher efficiency than fine-grained architectures.

According to [4], usually the programming of reconfigurable architectures is in low-level languages such as hardware description language (HDL) and assembly lan-

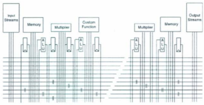

Figure 1.1: The RaPiD Architecture

guage. Although there are a number of behavioural languages such as VHDL (VHSIC HDL where VHSIC means Very High Speed Integrated Circuit) and C, none of the existing languages has as high level as object-oriented programming languages. The HARPO (standing for HARdware Parallel Objects) project aims to define a high-level object-oriented programming language (HARPO/L) which can be compiled into coarse-grained hardware configurations. It will provide a high-efficiency high-flexibility solution to the computations described in high-level object-oriented programming languages.

1.2 Overview of HARPO/L

The essential ideas of HARPO/L are: (1) the creations of objects start threads processing operations on their fields, (2) the objects are mapped into hardware configurations as individual rDPUs, and (3) the references and method calls are considered as interconnections between rDPUs. Accordingly, HARPO/L must have following features.

- Static: The allocations are done at compile-time. Dynamic allocating and referencing are not allowed.
- Concurrent: The threads in all the objects shall be concurrently executed. Besides, the language supports parallel compositions.
- Grainless Semantics: HARPO/L shall have no assumption of granularity and no restriction on implementation, so it shall have grainless semantics.

The compilation/synthesis flow[4] of HARPO/L is shown in Fig 1.2. The front-end[5] does type checking, and generates abstract syntax trees (ASTs) and an object graph from the source code. Then the middle module generates dataflow graph. Finally the back-end[6] generates hardware configurations.

1.3 Contribution and Thesis Outline

This thesis addresses two problems. The first is how to generate dataflow graphs from given abstract syntax trees and object graphs, and the other is how to automatically verify the partial correctness of HARPO/L programs.

The contributions of this thesis include (1) an extension of Static Token data flow synthesis method[7] which is applicable on HARPO/L and extendable to other parallel object-oriented programming languages, and (2) the architecture of an automatic verification system of HARPO/L which is extendable to other parallel object-oriented programming languages, with an algorithm to compute the weakest liberal precondition of parallel compositions.

Chapter 2

Related Work

2.1 HARPO/L Language Design

This subsection will briefly describe the syntax of HARPO/L. The details can be found in [8]. First, I will give some metanotations which are used later.

$N \to E$	Nonterminal N can be an E
$\lceil E \rceil$	Grouping
E^*	Zero or more
E^{*F}	Zero or more separated by Fs
E^+	One or more
E^{+F}	One or more separated by Fs
$E^?$	Zero or one
$E \vert F$	Choice

A HARPO/L program consists of a set of classes, interfaces, objects, and constants. The class declarations and interface declarations add new types to the type system, and the object declarations and constant declarations add objects to the object graph. The details of object declarations and constant declarations are similar to other object-oriented programming languages and will not be listed in this thesis.

$$
\begin{aligned}
\mathit{program} &\rightarrow \lceil \mathit{ClassDecl} | \mathit{IntDecl} | \mathit{ObjectDecl} | \mathit{ConstDecl} |; \rceil^{*} \\
\mathit{ObjectDecl} &\rightarrow \mathbf{obj}\ \mathit{Name}\ \lceil :\ \mathit{Type} \rceil^{?} := \mathit{InitExp} \\
\mathit{ConstDecl} &\rightarrow \mathbf{const}\ \mathit{Name}\ \lceil :\ \mathit{Type} \rceil^{?} := \mathit{ConstExp}
\end{aligned}
$$

The classes and interfaces may be generic or nongeneric. The generic classes and interfaces can be parameterized by other nongeneric types. Each class has a constructor method with a list of constructor parameters representing objects to which this object is connected.

$$
\begin{aligned}
\mathit{IntDecl} &\rightarrow \Big(\mathbf{interface}\ \mathit{Name}\ \mathit{GParams}^{?}\ \lceil \mathbf{extends}\ \mathit{Type}^{+}\ , \rceil^{?} \\
&\qquad \lceil \mathit{IntMember} \rceil^{*}\ \Big\lceil \mathbf{interface}\ \lceil \mathit{Name} \rceil^{?} \Big\rceil^{?} \Big) \\
\mathit{ClassDecl} &\rightarrow \Big(\mathbf{class}\ \mathit{Name}\ \mathit{Gparams}^{?}\ \lceil \mathbf{implements}\ \mathit{Type}^{+}\ , \rceil^{?} \\
&\qquad \mathbf{constructor}\ \big(\mathit{CPar}^{+}\ ,\big)\ \lceil \mathit{ClassMember} \rceil^{*}\ \Big\lceil \mathbf{class}\ \lceil \mathit{Name} \rceil^{?} \Big\rceil^{?} \Big) \\
\mathit{CPar} &\rightarrow\ \mathbf{obj}\ \mathit{Name} : \mathit{Type} |\ \mathbf{in}\ \mathit{Name} : \mathit{Type} \\
\mathit{GParams} &\rightarrow \big\{ \mathit{GParam}^{+} \big\} \\
\mathit{GParam} &\rightarrow\ \mathbf{type}\ \mathit{Name}\ \lceil \mathbf{extends}\ \mathit{Type} \rceil
\end{aligned}
$$

Interface members can be fields, methods, and constants, and class members can be fields, methods, threads, and constants. Fields can be either private or public, and can be declared as a specific type or be automatically assigned to the type of the initial expression. Methods can be either private or public.

$$
\begin{aligned}
\mathit{IntMember} &\rightarrow \mathit{Field}|\mathit{Method}|\mathit{ConstDecl}|; \\
\mathit{ClassMember} &\rightarrow \mathit{Field}|\mathit{Method}|\mathit{Thread}|\mathit{ConstDecl}|; \\
\mathit{Field} &\rightarrow \mathit{Access}\ \mathbf{obj}\ \mathit{Name}\ \lceil :\ \mathit{Type}\rceil^{?} := \mathit{InitExp} \\
\mathit{Method} &\rightarrow \mathit{Access}\ \mathbf{proc}\ \mathit{Name}\left(\left\lceil \mathit{Direction}\ \lceil \mathit{Name} :\rceil^{?}\ \mathit{Type}\right\rceil^{*\,,}\right) \\
\mathit{Access} &\rightarrow\ \mathbf{private}\ |\ \mathbf{public} \\
\mathit{Direction} &\rightarrow\ \mathbf{in}\ |\ \mathbf{out}
\end{aligned}
$$

Types can be names of classes, array types, or specializations of generic types.

$$
\begin{aligned}
\mathit{Type} &\rightarrow \mathit{Name}|\mathit{Name}\ \mathit{GArgs}|\mathit{Type}\,[\mathit{Bounds}] \\
\mathit{GArg} &\rightarrow \{\,\mathit{Type}^{+\,,}\} \\
\mathit{Bounds} &\rightarrow \mathit{ConstIntExp}
\end{aligned}
$$

Threads consists of Statements which are executed once the object is instantiated. Specifically, each thread has a block statement which represents the sequential composition of the statements.

$$
\begin{aligned}
\mathit{Threads} &\rightarrow \left(\mathbf{thread}\ \mathit{Block}\ \lceil \mathbf{thread}\rceil^{?}\right) \\
\mathit{Block} &\rightarrow \mathit{Statement}^{+}
\end{aligned}
$$

Beside sequential compositions (**block**), statements can be local variable declarations, constant declarations, assignments, method calls, sequential control flows (**if**, **while**, or **for**), parallelisms (**co**), lockings (**with**), and method implementations

Initialization of an object can be an expression or an array initialization

$$InitExp \rightarrow Expresion | ArrayInit | \textbf{ new } Type\ (CArg^{+}\ \cdot)$$
$$|\ (\textbf{if } Expression \textbf{ then } InitExp$$
$$[\textbf{else if } Expression\ InitExp]^{*} \textbf{ else } InitExp \textbf{ if})$$
$$ArrayInit \rightarrow (\textbf{for } Name : Bounds \textbf{ do } InitExp \textbf{ for})$$
$$CArg \rightarrow Expression$$

In HARPO/L, method calls are implemented in the threads of objects, and play a role in thread synchronization. The rendezvous[9][10] mechanism is used to realize this functionality. In client's view, rendezvous is almost the same as method calls in other high level object-oriented programming languages such as Java and C++. In server's view, rendezvous is an accept statement; when the thread reaches the rendezvous, the thread waits until the implemented method is called. In other words, not only the server has to wait for the client's thread to reach the rendezvous, but also the client has to wait for the server's thread to reach the rendezvous. The rendezvous mechanism also provides guards in accept statement; the client has to wait for the server to reach the rendezvous with a true value guard. If the guard is false when the server reaches the rendezvous, the client has to wait for the server to reach the rendezvous again and then re-evaluate the guard.

2.2 Grainless Semantics of HARPO/L

Grainless semantics[11] is introduced for shared-variable concurrent programs with smallest granularity[12] in which none of the operations is considered atomic. The word "grainless" means that programs with data races are simply considered to have a semantics of "**wrong**", i.e. not to have any useful meaning. Because HARPO/L is a programming language in which the programs are compiled into hardware configurations, and small granularity is a nature of hardware, HARPO/L needs to have a grainless semantics.

The grainless semantics of HARPO/L is given in [12]. Because HARPO/L is a static language, the object instantiation and connection are done at compile-time, and there is no reference/pointer assignment in run-time. The context of HARPO/L contains two parts, static allocations and state commands. The semantics of the state commands is based on the approach in [11].

In the semantics of the state commands, all the commands are translated into a finite or infinite sequence of primitive actions. The primitive actions include **start**, **fin**, **chaos**, **try**, **acq**, and **rel**. Some non-primitive actions are also defined such as **filter** and **enter**. The detailed meanings of the actions can be find in [12] or [13].

2.3 HARPO/L Compiler Front-End

The object graph is one of the outputs of the HARPO front-end (others include a type system, which is not involved in this thesis, and the ASTs) which is based on the grainless semantics of HARPO/L[12]. There are 7 types of object graph nodes: *Constant*, *Object*, *Array*, *Location*, *Variable*, *Method*, and *Thread*. The difference between *Location* and *Variable* is that each *Location* node is associated with a memory address while the *Variables* are not. Specifically, the fields with primitive types, the array elements with primitive types, the shared variables in parallel compositions, and the arguments of the methods are *Locations*, and the local variables which are only accessed by at most one thread are *Variables*.

A complete object graph has a root *Object* whose fields are the public *Objects* declared in the program. Each *Object* has a set of *Constants*, *Locations*, or *Variables*, as primitive fields, a set of *Arrays* as array fields, a set of *Objects* as reference fields, a set of *Methods*, and a set of *Threads*. An *Array* has an integer *size*, and has a set of elements of *Array* type, *Object* type, or *Location* type. A *Location* node has an integer *address*. A *Variable* node has a name. A *Method* has a *MethodType* which provides the information of method name, arguments, return values, etc. A *Thread* has a set of local *Variables* or *Constants*, and has a AST of the statements in the thread.

The *Locations*, *Variables*, and *Methods* can be accessed with an expression (which is a sequence of names connected by "."). The object graph interface provides a number of methods to get accesses to these nodes: *location*(expression), *variable*(expression), *method*(expression).

The AST of a thread in HARPO/L[5] has a root node, and all the tree nodes are *Statement* nodes, or *Expression* nodes.

Before generating the ASTs, the front-end normalizes all the statements. All the *FOR Statements* are transformed into *WHILE Statements*, and all the *COLOOP Statements* are transformed into *CO Statements*. Each *ACCEPT* body contains a *guard*, a set of *arguments*, a *body*, and an *afterbody*; *IF Statements* with multiple else-if-clauses are transformed into multiple *IF Statements*; each *IF Statement* has a *guard*, a *then*-clause, and an *else*-clause; and each *WITH* Statement has a *guard* and a *body*.

Figures 2.1 to 2.8 show the ASTs of 8 types of *Statements*. A block with an arrow under it (such as the "acceptbodies" block in the abstract syntax tree of *ACCEPT Statement*) represents a set (a List in the front-end implementation) of AST nodes.

The *Expression* nodes are divided into a number of types. The *Expression* types used in this thesis are *NULL, IDENTIFIER, REFERENCE, INDEX, LITERAL,*

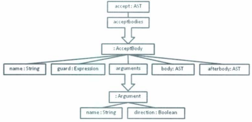

Figure 2.1: Abstract Syntax Tree of *ACCEPT Statement*

Figure 2.2: Abstract Syntax Tree of *ASSIGN Statement*

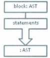

Figure 2.3: Abstract Syntax Tree of *BLOCK Statement*

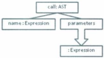

Figure 2.4: Abstract Syntax Tree of *CALL Statement*

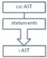

Figure 2.5: Abstract Syntax Tree of *CO Statement*

Figure 2.6: Abstract Syntax Tree of *IF Statement*

Figure 2.7: Abstract Syntax Tree of *WHILE Statement*

Figure 2.8: Abstract Syntax Tree of *WITH Statement*

NEG, and *MATH*. There are some other types of *Expressions* used in the type system. *Expressions* of *NULL* type are null; *Expressions* of *IDENTIFIER* type are names; *Expressions* of *REFERENCE* type have a left child *Expression* and a right child *Expression* connected by operator ".": *Expressions* of *INDEX* type are the array elements, and have a left child *Expression* identifying the array and a right child *Expression* representing the index; *Expressions* of *LITERAL* type are constant values; *Expressions* of *NEG* type has a right child *Expression*, and means the negative value of that child *Expression*; *Expressions* of *MATH* type are expressions with one operator and two operands. If an *Expression* does not have left child or right child, it is considered to have a *NULL* left child or a *NULL* right child.

Chapter 3

Data Flow Synthesis of Parallel Object-Oriented Programs

In this chapter I propose a method to synthesize data flow graphs for parallel object-oriented programs. Through this synthesis, the AST of a program is transformed into a data flow graph which is very close to the representation of a schedulable datapath unit. The data flow graph will be scheduled into a hardware configuration by the back-end of a HARPO/L compiler.

3.1 Background

The essential point of data flow analysis is to find all the use-definition chains for each use of the variables.[14] A definition of a variable is a write access to that variable, and a use of a variable is a read access to that variable. The link from the use to a definition is called use-definition chain which indicates the data flow of the used/defined variable. In a use-definition chain, we also say that the definition reaches the use.

The high-level data flow analysis is applicable on programs in high level programming languages which are well structured and contain no *escape* or *goto* statements.[14]

The idea is to perform two passes of computations on each statement. The first pass computes the variable sets that are the non-context property of the statement, such as what variables are used, what variables are defined, and so on. Different approaches may have different sets to compute. For instance, the approach in [7] computes used variable set *use* and defined variable set *def*; [14] chooses used variable set *IN* and used-or-defined variable set *THRU*; in [15], *defsout*, a set of variables that are used or defined, and not killed, and *killed*, a set of variables that are used or defined, and killed, are computed along with a used variable set *use*; etc. These computations must be bottom-up because the sets of control flow statements (branch and loop) depend on the sets of the nested statements (then/else-clauses or loop bodies) which are their descendants in the AST. The second pass computes the variable sets that relates to the context, such as what variables are live before executing the statements according to the previous program context, what variables are live after executing the statements. Common choices of defining the second pass variable sets include *livein* (live before) and *liveout* (live after) such as in [14], [7], etc.

Since the only control flows are branch and loop, in any statement, a live variable (a variable that has been assigned and has not been killed before) either comes from the following part of the loop body if the statement is in a loop body, or comes from the previous statements, so the control flow analysis is unnecessary, and the data flow analysis can be done directly on a sequence of statements.

3.2 Related Work

In the state-of-art data flow analysis techniques, the sequence of statements is in an intermediate format for the convenience of the data flow synthesis.

The Static Single Assignment (SSA) form[16] adds extra assignments for the variables which are assigned in one or both of the branches after the branch control flow (considering the merges of the control flow, the loop control flow is treated as a special branch control flow that one branch is the loop body and the other is the previous statements). The assigned variables in the branches are renamed and those extra assignments assign the variable with the old name to the value of ϕ-function on the renamed variables which guarantees the further uses will find only one previous definition (they will find the definition with the ϕ-function instead of the definitions in the branches). Thus, the uses of all the variables can find only one previous definition. For example, if a variable x is defined in both branches of a branch control flow, all the appearances of x in one branch are renamed, say, as x_0, and all the appearances of x in the other branch are renamed as x_1; both x_0 and x_1 are defined by x at the beginning of the branch control flow, and at the end of the branch control flow, there is an extra assignment that $x := \phi(x_0, x_1)$ which means x is assigned to either x_0 or x_1. All the uses of x in the branches, which have been renamed as x_0 or x_1, will find a definition of x_0 or x_1 to link the use-definition chain, and all the uses of x after the branch control flow will still find a definition of x.

The Static Single Information (SSI) form[17] is an extension of SSA form. In SSI form, σ-function is defined as the inverse function of ϕ-function. The renaming is also applied on the variables that are used in the branches and the extra assignments with σ-function, such as $(x_0, x_1) := \sigma(x)$, are added in front of the branch control flow. SSI form guarantees that for a use-definition chain, the path from the definition to the use in the program is determined (contains no branch or loop control flow). In addition, a loop control flow is treated as not only the merge of the control flow for the loop's defined variables, but also the split of the control flow for its used variables.

The Static Token (ST) form[7] is an extension of SSI form. In ST form, each ϕ-function and σ-function is given a subscript indicating the choice. The choice can be either a constant or an expression on the variables. The same article [7] also shows how to synthesize a data flow graph from a program. Seven kinds of data flow graph nodes are defined, and they use these seven kinds of nodes as primitive operations to construct a sequence of program which is equivalent to the original program.

In [7], the original program and the definitions of the nodes are in CHP[18] of which I will give a summary here. The send operation $Z!a$ means "data a is transmitted to edge Z"; the receive operation $A?a$ means "wait until data a is received from edge A"; both send and receive are synchronous. Assignment operation $a := b$ means "assign the value of b to a"; Boolean assignment operations $B\uparrow$ and $B\downarrow$ mean "assign $\mathfrak{true}$ / $\mathfrak{false}$ to B". Selection structure $[G_0 \rightarrow S_0 [] \ldots [] G_{n-1} \rightarrow S_{n-1}]$ means "wait until one of the guards is $\mathfrak{true}$ and then execute the corresponding operations" where $G_0, \ldots, G_{n-1}$[1] are the guards, and $S_0, \ldots, S_n - 1$ are the operations. Repetition structure $*[G_0 \rightarrow S_0 [] \ldots [] G_n - 1 \rightarrow S_n - 1]$ means "choose one of the $\mathfrak{true}$ guards and execute the corresponding operations, and then repeat this until all guards are $\mathfrak{false}$". The angle brackets $\langle\rangle$ mean atomic operations. The semicolons indicate sequential compositions, and the commas indicate parallel compositions.

In SSA, SSI, or ST approach, the reason for renaming and inserting extra assignments before the data flow analysis is to preserve the data flow information for generating the graph, because the two pass data flow analysis and data flow graph

[1]In this thesis, the notation $f(i), \ldots, f(k)$ means functions f on integers from i to k (inclusively); the notation $f(i), \ldots f(k)$ means functions f on integers from i to $k-1$. e.g. $i, \ldots i+2$ means i and $i+1$; $G_0, \ldots, G_2$ means G_0, G_1, and G_2.

generation are considered as two separated steps. If we could combine the analysis and the graph generation, renaming and extra assignment inserting would be unnecessary: the data flow of both ϕ-function and σ-function is generated along with other definitions and uses of the variables.

In the following subsections, I show how to synthesize the data flow of programs in HARPO/L, which is an object-oriented programming language with concurrency. Because the semantics of HARPO/L contains a number of complicated control flow structures such as locks, I treat the control flow as a special kind of data flow, and the data flow graphs generated are a mixture of control flow and data flow and are executable: the activeness of all the non-control data flows are controlled by the control flows which represent both the paths of control signals in hardware and the execution of the program. The nodes in data flow are given definitions in CHP and are simple enough to implement in hardware, and the behaviour of the executable data flow is equivalent to the grainless semantics[12] of the original program in HARPO/L.

3.3 Overview of Dataflow Graph for HARPO/L

A dataflow graph is a directed graph represented by a tuple $(N, E, type, I, O)$ where N is a set of nodes, E is a set of directed edges, *type* is a function: $N \rightarrow NodeTypes$, I is a node representing the start of the graph, and O is a node representing the end of the graph. Each node has an ordered set of input edges and an ordered set of output edges, and each edge has exactly one source node and exactly one target node.

The directed edges between dataflow graph nodes are divided into two kinds: $E = C \cup D$ where C is a set of control flow edges and D is a set of data flow edges.

A data flow edge represents the synchronized transmission of a primitive value between dataflow graph nodes. When a node is receiving data from an edge, it is waiting for the edge being active, and once the edge is, the node will receive the data and set the edge's activeness expired; when a node is transmitting data to an edge, it will transmit the data and set the edge active. The control flow edges are the edges transmitting only the activeness and no data.

There are 13 types of dataflow graph nodes. The graphic representations are shown in Fig 3.1. The behaviour of each type of nodes is described in CHP notation. In addition, I define that control flow send operation $Z!$ means "activate control flow edge Z" and that control flow receive operation $A?$ means "wait until edge A is active, and set the activeness expired".

$$\mathrm{START} \equiv Z!$$

$$\mathrm{VALUE} \equiv *[Z!\text{``constant''}]$$

$$\mathrm{INIT} \equiv Z!\text{``constant''}; *[A?a; Z!a]$$

$$\mathrm{SINK} \equiv A?$$

$$\mathrm{COPY} \equiv *[A?a; Z_0!a, \ldots, Z_{n-1}!a]$$

$$\mathrm{MERGE} \equiv *[C?c; A_c?a; Z!a]$$

$$\mathrm{SPLIT} \equiv *[C?c, A?a; Z_c!a]$$

$$\mathrm{FETCH} \equiv *[A?; a := \mathrm{fetch}\,(); Z!a]$$

$$\mathrm{STORE} \equiv *[C?, A?a; \mathrm{store}\,(a); Z!]$$

$$\mathrm{FUNC} \equiv *[A_0?a_0, \ldots, A_{n-1}?a_{n-1}; Z!f\,(a_0, \ldots, a_{n-1})]$$

$$\mathrm{JOIN} \equiv *[A_0?, \ldots, A_{n-1}?; Z!]$$

$$\begin{aligned}\text{MULTI-LOCK} \equiv *[&A?, C_0?c_0, \ldots, C_{n-1}?c_{n-1};\\ &\langle [c_0 \wedge \neg\mathrm{lock}_0 \rightarrow \mathrm{lock}_0 \uparrow, d := 0 \,[]\, \ldots \,[]\\ &c_{n-1} \wedge \neg\mathrm{lock}_{n-1} \rightarrow \mathrm{lock}_{n-1} \uparrow, d := n-1] \rangle ; D!d, Z_d!]\end{aligned}$$

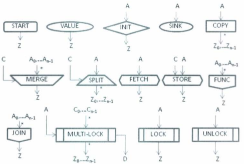

Figure 3.1: Dataflow Graph Nodes

$$\text{LOCK} \equiv *\,[A?; \langle [\neg\text{lock} \rightarrow \text{lock}\uparrow] \rangle\,; Z!]$$

$$\text{UNLOCK} \equiv *\,[A?; \text{lock}\downarrow; Z!]$$

Additional comments are listed below:

- MULTI-LOCK: each MULTI-LOCK node is associated with a number of locks, and Boolean variables $\text{lock}_0, \ldots, \text{lock}_{n-1}$ indicate whether the locks are free.
- LOCK and UNLOCK: each LOCK or UNLOCK node is associated with a lock, and Boolean variable lock indicates whether the lock is free.
- FETCH: Each FETCH node is associated with a location. The operation fetch()

means "fetch the value in the location".

- STORE: Each STORE node is associated with a location. The operation store(a) means "store the value of a in the location".
- FETCH and STORE: If the associated location is represented by an *INDEX Expression* with non-constant index(s), the node will have additional input edge(s) providing the evaluation of the index(s).

Note that the definition of data flow graph is different with the definition of executable data flow graph in [6], although they are very similar. The *inRole* in [6] is the edge before I, and the *outRole* in [6] is the edge before E. In the later subsections it is shown that I and E are always *COPY* nodes which have only one edge before them. Therefore, the results of my data flow synthesis can be used as an input of the back-end described in [6].

3.4 Generating Dataflow Graph for HARPO/L

The dataflow graph generation takes an object graph and an AST of a thread from the front-end as its inputs, and uses a high level dataflow analysis algorithm with two passes. In this subsection, I will call all the local variables "*Variables*", and all the shared variables "*Locations*" (because they require real memory locations in the hardware configurations). In addition, I will use the traditional definitions of "definition" and "use": a definition is an assignment of some value to a *Variable* or a *Location*; and a use is a read-only access to a *Variable* or a *Location*.

3.4.1 The First Pass

The first pass computes *syn*, *useLoc*, *defVar*, *defLoc*, and *defVar* for each statement according to its AST. The definition of these five functions are:

$$syn : Statement \rightarrow Boolean$$
$$useLoc : Statement \rightarrow (Location \rightarrow Boolean)$$
$$useVar : Statement \rightarrow (Variable \rightarrow Boolean)$$
$$defLoc : Statement \rightarrow (Location \rightarrow Boolean)$$
$$defVar : Statement \rightarrow (Variable \rightarrow Boolean)$$

The *syn* function indicates whether the *Statement* has synchronization in it. The *useLoc*/*useVar* functions give the set of *Locations*/*Variables* used and without prior definitions in the *Statement*. The *defLoc*/*defVar* functions give the set of *Locations*/*Variables* potentially defined and not killed in the *Statement*. A synchronization will kill the definitions of all the *Locations*. For example, suppose S is the following *IF Statement*.

(**if** $Var_0 < Loc_0$ **then**
  $Stmt_0 : Loc_1 := Var_0$
  $Stmt_1 : Var_1 := Loc_1$
  $Stmt_2 :$ (**with** $Lock_0$ **when true do** $Var_0 := Loc_2$)
  $Stmt_3 : Loc_3 := Loc_4$
  $Stmt_4 :$ (**with** $Lock_1$ **when true do** $Loc_5 := Var_1$)
  $Stmt_5 : Loc_5 := Var_2$
else $Stmt_6 : Var_3 := Loc_5$ **if**)

The first pass results are

$$syn(S) = \text{true}$$
$$useLoc(S) = \{Loc_0, Loc_5\}$$
$$useVar(S) = \{Var_0, Var_2\}$$
$$defLoc(S) = \{Loc_5\}$$
$$defVar(S) = \{Var_0, Var_1, Var_3\}$$

Note that $Loc_1 \notin useLoc(S)$ although $Stmt_1$ uses it, because Loc_1 is defined in $Stmt_0$ (prior definition), and $Var_1 \notin useVar(S)$ for a similar reason; and $Loc_3 \notin defLoc(S)$ although $Stmt_3$ defines it, because $Stmt_4$ has a synchronization that kills that definition.

The computations need four other functions: *expUseLoc*/*expUseVar*, the set of *Locations*/*Variables* used in an *Expression*, and *indexUseLoc*/*indexUseVar*, the set of *Locations*/*Variables* used in the indexes of the array sub-Expressions in an *Expression*.

$$expUseLoc : Expression \rightarrow (Location \rightarrow Boolean)$$
$$indexUseLoc : Expression \rightarrow (Location \rightarrow Boolean)$$
$$expUseVar : Expression \rightarrow (Variable \rightarrow Boolean)$$
$$indexUseVar : Expression \rightarrow (Variable \rightarrow Boolean)$$

The computations of the above nine functions are listed below. First I will give the computations of the functions on *Statements*, in a pattern of grammar rule, attribute grammar instantiation, and computation. Then I will give the computations of the functions on *Expressions*.

$$\begin{array}{l} \textit{Statement} \rightarrow \left(\textbf{accept } (\textit{MethodImp})^{+\ |} \ [\textbf{accept}]^{?}\right) \\ \textit{MethodImp} \rightarrow \textit{Name}\ ((\textit{Argument})^{*\ ,})\ \textbf{when } \textit{Expression} \\ \qquad \textit{Statement } \textbf{then } \textit{Statement} \end{array}$$

$$S = (\textbf{accept } S.\textit{acceptbodies } \textbf{accept})$$

$$b \in S.\textit{acceptbodies}$$

$$b = b.\textit{name}\,(b.\textit{arguments})\ \textbf{when } b.\textit{guard}\ b.\textit{body}\ \textbf{then } b.\textit{afterbody}$$

$$\textit{syn}\,(S) = \mathsf{true}$$

$$\textit{useLoc}\,(S) = \bigcup \{\textit{expUseLoc}\,(b.\textit{guard})\ |b \in S.\textit{acceptbodies}\}$$

$$\textit{useVar}\,(S) = \bigcup \{\textit{expUseVar}\,(b.\textit{guard}) \cup \textit{expUseVar}\,(b.\textit{body}\ b.\textit{afterbody})\ |b \in S.\textit{acceptbodies}\}$$

$$\cup \left(\bigcup \{\textit{defVar}\,(b.\textit{body}\ b.\textit{afterbody})\} - \bigcap \{\textit{defVar}\,(b.\textit{body}\ b.\textit{afterbody})\}\right)$$

$$\textit{defLoc}\,(S) = \bigcup \{\textit{defLoc}\,(b.\textit{afterbody})\ |b \in S.\textit{acceptbodies}\}$$

$$\textit{defVar}\,(S) = \bigcup \{\textit{defVar}\,(b.\textit{body}) \cup \textit{defVar}\,(b.\textit{afterbody})\ |b \in S.\textit{acceptbodies}\}$$

The *useVar* set contains not only the *Variables* that are used in the implementation bodies but also those that are defined in some bodies (and not defined in the others) because if a defined *Variable* is not defined in some other bodies, the merging (ϕ-function) of it is a use of it in those other bodies. The inferred lock operations lead to synchronization after $b.\textit{body}$, so the *defLoc* set only contains the *Locations* defined in $b.\textit{afterbody}$.

$$\textit{Statement} \rightarrow \textit{ObjectId} := \textit{Expression}$$

$$S = S.\textit{left} := S.\textit{right}$$

$$\textit{syn}\,(S) = \mathsf{false}$$

$$\textit{useLoc}\,(S) = \textit{indexUseLoc}\,(S.\textit{left}) \cup \textit{expUseLoc}\,(S.\textit{right})$$

$$\textit{useVar}\,(S) = \textit{indexUseVar}\,(S.\textit{left}) \cup \textit{expUseVar}\,(S.\textit{right})$$

$$\textit{defLoc}\,(S) = \begin{cases} \textit{location}\,(S.\textit{left}) & \text{if } S.\textit{left} \text{ represents a known } \textit{Location} \\ \emptyset & \text{otherwise} \end{cases}$$

$$defVar\,(S) = \begin{cases} variable\,(S.left) & \text{if } S.left \text{ represents a } Variable \\ \emptyset & \text{otherwise} \end{cases}$$

The definition of known *Location* (distinguished with *unknown Location*) will be given later.

Statement $\rightarrow$ *Block*
Block $\rightarrow$ $(Statement)^+$

$$S = b_0 \ldots b_{size-1}$$

Let f be an integer so that b_f is the first synchronized *Statement* in block, or f equals to *size* in case that block is not synchronized. In other words, f satisfies that $(syn(b_f) \wedge \forall i \in \{0, \ldots f\}.\neg syn(b_i)) \vee (f = size \wedge \forall i \in \{0, \ldots size\}.\neg syn(b_i))$

Let l be an integer so that b_l is the last synchronized *Statement* in block, or l equals to 0 in case that block is not synchronized. In other words, l satisfies that $(l = 0 \wedge \forall i \in \{0, \ldots size\}.\neg syn\,(b_i)) \vee (syn(b_l) \wedge \forall i \in \{l+1, \ldots size\}.\neg syn(b_i))$

$$syn\,(S) = \bigvee \{syn\,(b_i)\,|i \in \{0, \ldots size\}\}$$

$$useLoc\,(S) = useLoc\,(b_0) \cup$$
$$\left(\bigcup \left\{ \left(useLoc\,(b_i) - \bigcup \{defLoc\,(b_j)\,|j \in \{0, \ldots i\}\} \right) |i \in \{1, \ldots f\} \right\} \right)$$

$$useVar\,(S) = useVar\,(b_0) \cup$$
$$\left(\bigcup \left\{ \left(useVar\,(b_i) - \bigcup \{defVar\,(b_j)\,|j \in \{0, \ldots i\}\} \right) |i \in \{1, \ldots size\} \right\} \right)$$

$$defLoc\,(S) = \bigcup \{defLoc\,(b_i)\,|i \in \{l, \ldots size\}\}$$

$$defVar\,(S) = \bigcup \{defVar\,(b_i)\,|i \in \{0, \ldots size\}\}$$

The *useLoc* set is the union of the used *Locations* in b_0, the *Locations* that are used in b_1 but not defined in b_0, the *Locations* that are used in b_2 but not defined in b_0b_1, and so on until the first synchronization. The *useVar* set is similarly computed.

$$Statement \rightarrow \textbf{call}\ \lceil ObjectId.Name | Name \rceil\ (Arguments)$$
$$Arguments \rightarrow (Expression)^{*}$$

$$\begin{aligned}
S &= \textbf{call}\ S.name\ (S.parameters) \\
S.parameters &: Direction \times Expression \\
p &\in S.parameters \\
p &= (p.D, p.E) \\
syn\,(S) &= \text{true} \\
useLoc\,(S) &= \emptyset \\
useVar\,(S) &= expUseVar\,(S.name) \cup \\
&\quad \{q | q \in S.parameters \wedge q.E \in Variables \wedge q.D = \text{"in"}\} \\
defLoc\,(S) &= \{q | q \in S.parameters \wedge q.E \in Locations \wedge q.D = \text{"out"}\} \\
defVar\,(S) &= \{q | q \in S.parameters \wedge q.E \in Variables \wedge q.D = \text{"out"}\}
\end{aligned}$$

The direction information of the parameters comes from the *MethodType* of the method in the object graph (see description in page 12).

$$Statement \rightarrow \left(\textbf{co}\ (Statement)^{+\ \|}\ \lceil \textbf{co} \rceil^{?}\right)$$

$$\begin{aligned}
S &= (\textbf{co}\ b_0 \| \ldots \| b_{size-1}\ \textbf{co}) \\
syn\,(S) &= \text{true} \\
useLoc\,(S) &= \emptyset \\
useVar\,(S) &= \bigcup \{useVar\,(b_i)\, | i \in \{0, \ldots size\}\} \\
defLoc\,(S) &= \emptyset \\
defVar\,(S) &= \bigcup \{defVar\,(b_i)\, | i \in \{0, \ldots size\}\}
\end{aligned}$$

$$Statement \rightarrow \left(\textbf{if}\ Expression\ \textbf{then}\ Statement\ \textbf{else}\ Statement\ \lceil \textbf{if} \rceil^{?}\right)$$

$$\begin{aligned}
S &= (\textbf{if}\ S.guard\ \textbf{then}\ S.then\ \textbf{else}\ S.else\ \textbf{if}) \\
syn\,(S) &= syn\,(S.then) \vee syn\,(S.else)
\end{aligned}$$

$$
useLoc\,(S) = \begin{cases} expUseLoc\,(S.guard) \cup \\ useLoc\,(S.then) \cup useLoc\,(S.else) & \text{if } syn\,(S) \\[2ex] expUseLoc\,(S.guard) \cup \\ useLoc\,(S.then) \cup useLoc\,(S.else) \cup \\ ((defLoc\,(S.then) \cup defLoc\,(S.else)) - & \text{if } \neg syn(S) \\ (defLoc\,(S.then) \cap defLoc\,(S.else))) \end{cases}
$$

$$
\begin{aligned}
useVar\,(S) &= expUseVar\,(S.guard) \cup useVar\,(S.then) \cup useVar\,(S.else) \cup \\
&\quad (defVar\,(S.then) \cup defVar\,(S.else)) - (defVar\,(S.then) \cap defVar\,(S.else)) \\
defLoc\,(S) &= defLoc\,(S.then) \cup defLoc\,(S.else) \\
defVar\,(S) &= defVar\,(S.then) \cup defVar\,(S.else)
\end{aligned}
$$

If an *IF Statement* is not synchronized, the *useLoc* set contains not only the *Locations* that are used, but also those that are defined in exactly one of the branches because if a *Location* is defined in one and only one branch, the merging (ϕ-function) of it is a use of it. The *useVar* set contains not only the *Variables* that are used, but also those are defined in exactly one of the branches for the same reason.

$$
\textit{Statement} \rightarrow \left(\textbf{wh}\ \textit{Expression}\ \textbf{do}\ \textit{Statement}\ \lceil\textbf{wh}\rceil^{?}\right)
$$

$$
\begin{aligned}
S &= (\textbf{wh}\ S.guard\ \textbf{do}\ S.body\ \textbf{wh}) \\
syn\,(S) &= syn\,(S.body) \\
useLoc\,(S) &= expUseLoc\,(S.guard) \cup useLoc\,(S.body) \cup defLoc\,(S.body) \\
useVar\,(S) &= expUseVar\,(S.guard) \cup useVar\,(S.body) \cup defVar\,(S.body) \\
defLoc\,(S) &= defLoc\,(S.body) \\
defVar\,(S) &= defVar\,(S.body)
\end{aligned}
$$

The *useLoc* set contains not only the *Locations* that are used, but also those that are defined in the loop body because if a *Location* is defined in the loop body, the merging (ϕ-function) of it is a use of it. The computation of the *useVar* set is similar to the *useLoc* set.

$$\mathit{Statement} \rightarrow \left(\textbf{with}\ \mathit{ObjectId}\ \textbf{when}\ \mathit{Expression}\ \textbf{do}\ \mathit{Statement}\ [\textbf{with}]^{?}\right)$$

$$S = (\textbf{with}\ S.\mathit{lock}\ \textbf{when}\ S.\mathit{guard}\ \textbf{do}\ S.\mathit{body}\ \textbf{with})$$

$$\mathit{syn}\,(S) = \mathsf{true}$$

$$\mathit{useLoc}\,(S) = \emptyset$$

$$\mathit{useVar}\,(S) = \mathit{expUseVar}\,(S.\mathit{lock}) \cup \mathit{expUseVar}\,(S.\mathit{guard}) \cup \mathit{useVar}\,(S.\mathit{body})$$

$$\mathit{defLoc}\,(S) = \emptyset$$

$$\mathit{defVar}\,(S) = \mathit{defVar}\,(S.\mathit{body})$$

The computation of the functions on *Expressions* is shown in the following tables.

Type of *Expression* E	$\mathit{expUseLoc}\,(E)$	$\mathit{indexUseLoc}\,(E)$
NULL	$\emptyset$	$\emptyset$
IDENTIFIER	$\{\mathit{location}\,(E)\}$	$\emptyset$
REFERENCE	$\{\mathit{location}\,(E)\}$ $\cup\ \mathit{indexUseLoc}\,(E)$	$\mathit{indexUseLoc}\,(E.\mathit{left})$
INDEX	$\{\mathit{location}\,(E)\}$ $\cup\ \mathit{indexUseLoc}\,(E)$	$\mathit{expUseLoc}\,(E.\mathit{right})$ $\cup\ \mathit{indexUseLoc}\,(E.\mathit{left})$
other	$\mathit{expUseLoc}\,(E.\mathit{left})$ $\cup\ \mathit{expUseLoc}\,(E.\mathit{right})$	$\mathit{indexUseLoc}\,(E.\mathit{left})$ $\cup\ \mathit{indexUseLoc}\,(E.\mathit{right})$

Type of *Expression* E	$\mathit{expUseVar}\,(E)$	$\mathit{indexUseVar}\,(E)$
NULL	$\emptyset$	$\emptyset$
IDENTIFIER	$\{\mathit{variable}\,(E)\}$	$\emptyset$
REFERENCE	$\mathit{indexUseVar}\,(E)$	$\mathit{indexUseVar}\,(E.\mathit{left})$
INDEX	$\mathit{indexUseVar}\,(E)$	$\mathit{expUseVar}\,(E.\mathit{right})$ $\cup\ \mathit{indexUseVar}\,(E.\mathit{left})$
other	$\mathit{expUseVar}\,(E.\mathit{left})$ $\cup\ \mathit{expUseVar}\,(E.\mathit{right})$	$\mathit{indexUseVar}\,(E.\mathit{left})$ $\cup\ \mathit{indexUseVar}\,(E.\mathit{right})$

3.4.2 The Second Pass

The second pass analysis is to generate dataflow graph for each *Statement*. The *interpret* procedure of each *Statement* has three inputs, a control flow root node, an alive *Variable* set, and an alive *Location* set, and has four outputs, a control flow root node, an alive *Variable* set, an alive *Location* set, and a sub dataflow graph. I denote the inputs as *rootIn*, *liveInVar*, and *liveInLoc*, and the outputs as *rootOut*, *liveOutVar*, *liveOutLoc*, respectively.

$$\begin{aligned} \mathit{liveInVar} &: \mathit{Statement} \to (\mathit{Variable} \to \mathit{Boolean}) \\ \mathit{liveInLoc} &: \mathit{Statement} \to (\mathit{Location} \to \mathit{Boolean}) \\ \mathit{liveOutVar} &: \mathit{Statement} \to (\mathit{Variable} \to \mathit{Boolean}) \\ \mathit{liveOutLoc} &: \mathit{Statement} \to (\mathit{Location} \to \mathit{Boolean}) \end{aligned}$$

If the *Statement* has nested *Statements*, the analysis also gives the three inputs of the analysis of those nested *Statements*: the rootIn node will be shown in the dataflow graph as an input edge of the nested *interpret* rectangle; the computations of the *liveInVar* set and the *liveInLoc* set will be given in a formula list.

***interpret* Procedure** The *interpret* procedure the four 2nd-pass sets and the dataflow graph generation. The detail of the rest part of the *interpret* procedure is shown below.

$$\begin{aligned} &\mathit{Statement} \to \left(\textbf{accept}\ (\mathit{MethodImp})^{+\,|}\,\lceil\textbf{accept}\rceil^{?}\right) \\ &\mathit{MethodImp} \to \mathit{Name}\,((\mathit{Argument})^{*}\,{}^{,})\ \textbf{when}\mathit{Expression} \\ &\qquad \mathit{Statement}\ \textbf{then}\ \mathit{Statement} \end{aligned}$$

Figure 3.2 (page 35) shows the dataflow graph of the *Statement* S which is instantiated as an *ACCEPT Statement*. The * symbols represents multiple similar edges, and overlapped contents (such as the *evaluate* procedures) represents multiple similar sub-graphs. The dashed arrows represent the data dependency or control dependency which indicates the order of construction of either control flows or data flows. The function method(...) is to get the method from the object graph. For instance, method(b.name).lock__a represents the inferred lock "a" of the method with the name of b.name in the object whose thread is being analyzed. All the *STORE* nodes and *SINK* nodes have the side effect of killing all definitions.

$$S = (\textbf{accept}\ S.acceptbodies\ \textbf{accept})$$

$$b \in S.acceptbodies$$

$$b = b.name\,(b.arguments)\ \textbf{when}\ b.guard\ b.body\ \textbf{then}\ b.afterbody$$

$$b.arguments : Direction \times Name$$

$$a \in b.arguments$$

$$a = (a.d, a.name)$$

$$liveInLoc\,(b.body) = \emptyset$$

$$liveInVar\,(b.body) = liveInVar\,(S) \cup useVar\,(S) \cup$$
$$\{variable\,(a.name)\,|a \in b.arguments \wedge a.d = \text{"in"}\}$$

$$liveInLoc\,(b.afterbody) = \emptyset$$

$$liveInVar\,(b.afterbody) = liveOutVar\,(b.body)$$

$$liveOutLoc\,(S) = \cup\,\{defLoc\,(b.afterbody)\,|b \in S.acceptbodies\}$$

$$liveOutVar\,(S) = liveInVar\,(S) \cup \bigcup\{(defVar\,(b.body\ b.afterbody)) -$$
$$\{variable\,(a.name)\,|a \in b.arguments\}\,|b \in S.acceptbodies\}$$

Statement → *ObjectId* := *Expression*

$$S = left := right$$
$$liveOutLoc\,(S) = liveInLoc\,(S) \cup defLoc\,(S)$$
$$liveOutVar\,(S) = liveInVar\,(S) \cup defVar\,(S)$$

The data flow graph of *ASSIGNMENT Statement* is shown in Figure 3.3.

Statement → *Block*
Block → $(Statement)^{+}$

$$S = S.statements$$
$$S.Statements = b_0 \ldots b_{size-1}$$
$$liveInLoc\,(b_0) = liveInLoc\,(S)$$
$$liveInVar\,(b_0) = liveInVar\,(S)$$
$$liveInLoc\,(b_i) = liveOutLoc\,(b_{i-1})$$
$$liveInVar\,(b_i) = liveOutVar\,(b_{i-1})$$
$$liveOutLoc\,(S) = liveOutLoc\,(b_{s-1})$$
$$liveOutVar\,(S) = liveOutVar\,(b_{s-1})$$

The data flow graph of *BLOCK Statement* is shown in Figure 3.4.

Statement → **call** $\lceil ObjectId.Name | Name \rceil$ $(Arguments)$
Arguments → $(Expression)^{*}$

$$S = \textbf{call}\ S.name\,(S.parameters)$$
$$liveOutLoc\,(S) = defLoc\,(S)$$
$$liveOutVar\,(S) = liveInVar\,(S) \cup defVar\,(S)$$

The data flow graph of *CALL Statement* is shown in Figure 3.5.

$$Statement \rightarrow \left(\textbf{co}\ (Statement)^{+}\ \|\ \lceil\textbf{co}\rceil^{?}\right)$$

$$\begin{aligned}
S &= (\textbf{co}\ S.statements\ \textbf{co}) \\
b &\in S.statements \\
liveInLoc\,(b) &= \emptyset \\
liveInVar\,(b) &= liveInVar\,(S) \\
liveOutLoc\,(S) &= \emptyset \\
liveOutVar\,(S) &= liveInVar\,(S) \cup defVar\,(S)
\end{aligned}$$

The data flow graph of *CO Statement* is shown in Figure 3.6.

$$Statement \rightarrow \left(\textbf{if}\ Expression\ \textbf{then}\ Statement\ \textbf{else}\ Statement\ \lceil\textbf{if}\rceil^{?}\right)$$

$$\begin{aligned}
S &= (\textbf{if}\ S.guard\ \textbf{then}\ S.then\ \textbf{else}\ S.else\ \textbf{if}) \\
liveInLoc\,(S.then) &= liveInLoc\,(S) \cup expUseLoc\,(S.guard) \\
liveInVar\,(S.then) &= liveInVar\,(S) \\
liveInLoc\,(S.else) &= liveInLoc\,(S) \cup expUseLoc\,(S.guard) \\
liveInVar\,(S.else) &= liveInVar\,(S) \\
liveOutLoc\,(S) &= liveOutLoc\,(S.then) \cup liveOutLoc\,(S.else) \\
liveOutVar\,(S) &= liveInVar\,(S) \cup defVar\,(S)
\end{aligned}$$

The data flow graph of *IF Statement* is shown in Figure 3.7.

$$Statement \rightarrow \left(\textbf{wh}\ Expression\ \textbf{do}\ Statement\ \lceil\textbf{wh}\rceil^{?}\right)$$

$$\begin{aligned}
S &= (\textbf{wh}\ S.guard\ \textbf{do}\ S.body\ \textbf{wh}) \\
liveInLoc\,(S.body) &= liveInLoc\,(S) \cup expUseLoc\,(S.guard) \cup defLoc\,(S) \\
liveInVar\,(S.body) &= liveInVar\,(S) \\
liveOutLoc\,(S) &= liveInLoc\,(S) \cup liveOutLoc\,(S.body) \\
liveOutVar\,(S) &= liveInVar\,(S) \cup defVar\,(S)
\end{aligned}$$

The data flow graph of *WHILE Statement* is shown in Figure 3.8.

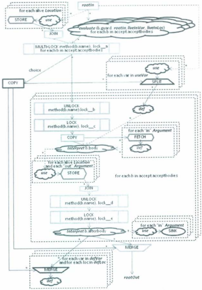

Figure 3.2: Data Flow Graph of *ACCEPT Statement*

Figure 3.3: Data Flow Graph of *ASSIGN Statement*

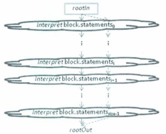

Figure 3.4: Data Flow Graph of *BLOCK Statement*

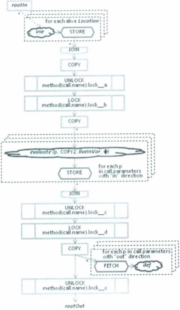

Figure 3.5: Data Flow Graph of *CALL Statement*

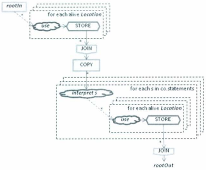

Figure 3.6: Data Flow Graph of *CO Statement*

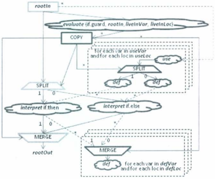

Figure 3.7: Data Flow Graph of *IF Statement*

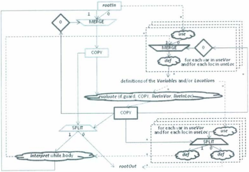

Figure 3.8: Data Flow Graph of *WHILE Statement*

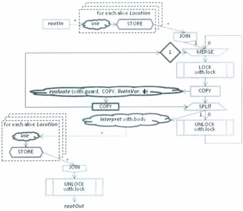

Figure 3.9: Data Flow Graph of *WITH Statement*

$$\textit{Statement} \rightarrow \left(\textbf{with } \textit{ObjectId} \textbf{ when } \textit{Expression} \textbf{ do } \textit{Statement} \; [\textbf{with}]^{?}\right)$$

$$S = (\textbf{with } \textit{S.lock} \textbf{ when } \textit{S.guard} \textbf{ do } \textit{S.body} \textbf{ with})$$
$$\textit{liveInLoc}\,(\textit{S.body}) = \textit{expUseLoc}\,(\textit{S.guard})$$
$$\textit{liveInVar}\,(\textit{S.body}) = \textit{liveInVar}\,(S)$$
$$\textit{liveOutLoc}\,(S) = \emptyset$$
$$\textit{liveOutVar}\,(S) = \textit{liveInVar}\,(S) \cup \textit{defVar}\,(S)$$

The data flow graph of *WITH Statement* is shown in Figure 3.9.

***evaluate*, *use*, and *def* Procedures** During the analysis, the definitions of all the *Variables* and *Locations* are tracked by two functions, *defNodeVar* and *defNodeLoc*. The definition of a *Variable/Location* can be found by one of these functions (using *use* procedure which returns a dataflow graph node), and the definition can be added, removed, or updated by modifying the mapping of these functions (using *def* procedure).

$$\textit{defNodeVar} : \textit{Variable} \rightarrow \textit{DataFlowGraphNode}$$

$$\textit{defNodeLoc} : \textit{Location} \cup \textit{Expression} \rightarrow (\textit{Boolean} \times \textit{DataFlowGraphNode})$$

The *Boolean* value in the *defNodeLoc* indicates whether the *Location* is defined. It will be false if that *Location* is only fetched and used. The *Expression* sub-domain of *defNodeLoc* function is used if and only if the *Expression* represents an *unknown Location* (defined later).

Compared with the data flows of the *Variables*, the data flows of the *Locations* are more complicated because (1) when a *Location* is used, it is allowed to have no reaching definition, and then it will be *FETCH*ed, and (2) a synchronization will

Before an *unknown Location* is either used or defined, all the alive *Locations* that are potentially the same as it have to be *stored* and removed from the domain of the function *defNodeLoc*; and when an *unknown Location* is either *fetched* or *stored*, the evaluations of the index components of that *INDEX Expression* (sometimes an *Expression* has more than one index components) are provided to the *Fetch* node or the *Store* node, and the addressing can be accomplished at run time.

```
procedure use(Expression exp) returns DataFlowGraphNode
    if (exp represents a Variable var)
        return defNodeVar(var)
    else if (exp represents a known Location loc)
        Store or Sink each alive potential-same unknown Location
        remove the mapping of each Stored Location from defNodeLoc
        Join those Stores, and Copy the Join
        update the root to the Copy of the Join
        if (loc is in the domain of defNodeLoc)
            res is a DataFlowGraphNode so that
                (loc ⟼ (true/false, res)) ∈ defNodeLoc
            return res
        else
            res := new Copy(new Fetch(loc))
            add (loc ⟼ (false, res)) to defNodeLoc
            return res
        end if
    else
        Store or Sink each alive potential-same Location
        remove the mapping of each Stored Location from defNodeLoc
        Join those Stores, and Copy the Join
        update the root to the Copy of the Join
        Fetch the exp, and Copy the Fetch
        add (exp ⟼ the Copy of the Fetch) to defNodeLoc
        return the Copy of the Fetch
```

```
    end if
end

procedure def(Expression exp, DataFlowGraphNode eDef)
    if (exp represents a Variable var)
        modify the mapping of var in defNodeVar into eDef
    else if (exp represents a known Location loc)
        Store or Sink each alive potential-same Location
        remove the mapping of each Stored Location from defNodeLoc
        Join those Stores, and Copy the Join
        update the root to the Copy of the Join
        if (loc is in the domain of defNodeLoc)
            if (defNodeLoc(loc) is not used) Sink defNodeLoc(loc) end if
            modify the mapping of loc in defNodeLoc into eDef
        else
            add (loc ⟼ (true, eDef)) to defNodeLoc
        construct a Store of loc for further use
    else
        Store or Sink each alive potential-same Location
        remove the mapping of each Stored Location from defNodeLoc
        Join those Stores, and Copy the Join
        update the root to the Copy of the Join
        add (exp ⟼ (true, eDef)) to defNodeLoc
        construct a Store of exp for further use
            (also evaluate all the index components in exp)
    end if
end
```

3.5 Low-Level Optimization

To perform a low-level optimization which gets rid of some unnecessary nodes, I defined three primitive procedures, *eliminate*, *replace*, and *disconnect*. The *eliminate* procedure removes a certain descendant of a certain node, and connect all the descendants of the removed descendant to that node as new descendants. The *replace* procedure replaces a certain descendant of a certain node by the descendant of that descendant when the descendant has only one descendant. The *disconnect* procedure removes a certain node from all its ascendents' descendant lists.

Based on these primitive procedures, the *removeRedundancy* procedure is defined: (1) if a *MULTI-LOCK* or *SPLIT* (these two types are descendant-index-sensitive) root node has a non-*SINK* descendant node with no descendant, *disconnect* the descendant and replace it with a *SINK* node in the root's descendant list; and (2) if a root node has a non-*SINK* descendant node with no descendant and the root is not *MULTI-LOCK* or *SPLIT*, *disconnect* the descendant and remove it from the root's decendant list.

The optimization procedure contains four steps, each of which is a bottom-up traverse of the dataflow graph. The first step deals with three situations: (1) if a *COPY* node has a *COPY* ascendent, *eliminate* that *COPY* node; (2) if a *COPY* node has only one descendant, *eliminate* that *COPY* node; and (3) if a *JOIN* node has only one ascendant, *eliminate* the *JOIN* node. The second step is to apply the *removeRedundancy* procedure. The third step is to replace *COPY* or *SPLIT* nodes with only *SINK* descendants by *SINK* nodes. The fourth step is to apply the *removeRedundancy* procedure again.

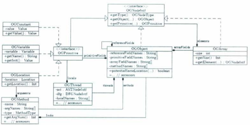

Figure 3.10: Class Diagram of Object Graph Package

3.6 Implementation Details

The data flow synthesis algorithm proposed above consists of two parts: the 1st pass is implemented as an extension of the AST module in the front-end; and the 2nd pass is implemented as an individual module. I implement them in Java using the interfaces and the data structures described in this section.

3.6.1 Interfaces and Extensions of the Front-End

The interfaces of object graph module are shown in Fig 3.10. The method *getType*() returns the object graph node type of a node. The enum *OGNodeType* used in the data flow synthesis includes *LOCATION*, *OBJECT*, *ARRAY*, *VARIABLE*, and *CONSTANT*, represented by sub-interfaces *OGLocation*, *OGObject*, *OGArray*, *OGVariable*, and *OGConstant*, respectively. The method *potentialSame*() in *OGObject* class judges whether a *Location* is possibly referred by an *Expression* referring to an unknwon *Location*, or whether an *Expression* referring to an unknown *Loca-*

tion is possibly referring to a same *Location* with the other *Expression*. The *getPrimitive()/getObject()* method in *OGNodeIntf* interface is to find the *OGPrimitive/OGObject* node represented by a given expression whose path starts from the current node. The class *MethodType* is provided by the front-end as a part of the type system. The instances of this class provide information of a method such as its parameters, name, etc.

ASTNodeIntf is the inteface of the AST node which is described in Fig 3.11.

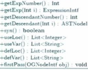

Figure 3.11: Class Diagram of interface ASTNodeIntf

My data flow synthesis deals with normalized ASTs. I will first describe the normalization, and then I will introduce each method in the interface.

ASTNodeType	normalization
ACCEPT_BODY	Each *ACCEPT_BODY* node has following descendants: methodType, guard, body, and afterBody.
COLOOP	Transformed into a *CO* node.
FOR	Transformed into a *WHILE* node.
IF	Each *IF* node has following descendants: guard, thenClause, and elseClause.
WITH	Each *WITH* node has following descendants: guard, lock, and body.

The method *getType*() returns the AST node type of the current node. The **enum** *ASTNodeType* used in the data flow synthesis includes *ASSIGNMENT*, *ACCEPT*, *ACCEPT_BODY*, *CALL*, *IF*, *WHILE*, *BLOCK*, *THREAD*, *CO*, and *WITH*. The method *getExpNumber*() returns the number of *Expression* descendants the node has, which is listed in the following table, and the method *getExp*() returns a descendant *Expression* node.

ASTNodeType	# exp	description
ASSIGNMENT	2	one for the left-hand-side, and the other for the right-hand-side
ACCEPT	0	-
ACCEPT_BODY	n+1	one for the guard, and n for the parameters
CALL	n+1	one for the name, and n for the parameters
IF	1	for the guard
WHILE	1	for the guard
BLOCK	0	-
THREAD	0	-
CO	0	-
WITH	2	one for the lock, and the other for the guard

The method *getNumber*() returns the number of AST node descendants the node has, which is listed in the following table, and the method *getDescendant*() returns a descendant AST node.

ASTNodeType	# des	description
ASSIGNMENT	0	-
ACCEPT	n	for n accept bodies
ACCEPT_BODY	2	one for the body, and the other for the after-Body
CALL	0	-
IF	2	one for the then clause, and the other for the else clause
WHILE	1	for the body
BLOCK	n	for n statements in the block
THREAD	1	for the body
CO	n	for n bodies
WITH	1	for the body

The methods *syn*(), *useLoc*(), *defLoc*(), and *defVar*() return the results of the 1st pass computation. The method *firstPass*() is to process the 1st pass computation.

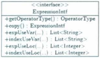

Figure 3.12: Class Diagram of interface ExpressionIntf

ExpressionIntf is the interface of *Expression* class (Fig 3.12). The method *getOperatorType*() returns the operator type of the current *Expression*. The **enum** *OperatorType* used in the data flow synthesis includes *LITERAL* for the constants, *IDENTIFIER*, *REFERENCE*, *INDEX* for the array elements, *MATH* for the mathematical operations with two operands, *NEG* for the negative operation, and *EQUAL*-

ITY and *COMPARISON* for the comparisons. The method *copy*() returns a copy of the current *Expression*. The methods *expUseLoc*() and *indexUseLoc*() are parts of the 1st pass computation. The interface also provides some other methods to the *Expressions* with particular operator types.

operator types	method
IDENTIFIER, REFERENCE	getToken() : Token
LITERAL	getValue() : Value
LITERAL	getConstant() : **boolean**
REFERENCE, INDEX, MATH, EQUALITY, COMPARATION	getLeft() : ExpressionIntf
INDEX, MATH, NEG, EQUALITY, COMPARATION	getRight() : ExpressionIntf

3.6.2 Data Structure of the Data Flow Graph

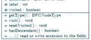

Figure 3.13: Class Diagram of DFGNode

The data flow graph is constructed as a set of nodes that each node knows both its descendants and its ascendents. The abstract class of the nodes is shown in Fig 3.13. The method *resetAscendent*() is to break all those edges with the given node as the source and the current node as the target. The method *resetDescendant*() is to break all those edges with the current node as the source and the target node as the target. The **enum** *DFGNodeType* includes *START, SINK, FUNC, COPY, JOIN, MERGE, SPLIT, MULTILOCK, LOCK, UNLOCK, FETCH, STORE,* and *VALUE.* The methods *visit*() and *resetVisited*() help the traversal of the graph. The method *hasDescendant*() returns true if the node has no descendant.

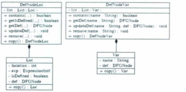

Figure 3.14: Implementations of *defNodeLoc* and *defNodeVar*

The design of implementing the functions *defNodeLoc* and *defNodeVar* (see subsection The Second Pass) is shown in Fig 3.14. The method *contains*() of either class indicates if the domain of the function contains the given *Location*/*Variable* (a *Location* can be represented by an integer value or an *Expression*). The method *getIsDefined*() in *DefNodeLoc* class returns true if a *Location* is defined, and false if the *Location* is *FETCH*ed for uses. The method *getDef*() returns the data flow graph node which is the living definition of the given *Location*/*Variable*. The method *updateDef*() edits the mapping of the function. The method *remove*() removes all the mapping from the given *Location*/*Variable*. The method *copy*() returns a copy of the function.

Each *Loc* object has four fields: *location* or *exp* is the argument of a mapping, and *isDefined* and *def* constitute the image of the mapping. Each *Var* object has two fields: *name* is the argument of a mapping, and *def* is the image of the mapping. The method *copy*() returns a copy of the object.

3.7 Example

The first example[2] is the data flow graph of a *FOR Statement* (which has been normalized into a *WHILE Statement*) with a nested *IF Statement*. The HARPO/L program is

```
(class Example1
    constructor ()
    private obj a := 3
    private obj b := 0
    private obj c := 1
    (thread
        obj i := 0
        (wh i<10
            (if a%2=0 then
                b := a-c
            else
                b := a+c
            if)
        a := b
        i := i+1
        wh)
    thread)
class)
obj obj1 := new Example1()
```

The data flow graph of the thread in object "obj1" is shown in Fig 3.15. The bi-connected SPLIT and *MERGE* (on the right-hand-side) are control flows of the

[2]Suggested by Rani Gnanaolivu.

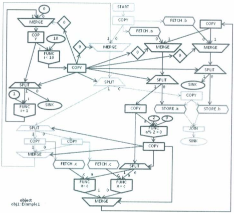

Figure 3.15: Data Flow Graph of Object "obj1"

IF Statement. They are equivalent to a node that keeps waiting for data from the *COPY* of the *FUNC* $a\%2 = 0$, and whatever it receives, it will pass the control flow along its output edge (to the *MERGE* on the top). This simplification is left to the further optimization.

The second example shows the data flow synthesis of the thread of the "producer" object implementing a FIFO buffer[3]. The buffer is described by the class FIFO with two public procedures which are implemented in one *ACCEPT Statement*, which means that they can not simultaneously execute. Once a client calls either of these procedures, the client will wait until next time the *ACCEPT Statement* is executed. Then, the guards are evaluated to judge whether the call is acceptable, and if so, the called procedure's implementation body is executed. The guards and the assignments of fields *producer.size* and *producer.front* ensures that the FIFO will never be overflowed or overdrained.

```
(class FIFO {type T extends primitive}
    constructor (in capacity: int8)
    public proc deposit (in value: T)
    public proc fetch (out value: T)
    private obj buf:T(capacity) := (for i:capacity do 0)
    private obj front := 0
    private obj size := 0
    (thread
        (wh true
            (accept
                deposit (in value: T) when size<capacity
                buf[(front+size)%capacity] := value
                size := size+1
            |
                fetch (out value:T) when size>0
```

[3]Suggested by Dr. Theodore S. Norvell.

```
                value := buf[front]
                front := (front+1)%capacity
                size := size-1
            accept)
        wh)
    thread)
class)
obj producer := new FIFO{int32}(40)
```

The constructor field *capacity* has primitive type, so it is considered a constant. According to the semantics of HARPO/L[12], each method has five inferred locks, $\mathfrak{a}$, $\mathfrak{b}$, $\mathfrak{c}$, $\mathfrak{d}$, and $\mathfrak{e}$. All the $\mathfrak{a}$ locks are controlled by a *MULTI-LOCK* node: every time the *ACCEPT Statement* is executed, only one of the $\mathfrak{a}$ locks are locked (only one client call is processed) and only one of the *MULTI-LOCK*'s outputing control flows is activated. As shown in this data flow graph in Fig 3.16, guards of the *ACCEPT*, $size < capacity$, and $size > 0$, are evaluated in parallel, and fed into the *MULTI-LOCK* node. The *MULTI-LOCK* node split the control flow into the implementation bodies of the two methods, *deposit* and *fetch*. In *deposit*, after *FETCH*ing the "in" argument value and evaluating the value of an array index $(front + size)\,\%capacity$, the assignments of $a\,[(front + size)\,\%capacity]$ and *size* are processed in parallel. In fetch, the assignments of *front* and *size* are processed in parallel, and the "out" argument value is *STORE*d before fetch.$\mathfrak{d}$ is unlocked.

In this data flow graph, since the guard of the while loop is always true, a number of nodes, such as the *SINK* node and the *SPLIT* nodes of *capacity* and *size*, are unnecessary. This redundancy and some other issues are left to further optimization.

Because the FIFO class has only one thread, we can also declare *front* and *size* as variables rather than fields. The data flow graph of the following program is shown in Fig 3.17.

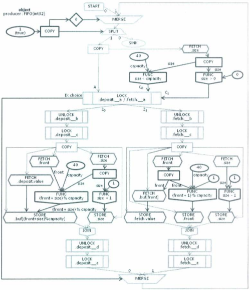

Figure 3.16: Data Flow Graph of the Thread of the "producer" Object

```
(class FIFO {type T extends primitive}
    constructor (in capacity: int8)
    public proc deposit (in value: T)
    public proc fetch (out value: T)
    private obj buf:T(capacity) := (for i:capacity do 0)
    (thread
        obj front := 0
        obj size := 0
        (wh true
            (accept
                deposit (in value: T) when size<capacity
                buf[(front+size)%capacity] := value
                size := size+1
            |
                fetch (out value:T) when size>0
                value := buf[front]
                front := (front+1)%capacity
                size := size-1
            accept)
        wh)
    thread)
class)
obj producer := new FIFO{int32}(40)
```

In the data flow graph, front and size are no longer *FETCH*ed. Instead, they are *MERGE*d and *SPLIT* with the control flow of the *WHILE Statement*, and *SPLIT* and *MERGE*d with the control flow of the *ACCEPT Statement*.

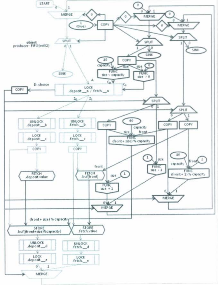

Figure 3.17: Data Flow Graph of the Thread of the "producer" Object version 2

Chapter 4

Verification of Parallel Object-Oriented Programs

In this chapter, I show how to automate verification of parallel object-oriented programs. The intention of this research is to build a verifying compiler[19] for HARPO/L. Although this task is not accomplished, I make positive progress on filling the gap between automatic verification of sequential programs and that of parallel programs.

4.1 Background

The verification of programs judges whether a program satisfies a specification. Usually, a specification contains two predicate formulas: a precondition, which is required to hold before the execution of the program; and a postcondition, which is required to hold after the execution of the program. Some other formulas, such as loop invariants and type invariants, may be also given to constrain the program.

Verification has been formalized axiomatically since Hoare triples were defined in 1969[20]. A Hoare triple $\{P\}\, S\, \{Q\}$ contains a pair of Boolean expressions (precondition P and postcondition Q) and a program S. If P being **true** before the execution of S guarantees Q to be **true** when the execution terminates, then we say this triple is valid, denoted $\vdash \{P\}\, S\, \{Q\}$, which represents that the command S has partial behavioural correctness. With the verification rules for primitive commands (assignments) and the verification rules of control flows (sequential compositions, branches, and loops), the verification of the entire program can be achieved.

The axiomatic approaches have been extended to parallel programs by Gries and Owiki[21][22], and Lamport[23]. This extension is summarized by proof outline logic[24][25]. In proof outline logic, the contracts (precondition, postcondition, and annotations) help verify both sequential reasoning (local reasoning) and concurrent reasoning (absence of interference). A typical proof outline in the notation of [25] is:

$\{precondition\}$
co
    $\ldots$
    $\{Annotation_0\}$
    $Command_0$
    $\{Annotation_1\}$
    $\ldots$
$\|$
    $\ldots$
    $\{Annotation_2\}$
    $\ldots$
oc
$\{postcondition\}$

The contents between $\{\ \}$ are the assertions, and other contents are program texts. Each pair of neighbour commands has an annotation between them. If a command may cause an annotation in another thread to be unstable (the command may change the value of that annotation from true to false), we say the command interferes with the annotation. For example, we say an atomic $Command_0$ does not interfere with $Annotation_2$ if $\vdash \{Annotation_2 \wedge Annotation_0\}\ Command_0\ \{Annotation_2\}$, where $Annotation_0$ is the assertion preceding $Command_0$.

The predicate transformer wlp[26][27], standing for weakest liberal precondition, provides formal calculus to compute the annotations in sequential programs. A program S is partially correct if the precondition (pre) implies its weakest liberal precondition ($wlpre = \mathsf{wlp}[S, post]$). The formula $pre \Rightarrow wlpre$ is called a verification condition.

Weakest liberal precondition reasoning transforms problems of verifying programs into problems of predicate proving, and therefore satisfiability-modulo-theories (SMT) solvers, such as Z3[28] and Simplify[29], may be used to automate the verification [30][31][32].

Boogie[30] is a verification tool for either object-oriented programs[31][32][33][34] or procedure-oriented programs[35] based on a weakest liberal precondition methodology. In a Boogie pipeline shown in Fig. 4.1[30], the source code (or an intermediate representation of the source code) is firstly translated into a program in BoogiePL[36] which is in a procedure-oriented style. Then, the type invariants and explicit loop invariants are inferred for the BoogiePL program. A Boogie compiler will generate first-order formulas as the verification conditions from the contracts and the implementations of the methods, and then use the theorem prover to verify them.

In object-oriented applications of Boogie, the verification conditions for methods' partial correctness are slightly different. The weakest liberal precondition $wlpre$ equals to $\mathsf{wlp}[S, post \wedge inv]$ where S is the program, $post$ is the postcondition, and inv is the type invariant. The verification condition is $pre \wedge inv \Rightarrow wlpre$. The method implementation in BoogiePL is composed of variable/constant declarations

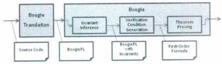

Figure 4.1: Boogie Pipeline

and statements. Each statement grammar rule has an associated wlp rule.[32]

$Stmt \rightarrow Stmt\ Stmt$

$$\mathsf{wlp}\,[S\ T, Q] = \mathsf{wlp}\,[S, \mathsf{wlp}\,[T, Q]]$$

$Stmt \rightarrow \textbf{skip};$

$$\mathsf{wlp}\,[\textbf{skip};, Q] = Q$$

$Stmt \rightarrow xs := Exprs;$

$$\mathsf{wlp}\,[xs := Es;, Q] = Q_{xs \leftarrow Es}$$

$Stmt \rightarrow x[Exprs] := Exprs;$

$$\mathsf{wlp}\,[m[jj] := E;, Q] = Q_{m \leftarrow m[jj := E]}$$

$Stmt \rightarrow \textbf{while}\ (Expr)\ Invs\ \{Stmt\}$

$$\mathsf{wlp}\,[\textbf{while}\ (E)\ \textbf{invariant}\ J; \{S\}, Q] = J \wedge$$

$$(\forall xs \cdot J \wedge E \Rightarrow \mathsf{wlp}\,[S, J]) \wedge (\forall xs \cdot J \wedge \neg E \Rightarrow Q)$$

where xs denotes the shared assignment targets of S

$Stmt \rightarrow \textbf{if}\ (Expr)\ \{Stmt\}\ \textbf{else}\ \{Stmt\}$

$$\mathsf{wlp}\,[\textbf{if}\ (E)\ \{S\}\ \textbf{else}\ \{T\}, Q]$$

$$= (E \Rightarrow \mathsf{wlp}\,[S, Q]) \wedge (\neg E \Rightarrow \mathsf{wlp}\,[T, Q])$$

$Stmt \rightarrow$ **havoc** xs; (to assign arbitrary values to the variables)

$\mathsf{wlp}\left[\textbf{havoc } xs;, Q\right] = (\forall xs \cdot Q)$

$Stmt \rightarrow$ **assert** $Expr$;

$\mathsf{wlp}\left[\textbf{assert } E;, Q\right] = E \wedge Q$

$Stmt \rightarrow$ **assume** $Expr$;

$\mathsf{wlp}\left[\textbf{assume } E;, Q\right] = E \Rightarrow Q$

$Stmt \rightarrow$ **call** $xs := P\,(EE)$;

The call statement is decoded into a sequence of other statements[32]. The sequence includes evaluation of input parameters, assertion of the precondition, copying old values of variables in modifies clause, initialization of the output parameters by arbitrary values, assumption of the postcondition, and the assignments to the output parameters. The precondition in this sequence is the conjunction of the object method precondition and the object invariant; and the postcondition is the conjunction of the method postcondition and the object invariant.

Consider the following procedure:

P (**in** ins, **out** $outs$) ;
  requires Pre;
  modifies gs;
  ensures $Post$;

The call to this procedure **call** $xs := P\,(EE)$; is decoded into:

$ins' := EE$;
assert Pre';
$gs' := gs$;

havoc $gs, outs'$;
assume $Post'$;
$xs := outs'$;

New variables ins', $outs'$, and gs' are introduced for each variable in ins, $outs$, and gs, respectively, The expression Pre' represents a copy of expression Pre in which all the variables from ins are substituted by the corresponding ones from ins'. The expression $Post'$ represents a copy of expression $Post$ in which (1) all the variables from ins are substituted by the corresponding ones from ins', (2) all the variables from $outs$ are substituted by the corresponding ones from $outs'$, and (3) all the occurrences of $\mathbf{old}(E)$, where E is a variable from gs, are substituted by the corresponding variable in gs'.

In some Boogie applications such as Chalice[31], Dafny[32][34], a concurrent extension to Spec#[33], and VCC[37], concurrent features are attempted. Most of these applications use the monitor[38] mechanism to provide mutual exclusion for asynchronous method calls. However, none of these applications solved the problem of automatically verifying parallel compositions.

The verification of HARPO/L uses an extension of Boogie technology with parallel compositions (I call it parallel Boogie). The following sections describe the translation from HARPO/L to parallel BoogiePL and the Verification Condition Generation from parallel BoogiePL to the verification conditions.

4.2 Verification of HARPO/L

In contrast to other Boogie applications[31][33][32], since HARPO/L is a static language, the input of the Boogie Translation is a specified object graph with specified ASTs in it, and the target BoogiePL program is an object-level program, rather than a class-level program. The specifications are instantiated along with the objects: the fields and shared variables of objects are allocated and the fields and variables in the class specifications are renamed into the instantiated fields and variables of objects. In addition, all the **modifies**-clauses of an object's methods are assigned to the set of the object's primitive fields unioned with the variables in the thread where the method is. In the following translations, $Tr[S]$ means the recursive translation of statement S.

$Tr[S_0 S_1] \qquad\qquad \equiv Tr[S_0]\ Tr[S_1]$

$Tr[E_0 := E_1] \qquad\qquad \equiv E_0 := E_1;$

$Tr[E.m(InE_0, \ldots, InE_k, OutE_0, \ldots, OutE_l)]$

$\quad \equiv$ **call** $OutE_0, \ldots, OutE_l := E.m(InE_0, \ldots, InE_k)$;

$Tr[$(**if** E **then** S_0 **else** S_1 **if**)$] \equiv$ **if** (E) $\{Tr[S_0]\}$ **else** $\{Tr[S_1]\}$

$Tr[$(**wh** E **invariant** J **do** S **wh**)$]$

$\quad \equiv$ **while** (E) **invariant** J; $\{Tr[S]\}$

$Tr[$(**accept** M_0 **when** $G_0 | \ \ldots\ | M_k$ **when** G_k **accept**)$]$

$\quad \equiv$ **assert** $G_0 \vee \ldots \vee G_k$; **havoc** all the fields and shared variables;

$Tr[$(**with** L **when** G **do** S **with**)$]$

$\quad \equiv$ **havoc** all the fields and shared variables;

$\qquad\qquad \langle$**await** $(G \wedge L \neq \text{"locked"})$ $L :=$ "locked"; $Tr[S]\rangle$

$Tr[$(**co** $S_0 \| \ldots \| S_k$ **co**)$] \qquad \equiv$ **co** $\{ Tr[S_0] \| \ldots \| Tr[S_k] \}$;

According to the semantics[12] of **accept** statement, the guards are evaluated first, and at least one of the guards should be true. Then the **accept** statement waits for a client's call. Note that the verification of the **accept** bodies (i.e. the implementation bodies of the methods) is treated separately (it generates extra verification conditions), and when verifying a thread, the **accept** statements are translated as shown. The verification conditions of implementation bodies are $G \wedge Inv \wedge pre \Rightarrow \mathsf{wlp}[B, Inv \wedge post]$, where G is the guard, Inv is the object invariant, pre is the precondition, $post$ is the postcondition, and B is the implementation body.

The await statement and the parallel composition are the main extensions to BoogiePL.

$$
\begin{aligned}
&pStmt \rightarrow \mathbf{co}\ \{Thrd\}\\
&Thrd \rightarrow Stmt\\
&Thrd \rightarrow Stmt \mid\mid Thrd\\
&Stmt \rightarrow \langle \mathbf{await}\ (Expr)\ Stmt \rangle
\end{aligned}
$$

We assume that await statements are not nested within other await statements. An await statement $\langle \mathbf{await}\ (E)\ S \rangle$ means "wait until E is $\mathfrak{true}$ and L is unlocked, then execute S" where L is a global lock. If E is always $\mathfrak{true}$, or S is $\mathbf{skip};$, the statement can be abbreviated.

$$\langle S \rangle \equiv \langle \mathbf{await}\ (\mathfrak{true})\ S \rangle$$

$$\langle \mathbf{await}\ (E) \rangle \equiv \langle \mathbf{await}\ (E)\ \mathbf{skip}; \rangle$$

The weakest liberal precondition of await statements is

$$\mathsf{wlp}\left[\langle \mathbf{await}\ (E)\ S \rangle, Q\right] = (E \Rightarrow \mathsf{wlp}\left[S, Q\right])$$

where xs denotes the syntactic read and/or write access targets of E and S.

4.3 Weakest Liberal Precondition of Parallel Compositions

I do not give a formal algebra rule of wlp for parallel compositions in this paper. Instead, I give an algorithm to compute it I assume an interleaving model of concurrency with mutual exclusion for await statement. For Section 4.3 I ignore issues that arise from data races; these issues are addressed in Section 4.5.

For the convenience of the description of my algorithm, each simple statement in a parallel composition is marked by two numbers: thread number and statement number. The thread number is straightforward; the statement number is given in the following way. Suppose the parallel composition has t threads. For each thread i, a left-to-right depth-first travel of that thread's abstract syntax tree is performed, and each statement node is given a statement number according to the order of being visited. Specifically, the earlier a statement is visited, the smaller its statement number is; the minimum statement number is 0, and the maximum statement number of thread i is denoted S_i. For example, all the statements of the thread in the following program are marked: the first subscripts of the names "*Stmt*" are the thread numbers, and the second subscripts are the statement numbers.

$Stmt_{i,0} : a := b;$
$Stmt_{i,1} :$ **if** $(a > 0)$ {
  $Stmt_{i,2} : c := a;$
} **else** {
  $Stmt_{i,3} : c := a + 10;$
  $Stmt_{i,4} :$ **while** $(c < 10)$ **invariant** $c \leq 10;$
    $\{Stmt_{i,5} : c := c + 1;\}$
}
$Stmt_{i,6} : e := c \times c;$

Now we define a number of helpful concepts related to program counters. The maximum program counter value for thread i is denoted S_i. A program counter expression α of a parallel composition, is a tuple $(\alpha_0, \alpha_1, \ldots, \alpha_{t-1})$ where $\alpha_i \in \{0, \ldots, S_i + 1\}$. The range of all program counter expressions is called control space, denoted C.

$$C \equiv \{\alpha \in \mathbb{N}^t | \forall i \in \{0, \ldots t\} \cdot \alpha_i \in \{0, \ldots, S_i + 1\}\}$$

Each thread i has a ghost variable, program counter P_i, and the program counter expression $(\alpha_0, \alpha_1, \ldots, \alpha_{t-1})$ means $(P_0 = \alpha_0) \wedge \ldots \wedge (P_{t-1} = \alpha_{t-1})$. If E is a first order formula on the program states, a program counter expression associated formula E_α, is an abbreviation of

$$E_\alpha \equiv ((P_0 = \alpha_0) \wedge \ldots \wedge (P_{t-1} = \alpha_{t-1}) \Rightarrow E)$$

A formula in *FormulaSet* form is a set of program counter expression associated first-order formulas, and the meaning of a *FormulaSet* is

$$\mathit{formula}\left(\bigcup_m \{E^m_{\alpha^m}\}\right) \equiv \bigwedge_m E^m_{\alpha^m}$$

To compute the weakest liberal precondition of a parallel composition, we compute a global invariant representing the relationship between program counters and program states. For the convenience of the computations, the global invariant G is in *FormulaSet* form which has $\prod_{i \in \{0, \ldots t\}} (S_i + 1)$ formulas in it: one formula for each program counter expression. The access of a formula in G with a particular program counter expression α is denoted G_α. The formula G_α is the condition which has to hold when the parallel execution enters a state that is at the beginning of the execution of $\mathit{Stmt}_{0,\alpha_0}, \ldots, \mathit{Stmt}_{t-1,\alpha_{t-1}}$ in each thread.

```
co {... Stmt_{0,1} ...
||   ... Stmt_{1,9} ...
||   ...
     if (E) {
          ... Stmt_{2,3}
     } else {...}
     Stmt_{2,6} ...
}
```

Take the above parallel composition as an example, according to the weakest liberal precondition reasoning, the formula $G_{1,9,3}$ must imply $\mathsf{wlp}[Stmt_{0,1}, G_{2,9,3}] \wedge \mathsf{wlp}[Stmt_{1,9}, G_{1,10,3}] \wedge \mathsf{wlp}[Stmt_{2,3}, G_{1,9,6}]$. Note that semantically $Stmt_{2,3}$ is followed by $Stmt_{2,6}$ rather than $Stmt_{2,4}$, which is in the **else**-clause.

A partial order $\leq$ is defined in control space C:

$$\alpha \leq \alpha' \textit{ iff } \forall i \in \{0, \ldots t\} \cdot \alpha_i \leq \alpha'_i$$

The computation of a formula G_α should be processed after the computations of all the formulas $G_{\alpha'}$ where $\alpha \leq \alpha'$.

The algorithm to compute the weakest liberal precondition of parallel compositions is

```
// Initialization
G := {true_α | α ∈ C}
G_{S_0+1,...,S_{t-1}+1} := postcondition
// Strengthening
for α ∈ C in a descending order
(any α is processed after the computations of all α' ≥ α)
    for i ∈ {0, ... t}
        if α_i < S_i + 1
            if Stmt_{i,α_i} is the last statement in
```

the then/else-clause in an if statement

suppose the *IF* statement is followed by $Stmt_{i,k}$

$tmp := \mathsf{wlp}\left[Stmt_{i,\alpha_i}, G_{\alpha_0,\ldots,\alpha_{i-1},k,\alpha_{i+1},\ldots,\alpha_{t-1}}\right]$

else if $Stmt_{i,\alpha_i}$ is the last statement in

the loop body of a while statement

suppose the loop invariant of the while statement is Inv

$tmp := \mathsf{wlp}[Stmt_{i,\alpha_i}, Inv]$

else

suppose $Stmt_{i,\alpha_i}$ is followed by a statement $Stmt_{i,k}$

if $Stmt_{i,\alpha_i}$ is an if statement

suppose the guard is E,

the first statement in the then-clause is $Stmt_{i,m}$, and

the first statement in the else-clause is $Stmt_{i,n}$

$tmp := \left(E \Rightarrow G_{\alpha_0,\ldots,\alpha_{i-1},m,\alpha_{i+1},\ldots,\alpha_{t-1}}\right)$

$\wedge \left(\neg E \Rightarrow G_{\alpha_0,\ldots,\alpha_{i-1},n,\alpha_{i+1},\ldots,\alpha_{t-1}}\right)$

else if $Stmt_{i,\alpha_i}$ is an while statement

suppose the guard is E, the loop invariant is J, and

the first statement in the loop body is $Stmt_{i,m}$

$tmp := J \wedge \left(E \Rightarrow G_{\alpha_0,\ldots,\alpha_{i-1},m,\alpha_{i+1},\ldots,\alpha_{t-1}}\right)$

$\wedge \left(\neg E \Rightarrow G_{\alpha_0,\ldots,\alpha_{i-1},k,\alpha_{i+1},\ldots,\alpha_{t-1}}\right)$

else

$tmp := \mathsf{wlp}\left[Stmt_{i,\alpha_i}, G_{\alpha_0,\ldots,\alpha_{i-1},k,\alpha_{i+1},\ldots,\alpha_{t-1}}\right]$

end if

end if

$G_\alpha := G_\alpha \wedge tmp$

end if

end for

end for

// Generating Results

$wlpre := G_{0,\ldots,0}$

The initialization assigns an over-weakened condition true to each formula in G, and then performs a bottom-up strengthening to all the formulas. Finally, each G_α is the weakest possible annotation at control state α. The weakest liberal precondition of the parallel composition is the formula $G_{0,\ldots,0}$.

The number of wlp computations is $t \times \prod_{i \in \{0,\ldots t\}} (S_i + 1)$ where t is the number of threads, and S_i is the number of internal annotations in thread i.

The global invariant G is also helpful for generating a proof outline for the parallel composition in a very simple way. The annotation preceding $Stmt_{i,k}$ is $P_i = k$, and the annotated threads along with the precondition and postcondition of the parallel composition and the global invariant compose a valid proof outline. We say it is a weakest valid proof outline for the given postcondition.

4.4 Example

The first example shows that the result of the algorithm covers all the interleaving possibilities. Consider the following parallel composition.

co {

$Stmt_{0,0} : \langle x := x + 3; \rangle$
$Stmt_{0,1} : \langle x := x + 2; \rangle$

||

$Stmt_{1,0} : \langle x := x \times 3; \rangle$
$Stmt_{1,1} : \langle x := x \times 2; \rangle$

}

We give this parallel composition a specification in which the precondition is $Q_0 : x = 2$ and the postcondition is $Q_1 : x \in \{17, 20, 22, 32, 42\}$. We want to verify that this program satisfies this specification. By enumerating all the interleaving possibilities, we know that a precondition of $x = 2$ leads to a postcondition of $x \in \{17, 20, 22, 32, 34, 42\}$. Since in Q_1, the range of x does not contain 34, the verification result should be negative.

The global invariant G that results from applying the algorithm is (in *FormulaSet* form)

$$G = \Big\{\mathsf{false}_{0,0}, (x \in \{6\})_{0,1}, (x \in \{12, 15, 17, 27, 37\})_{0,2},$$
$$\mathsf{false}_{1,0}, (x \in \{9\})_{1,1}, (x \in \{15, 18, 20, 30, 40\})_{1,2},$$
$$\left(x \in \left\{\frac{17}{6}, \frac{10}{3}, \frac{11}{3}, \frac{16}{3}, 7\right\}\right)_{2,0}, \left(x \in \left\{\frac{17}{2}, 10, 11, 16, 21\right\}\right)_{2,1},$$
$$(x \in \{17, 20, 22, 32, 42\})_{2,2}\Big\}$$

$$wlpre = \mathsf{false}$$

Therefore, the complete weakest proof outline is:

$\{wlpre : \mathsf{false}\}$

$\{$ **global invariant** G :

$(P_0 = 0 \wedge P_1 = 0 \Rightarrow \mathsf{false})$
$\wedge (P_0 = 0 \wedge P_1 = 1 \Rightarrow x \in \{6\})$
$\wedge (P_0 = 0 \wedge P_1 = 2 \Rightarrow x \in \{12, 15, 17, 27, 37\})$
$\wedge (P_0 = 1 \wedge P_1 = 0 \Rightarrow \mathsf{false})$
$\wedge (P_0 = 1 \wedge P_1 = 1 \Rightarrow x \in \{9\})$
$\wedge (P_0 = 1 \wedge P_1 = 2 \Rightarrow x \in \{15, 18, 20, 30, 40\})$
$\wedge \left(P_0 = 2 \wedge P_1 = 0 \Rightarrow x \in \left\{\frac{17}{6}, \frac{10}{3}, \frac{11}{3}, \frac{16}{3}, 7\right\}\right)$
$\wedge \left(P_0 = 2 \wedge P_1 = 1 \Rightarrow x \in \left\{\frac{17}{2}, 10, 11, 16, 21\right\}\right)$
$\wedge (P_0 = 2 \wedge P_1 = 2 \Rightarrow x \in \{17, 20, 22, 32, 42\})$ $\}$

co $\{$

$\{P_0 = 0\}$
$Stmt_{0,0} : \langle x := x + 3; \rangle$
$\{P_0 = 1\}$
$Stmt_{0,1} : \langle x := x + 2; \rangle$
$\{P_0 = 2\}$

$\|$

$\{P_1 = 0\}$

$Stmt_{1,0} : \langle x := x \times 3; \rangle$
$\{P_1 = 1\}$
$Stmt_{1,1} : \langle x := x \times 2; \rangle$
$\{P_1 = 2\}$

}

$\{postcondition : x \in \{17, 20, 22, 32, 42\}\}$

Because the specified precondition $Q_0 : x = 2$ does not imply the weakest liberal precondition $wlpre$: $\mathfrak{false}$, the verification result is negative, as expected. We can also check the local reasoning and the interference freedom of this proof outline, and find that this proof outline is valid.

Now, if we rewrite the postcondition as $16 < x < 44$, the verification result should be positive because this postcondition is weaker than the strongest postcondition from the precondition $Q_0 : x = 2$.

The global invariant G in the result is:

$$G = \left\{ \left(\frac{11}{6} < x < \frac{7}{3}\right)_{0,0}, \left(\frac{11}{2} < x < 18\right)_{0,1}, (11 < x < 39)_{0,2}, \right.$$
$$\left(\frac{7}{3} < x < \frac{16}{3}\right)_{1,0}, (7 < x < 20)_{1,1}, (14 < x < 42)_{1,2}$$
$$\left. \left(\frac{8}{3} < x < \frac{22}{3}\right)_{2,0}, (8 < x < 22)_{2,1}, (16 < x < 44)_{2,2} \right\}$$

The weakest proof outline is

$\left\{wlpre : \frac{11}{6} < x < \frac{7}{3}\right\}$

{ **global invariant** G :

$\left(P_0 = 0 \wedge P_1 = 0 \Rightarrow \frac{11}{6} < x < \frac{7}{3}\right)$
$\wedge \left(P_0 = 0 \wedge P_1 = 1 \Rightarrow \frac{11}{2} < x < 18\right)$
$\wedge \left(P_0 = 0 \wedge P_1 = 2 \Rightarrow 11 < x < 39\right)$
$\wedge \left(P_0 = 1 \wedge P_1 = 0 \Rightarrow \frac{7}{3} < x < \frac{16}{3}\right)$
$\wedge \left(P_0 = 1 \wedge P_1 = 1 \Rightarrow 7 < x < 20\right)$

$\wedge (P_0 = 1 \wedge P_1 = 2 \Rightarrow 14 < x < 42)$
$\wedge (P_0 = 2 \wedge P_1 = 0 \Rightarrow \frac{8}{3} < x < \frac{22}{3})$
$\wedge (P_0 = 2 \wedge P_1 = 1 \Rightarrow 8 < x < 22)$
$\wedge (P_0 = 2 \wedge P_1 = 2 \Rightarrow 16 < x < 44)$ }

co {

$\{P_0 = 0\}$
$Stmt_{0,0} : \langle x := x + 3; \rangle$
$\{P_0 = 1\}$
$Stmt_{0,1} : \langle x := x + 2; \rangle$
$\{P_0 = 2\}$

||

$\{P_1 = 0\}$
$Stmt_{1,0} : \langle x := x \times 3; \rangle$
$\{P_1 = 1\}$
$Stmt_{1,1} : \langle x := x \times 2; \rangle$
$\{P_1 = 2\}$

}
$\{postcondition : 16 < x < 44\}$

Because $x = 2 \Rightarrow \frac{11}{6} < x < \frac{7}{3}$, the verification result is positive, as expected.

4.5 Enhanced Weakest Precondition of Parallel Compositions

4.5.1 Absence of Data Race

Two data accesses (reads or writes) are said to *conflict* if they access the same location and are not both reads.[1]

[1] An alternative definition also considers two reads of the same location to be a conflict. (See, for example [12].) This alternative definition can also be adopted.

Two statements are said to *potentially conflict* if they may make conflicting accesses when run concurrently from some shared starting state. For two simple statements or expressions, we can easily compute the weakest precondition that ensures that they avoid conflict. For example $\mathsf{wncp}[x := 0, y := 1] = \mathfrak{true}$, meaning that there is no potential conflict, whereas $\mathsf{wncp}[y := 0, y := 1] = \mathfrak{false}$ and $\mathsf{wncp}[a[i] := 0, a[j] := 1] = (i \neq j)$. Await statements do not conflict with each other, but may potentially conflict with unprotected statements: $\mathsf{wncp}[\langle x := 0 \rangle, \langle x := 1 \rangle] = \mathfrak{true}$ whereas $\mathsf{wncp}[\langle x := 0 \rangle, x := 1] = \mathfrak{false}$. We can generalize wncp to sets of more than two simple statements or expressions:

$$\mathsf{wncpc}\,[X] = \bigwedge_{x,y \in X | x \neq y} \mathsf{wncp}\,[x, y]$$

Define a *synchronization point* as either the start or the end of an await statement. An execution of a concurrent program can be thought of as a sequence of actions interleaved from the various threads. The (executions of) synchronization points split an execution into *segments*. An execution has a *data race* if actions from different threads make conflicting data accesses in the same segment, i.e., without any intervening synchronization point.

The predicate that characterizes those initial states that ensure data-race-free executions is a concurrent program's *weakest data-race-free precondition*, written $\mathsf{wdrfp}[S]$. A program that has potentially conflicting statements in different threads may still be race free, as the programmer may use mechanisms such as semaphores to prevent conflicting statements from executing in the same period.

For example, consider the following parallel composition S.

co {
  $Stmt_{0,0} : \langle \textbf{await } s \geq 0 \rangle$
  $Stmt_{0,1} : x := 0;$
  $Stmt_{0,2} : \langle s := -1; \rangle$

apply the algorithm with a specified postcondition $(x = 0 \vee x = 1)$, the result is

$$G = \Big\{ (s > 0 \vee s < 0)_{0,0}, (s < 0)_{0,1}, \mathfrak{true}_{0,2}, \mathfrak{true}_{0,3},$$
$$(s > 0)_{1,0}, \mathfrak{false}_{1,1}, \mathfrak{true}_{1,2}, \mathfrak{true}_{1,3},$$
$$\mathfrak{true}_{2,0}, \mathfrak{true}_{2,1}, (x = 0 \vee x = 1)_{2,2}, (x = 0 \vee x = 1)_{2,3},$$
$$\mathfrak{true}_{3,0}, \mathfrak{true}_{3,1}, (x = 0 \vee x = 1)_{3,2}, (x = 0 \vee x = 1)_{3,3} \Big\}$$
$$wlpre = s > 0 \vee s < 0$$

As shown, $wlpre'$ implies $s \neq 0$ which guarantees the absence of data races.

$wlpre'$ is a similar concept to $winp$, weakest invariant of a postcondition, introduced in [39] which did not give the computation method.

4.5.2 Absence of Deadlock

In the execution of a parallel composition, a deadlock occurs when (1) each thread in the parallel composition reaches either the end of the thread or an await statement, (2) at least one thread reaches an await statement, and (3) the guard of each reached await statement is in the state $\mathfrak{false}$.

co {// thread 0

...

$\{Annot_{0,k}\}$
$Stmt_{0,k} : \langle \textbf{await}\ (E_0)\ \ldots \rangle$

...

$\{Annot_{0,S_0}\}$

|| // thread 1

...

$\{Annot_{1,l}\}$
$Stmt_{1,l} : \langle \textbf{await}\ (E_1)\ \ldots \rangle$

...

$\{Annot_{1,S_1}\}$

}

For example, the above two-threaded parallel composition has three possibilities to have a deadlock: thread 0 is waiting for E_0 to become $\mathfrak{true}$ and thread 1 reaches the end; thread 1 is waiting for E_1 to become $\mathfrak{true}$ and thread 0 reaches the end; and thread 0 is waiting for E_0 to become $\mathfrak{true}$ and thread 1 is waiting for E_1 to become $\mathfrak{true}$.

We say the weakest precondition that ensures avoidence of any deadlock states is the weakest deadlock-free precondition of S, denoted $\mathsf{wdlfp}[S]$.

We improve the algorithm again so that it can compute a precondition $wlpre''$ which ensures the postcondition, and implies both its weakest data-race-free precondition and weakest deadlock-free precondition, i.e. $wlpre'' = wlpre \wedge \mathsf{wdrfp}[S] \wedge \mathsf{wdlfp}[S]$.

The solution is similar to the one of weakest data-race-free precondition. In this example, if we strengthen the initial state of G_{k,S_1+1} to E_0, $G_{S_0+1,l}$ to E_1, and $G_{k,l}$ to $E_0 \vee E_1$, the result of $wlpre''$ will guarantee the absence of deadlock. Formally, the initialization of the enhanced (again) algorithm is:

// Initialization
$G := \{\mathfrak{true}_\alpha \mid \alpha \in C\}$
for each statement pair $Stmt_{i,k}$ and $Stmt_{j,l}$
  for each $\alpha \in \{(\alpha_0, \ldots, \alpha_{t-1}) \in C \mid \alpha_i = k \wedge \alpha_j = l\}$
    $G_\alpha := G_\alpha \wedge \mathsf{wncpc}[\{Stmt_{i,k}, Stmt_{j,l}\}]$
  end for
end for
for each α other than $(S_0 + 1, \ldots, S_{t-1} + 1)$ such that,
    for all $i \in \{0, \ldots t\}$, $Annot_{i,\alpha_i}$ is either the last annotation of thread i

Note that $\mathfrak{false}_{1,1}$ is for the data races between $Stmt_{0,1}$ and $Stmt_{1,1}$, and $(s > 0 \vee s < 0)_{0,0}$, $(s > 0)_{0,3}$, and $(s < 0)_{3,0}$ are for the deadlocks on $Stmt_{0,0}$ or $Stmt_{1,0}$. The result of the enhanced algorithm is

$$\begin{aligned}
G = \Big\{ & (s < 0)_{0,0}, (s \leq 0)_{0,1}, \mathfrak{true}_{0,2}, (s > 0)_{0,3}, \\
& \mathfrak{false}_{1,0}, \mathfrak{false}_{1,1}, \mathfrak{true}_{1,2}, \mathfrak{true}_{1,3}, \\
& \mathfrak{false}_{2,0}, \mathfrak{true}_{2,1}, (x = 0 \vee x = 1)_{2,2}, (x = 0 \vee x = 1)_{2,3}, \\
& (s < 0)_{3,0}, \mathfrak{true}_{3,1}, (x = 0 \vee x = 1)_{3,2}, (x = 0 \vee x = 1)_{3,3} \Big\} \\
wlpre = {} & (s < 0)
\end{aligned}$$

The weakest liberal precondition $wlpre'' = s < 0$ gaurantees both the absence of data races and the absence of deadlocks.

4.6 Grainless Semantics Issues

A grainless semantics[11][12] for concurrent programs posits that any data race during execution is an error of the worst sort; one that makes no guarantees, not even of termination or any indication of error. One benefit of grainless semantics is that it allows compilers or memory systems to reorder data accesses and to make any other optimizations that are valid under a sequential model, provided they do not expand the set of data accesses in regions between synchronization points. A statement $x := y; x := x+y;$ can be optimized as $x := 2*y;$. It is safe to ignore the possibility of a concurrent write to x or y since, when the code is executed, either there will be a data race, in which case "anything goes", or there will not, in which case the transformation makes no difference. When verifying a program under the assumption of grainless semantics we must ensure that there are no data races, as discussed in Section 4.5.1.

However, as long as we do that, it is safe to ignore the possibility of concurrent accesses, in other words, to assume that all interleaving is at the synchronization points.

Therefore, when verifying a program, we only need to consider program counter positions that are *synchronization points*, or that immediately follow branches. This greatly reduces the size of the control state space C and thus the time for verification. For example, in the following program, $Stmt_{0,j}Stmt_{0,j+1}$ is considered as an atomic segment, and $Stmt_{1,k}Stmt_{1,k+1}$ is considered as another atomic segment.

co {

...

// a synchronization point

$Stmt_{0,j}$ // no guarantee of mutual exclusion

// not a synchronization point

$Stmt_{0,j+1}$; // no guarantee of mutual exclusion

// a synchronization point

...

||

...

// a synchronization point

$Stmt_{1,j}$ // no guarantee of mutual exclusion

// not a synchronization point

$Stmt_{1,k+1}$; // no guarantee of mutual exclusion

// a synchronization point

...

}

With an assumption of the absence of data races, the order of the executions has no impact on the behaviour, so we can consider $Stmt_{0,j}Stmt_{0,j+1}$ atomic, and $Stmt_{1,k}Stmt_{1,k+1}$ atomic.

Therefore, the global invariant does not have to contain the formula associated with $(j+1, k+1)$, $(j, k+1)$, or $(j+1, k)$. However, it must contain the formulas for control states in which all the program counters point to synchronization points.

Take the following program for example.

$$
\begin{array}{l}
\textbf{co}\ \{ \\
\quad \mathit{Stmt}_{0,0} : \langle \textbf{await}\ s \geq 0 \rangle \\
\quad \mathit{Stmt}_{0,1} : x := 0; \\
\quad \mathit{Stmt}_{0,2} : y := x + 1; \\
\quad \mathit{Stmt}_{0,3} : \langle s := -1; \rangle \\
\ \| \\
\quad \mathit{Stmt}_{1,0} : \langle \textbf{await}\ s \leq 0 \rangle \\
\quad \mathit{Stmt}_{1,1} : x := 1; \\
\quad \mathit{Stmt}_{1,2} : y := x; \\
\quad \mathit{Stmt}_{1,3} : \langle s := 1; \rangle \\
\} \\
\{\mathit{postcondition} : y = 1\}
\end{array}
$$

The conflicting statement pairs are $(\mathit{Stmt}_{0,1}, \mathit{Stmt}_{1,1})$, $(\mathit{Stmt}_{0,1}, \mathit{Stmt}_{1,2})$, $(\mathit{Stmt}_{0,2}, \mathit{Stmt}_{1,1})$, and $(\mathit{Stmt}_{0,2}, \mathit{Stmt}_{1,2})$, so we initialize four elements in the global invariant G as false: $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, and $P_{2,2}$. The result after running the algorithm is

$$
\begin{aligned}
G = \Big\{ & (s < 0 \lor s > 0)_{0,0}\,, (s < 0)_{0,1}\,, (s < 0)_{0,2}\,, \mathsf{true}_{0,3}, \mathsf{true}_{0,4}, \\
& (s > 0)_{1,0}\,, \mathsf{false}_{1,1}, \mathsf{false}_{1,2}, \mathsf{true}_{1,3}, \mathsf{true}_{1,4}, \\
& (s > 0)_{2,0}\,, \mathsf{false}_{2,1}, \mathsf{false}_{2,2}, (x = 0)_{2,3}\,, (x = 0)_{2,4}\,, \\
& \mathsf{true}_{3,0}, \mathsf{true}_{3,1}, (x = 1)_{3,2}\,, (y = 1)_{3,3}\,, (y = 1)_{3,4}\,, \\
& \mathsf{true}_{4,0}, \mathsf{true}_{4,1}, (x = 1)_{4,2}\,, (y = 1)_{4,3}\,, (y = 1)_{4,4} \Big\} \\
\mathit{wlpre} = {} & (s < 0 \lor s > 0)
\end{aligned}
$$

The synchronizations occur when $P_0 \in \{0, 1, 3\}$ and $P_1 \in \{0, 1, 3\}$. Therefore the global invariant G does not need to contain the condition of $P_0 = 2$ or $P_1 = 2$. In other words, $Stmt_{0,1} Stmt_{0,2}$ is considered as an atomic segment, and so is $Stmt_{1,1} Stmt_{1,2}$. The result is

$$\begin{aligned}
G' = \Big\{ & (s < 0 \vee s > 0)_{0,0}, (s < 0)_{0,1}, \mathfrak{true}_{0,3}, \mathfrak{true}_{0,4}, \\
& (s > 0)_{1,0}, \mathfrak{false}_{1,1}, \mathfrak{true}_{1,3}, \mathfrak{true}_{1,4}, \\
& \mathfrak{true}_{3,0}, \mathfrak{true}_{3,1}, (y = 1)_{3,3}, (y = 1)_{3,4}, \\
& \mathfrak{true}_{4,0}, \mathfrak{true}_{4,1}, (y = 1)_{4,3}, (y = 1)_{4,4} \Big\} \\
wlpre = {} & (s < 0 \vee s > 0)
\end{aligned}$$

Note that all the conditions in G' are exactly the same as those in G, except that G contains the conditions of $P_0 = 2$ or $P_1 = 2$.

Another example shows that the positions immediately following branches need to be considered in the global invariant.

co {
  $Stmt_{0,0} : x := a;$
  if $(x \leq 0)$ {
    $Stmt_{0,1} : y := 3;$
  } **else** {
    $Stmt_{0,2} : z := 2;$
  }
||
  $Stmt_{1,0} : w := a;$
  if $(w \geq 0)$ {
    $Stmt_{1,1} : z := -3;$
  } **else** {
    $Stmt_{1,2} : y := -2;$

```
    }
}
{postcondition : true}
```

The conflicting statements assigning y and z are not in any await statement. However, if $a = 0$ in the pre-execution state, the data race will not occur. If we do not consider the positions that $P_0 = 1$, $P_0 = 2$, $P_1 = 1$, or $P_1 = 2$, we will obtain a $wlpre'' = \mathsf{false}$, which indicates no pre-execution states can establish the postcondition and ensure the absence of the data races and the absence of the deadlocks. If we consider those positions, we will obtain the correct result

$$
\begin{aligned}
G = \Big\{ & (a = 0)_{0,0}, (a \leq 0)_{0,1}, (a > 0)_{0,2}, \mathsf{true}_{0,3}, \\
& (a \geq 0)_{1,0}, \mathsf{true}_{1,1}, \mathsf{false}_{1,2}, \mathsf{true}_{1,3}, \\
& (a < 0)_{2,0}, \mathsf{false}_{2,1}, \mathsf{true}_{2,2}, \mathsf{true}_{2,3}, \\
& \mathsf{true}_{3,0}, \mathsf{true}_{3,1}, \mathsf{true}_{3,2}, \mathsf{true}_{3,3} \Big\} \\
wlpre'' = {} & (a = 0)
\end{aligned}
$$

4.7 One More Example

Consider the following problem: in a limited computing environment whereas the only allowed mathematic operations are addition, comparisons, and Boolean operations, we want to find a natural number a so that $a\%31 = 19$ and $a\%83 = 62$. We use the program (with grainless semantics and in BoogiePL's style) below, but we do not know if it is correct.

```
a, b, eq0, eq1 := 19, 62, false, false;
co {
    while (¬eq0) {
```

$\langle$**await** $(a < b)$ {

$a := a + 31;$

$eq_0 = (a = b)$;

})

}

||

while $(\neg eq_1)$ {

$\langle$**await** $(b < a)$ {

$b := b + 83;$

$eq_1 = (a = b)$;

})

}

}

In this program, a and b are shared variables, and eq_0 and eq_1 are local variables. The requirement indicates the postcondition must imply $a\%31 = 19 \wedge a\%83 = 62$, and I, as a verifier, will give a postcondition $a = b \wedge a\%31 = 19 \wedge b\%83 = 62$. I will also give a precondition of the parallel composition that $a \neq b \wedge a\%31 = 19 \wedge b\%83 = 62$ which is guaranteed by the initialization. In addition, the loop invariants are needed for verification (given below). Now, the specification (including precondition, postcondition, and loop invariants) is complete, and the algorithm will be applied to compute the global invariant of the following proof outline.

{*precondition* : $a \neq b \wedge a\%31 = 19 \wedge b\%83 = 62$}

{**global invariant** $G =$?}

co {

{$P_0 = 0$}

$Stmt_{0,0}$: **while** $(\neg eq_0)$ **invariant** $inv_0 : a\%31 = 19 \wedge eq_0 = (a = b)$ {

{$P_0 = 1$}

$Stmt_{0,1}$: $\langle$**await** $(a < b)$ {

$a := a + 31;$

$eq_0 := (a = b)$;

$\})$

$\{P_0 = 2\}$

$\}$

$\{P_0 = 3\}$

$||$

$\{P_1 = 0\}$

$Stmt_{1,0}$: **while** $(\neg eq_1)$ **invariant** $inv_1 : b\%83 = 62 \wedge eq_1 = (a = b)$ {

$\{P_1 = 1\}$

$Stmt_{1,1}$: ⟨**await** $(b < a)$ {

$b := b + 83;$

$eq_0 := (a = b)$;

$\})$

$\{P_1 = 2\}$

$\}$

$\{P_1 = 3\}$

$\}$

$\{postcondition : a = b \wedge a\%31 = 19 \wedge b\%83 = 62\}$

At the beginning of the procedure of the algorithm, $G_{1,1}$ is initialized to $a \neq b$, $G_{1,2}$ is initialized to $a < b$, and $G_{2,1}$ is initialized to $b > a$, to prevent deadlocks.

The result after we run the algorithm is

$G_{0,0} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_0 = (a = b) \wedge eq_1 = (a = b)$

$\wedge (\neg eq_0 \wedge \neg eq_1 \Rightarrow \mathsf{false}) \wedge (eq_0 \vee eq_1 \Rightarrow a = b)$

$G_{0,1} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_0 = (a = b) \wedge \neg eq_0 \wedge b < a$

$G_{1,0} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_1 = (a = b) \wedge \neg eq1 \wedge a < b$

$G_{0,2} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_0 = (a = b) \wedge eq_1 = (a = b)$

$G_{1,1} : a \neq b \wedge (a < b \Rightarrow a\%31 = 19 \wedge (b < a + 31 \Rightarrow b\%83 = 62))$

$\wedge (b < a \Rightarrow b\%83 = 62 \wedge (a < b + 83 \Rightarrow a\%31 = 19))$

$G_{2,0} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_0 = (a = b) \wedge eq_1 = (a = b)$

$G_{0,3} : a\%31 = 19 \wedge eq_0 = (a = b) \wedge (\neg eq_0 \Rightarrow a < b) \wedge (eq_0 \Rightarrow a = b \wedge b\%83 = 62)$

$G_{1,2} : b\%83 = 62 \wedge eq_1 = (a = b) \wedge (a < b \Rightarrow a\%31 = 19)$

$G_{2,1} : a\%31 = 19 \wedge eq_0 = (a = b) \wedge (b < a \Rightarrow b\%83 = 62)$

$G_{3,0} : b\%83 = 62 \wedge eq_1 = (a = b) \wedge (\neg eq_1 \Rightarrow b < a) \wedge (eq_1 \Rightarrow a = b \wedge a\%31 = 19)$

$G_{1,3} : a < b \wedge a\%31 = 19$

$G_{2,2} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_0 = (a = b) \wedge eq_1 = (a = b)$

$G_{3,1} : b < a \wedge b\%83 = 62$

$G_{2,3} : a\%31 = 19 \wedge eq_0 = (a = b)$

$G_{3,2} : b\%83 = 62 \wedge eq_1 = (a = b)$

$G_{3,3} : a = b \wedge a\%31 = 19 \wedge b\%83 = 62$

The specified precondition does not imply $G_{0,0}$, therefore the program is incorrect. Through a brief observation, we can find that there are potential deadlocks in the execution of the program. Imagine that a is 2201 and b is 2220 when both threads reach their own await statements. Thread 0 will go on, assign a to 2220, assign eq_0 to $\mathfrak{true}$, and then quit the loop. In this case, thread 1 will wait forever, and thus there will be a deadlock.

Now, the deadlock problem is somehow fixed as below, and we must re-verify its correctness.

$a, b, eq_0, eq_1 := 19, 62, \mathfrak{false}, \mathfrak{false};$
co {
  while $(\neg eq_0)$ {
    $\langle$**await** $(a \le b)$ {
      if $(a < b)$ $\{a := a + 31;\}$
      $eq_0 := (a = b)$;
    }$\rangle$}
$\|$

while $(\neg eq_1)$ {
    $\langle$**await** $(b \leq a)$ {
        if $(b < a)$ $\{b := b + 83;\}$
        $eq_1 := (a = b)$;
    }$\rangle$}
}

The specification remains the same. $G_{1,1}$, $G_{1,3}$, and $G_{3,1}$ are initialized to true, $a \leq b$, and $b \leq a$, respectively. The result of the algorithm is

$$
\begin{aligned}
&G_{0,0} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_0 = (a = b) \wedge eq_1 = (a = b) \\
&\quad \wedge (\neg eq_0 \wedge eq_1 \Rightarrow a \leq b) \wedge (eq_0 \wedge \neg eq_1 \Rightarrow b \leq a) \wedge (eq_0 \wedge eq_1 \Rightarrow a = b) \\
&G_{0,1} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_0 = (a = b) \\
&G_{1,0} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_1 = (a = b) \\
&G_{0,2} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_0 = (a = b) \wedge eq_1 = (a = b) \wedge (\neg eq_0 \Rightarrow b \leq a) \\
&G_{1,1} : (a < b \Rightarrow b\%83 = 62 \wedge eq_1 = (a + 31 = b) \\
&\qquad \wedge b \leq a + 31 \wedge (a + 31 = b \Rightarrow a\%31 = 19)) \\
&\quad \wedge (a = b \Rightarrow a\%31 = 19 \wedge b\%83 = 62 \wedge eq_0 = (a = b) \wedge eq_1 = (a = b)) \\
&\quad \wedge (b < a \Rightarrow a\%31 = 19 \wedge eq_0 = (a = b + 83) \\
&\qquad \wedge a \leq b + 83 \wedge (a = b + 83 \Rightarrow b\%83 = 62)) \\
&G_{2,0} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_0 = (a = b) \wedge eq_1 = (a = b) \wedge (\neg eq_1 \Rightarrow a \leq b) \\
&G_{0,3} : a\%31 = 19 \wedge eq_0 = (a = b) \wedge (\neg eq_0 \Rightarrow a \leq b) \wedge (eq_0 \Rightarrow a = b \wedge b\%83 = 62) \\
&G_{1,2} : b\%83 = 62 \wedge eq_1 = (a = b) \wedge a \leq b \wedge (a = b \Rightarrow a\%31 = 19) \\
&G_{2,1} : a\%31 = 19 \wedge eq_0 = (a = b) \wedge b \leq a \wedge (a = b \Rightarrow b\%83 = 62) \\
&G_{3,0} : b\%83 = 62 \wedge eq_1 = (a = b) \wedge (\neg eq_1 \Rightarrow b < a) \wedge (eq_1 \Rightarrow a = b \wedge a\%31 = 19) \\
&G_{1,3} : a \leq b \wedge a\%31 = 19 \\
&G_{2,2} : a\%31 = 19 \wedge b\%83 = 62 \wedge eq_0 = (a = b) \wedge eq_1 = (a = b) \\
&G_{3,1} : b \leq a \wedge b\%83 = 62 \\
&G_{2,3} : a\%31 = 19 \wedge eq_0 = (a = b) \\
&G_{3,2} : b\%83 = 62 \wedge eq_1 = (a = b) \\
&G_{3,3} : a = b \wedge a\%31 = 19 \wedge b\%83 = 62
\end{aligned}
$$

Because the specified precondition implies $G_{0,0}$, the verification result is positive.

Chapter 5

Conclusion and Future Work

Reconfigurable computing is a computation solution with higher efficiency than software solutions and higher flexibility than hardware solutions. In reconfigurable architectures, Coarse-grained reconfigurable architectures (CGRAs) are more efficient, for many applications, than fine-grained architectures such as widely used field-programmable gate arrays (FPGAs).[3][1]

The HARPO project aims to define a high-level object-oriented programming language which is compiled into CGRA configurations. The objects in HARPO/L are mapped into reconfigurable datapath units (rDPUs), and the references and method calls are mapped into interconnections between those rDPUs. Besides, HARPO/L is

- Static: All the allocations and connections of objects are done at compile-time due to the nature of hardware configurations.
- Concurrent[8]: Each object has a number of threads and is considered as an active datapath after the compiling. The threads of all the objects are concurrently executed.
- Grainless[12]: The semantics of the language does not depend on the granularity

of memory access.

5.1 Contributions

Dataflow synthesis is an important component in the HARPO/L compiling process. One of the contributions of this thesis is the design and implementation (in Java) of the dataflow synthesis module. This thesis defines a number of types of dataflow graph nodes in CHP notation, and uses a high-level data flow analysis algorithm, which analyzes object-oriented programs, to generate dataflow graphs for HARPO/L. This dataflow synthesis is extendable to most object-oriented parallel languages.

The other main contribution of this thesis is the design of a verification system for HARPO/L. The architecture of a verifying compiler based on Boogie[30] technology for HARPO/L is constructed, and an algorithm to compute weakest global invariant and weakest liberal precondition (wlp) of parallel compositions is proposed. This algorithm fills the gap between state-of-art wlp approaches and verifying languages with parallel compositions. The weakest proof outline can be generated from the algorithm results. Moreover, the variants of this algorithm computing enhanced global invariant and enhanced weakest liberal precondition can be used to verify absence of data race or absence of deadlock along with the behavioural correctness. This algorithm also has good interplay with grainless semantics.

5.2 Future Work

The following module remains incomplete in HARPO/L project.

- Front-end: The object graph generation has not yet been implemented.

- Middle module: I only implemented a very low level optimization of the dataflow graph. More optimizations are needed.
- Back-end: The scheduling module has not yet been implemented.
- Verification: The proposed verification system has not yet been implemented. Besides, an idea of verifying with temporal logic[41] is suggested by Dr. Theodore S. Norvell.

Bibliography

[1] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[2] Jonathan Rose, Abbas El Gamal, Senior Member, and Albert Sangiovanni-vincentelli. Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency. *Proceedings of the IEEE*, 25:1217–1225, 1990.

[3] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 642–649, Piscataway, NJ, USA, 2001. IEEE Press. ISBN 0-7695-0993-2.

[4] Joo M.P. Cardoso and Pedro C. Diniz. *Compilation Techniques for Reconfigurable Architectures*. Springer Publishing Company, Incorporated, 2008. ISBN 0387096701, 9780387096704.

[5] Xiangwen Li. Analysis and compilation techniques for HARPO/L. Master's thesis, Memorial University of Newfoundland, 2008.

[6] Mohammed Ashraful Alam Tuhin and Theodore S. Norvell. Compiling parallel applications to coarse-grained reconfigurable architectures. In *CCECE*, 2008.

[7] John Teifel and Rajit Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *ASYNC*, pages 17–27, 2004.

[8] Theodore S. Novell, Xiangwen Li, Dianyong Zhang, and Md. Ashraful Alam Tuhin. HARPO/L: A language for hardware/software co-design. In *NECEC*, 2008.

[9] Jean D. Ichbiah, Bernd Krieg-Brueckner, Brian A. Wichmann, John G. P. Barnes, Olivier Roubine, and Jean-Claude Heliard. Rationale for the design of the ada programming language. *SIGPLAN Not.*, 14(6b):1–261, 1979.

[10] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 1999.

[11] John C. Reynolds. Toward a grainless semantics for shared-variable concurrency. In *FSTTCS*, pages 35–48, 2004.

[12] Theodore S. Norvell. A grainless semantics for the HARPO/L language. In *CCECE*, pages 810–814, 2009.

[13] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[14] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice-Hall Software Series. Prentice-Hall, Inc., Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[15] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgen Kaufmann Publishers, Inc., 2002.

[36] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, Micorsoft Corporation, One Microsoft Way, Redmond, WA 98052, May 2005. http://research.microsoft.com/pubs/70179/tr-2005-70.pdf.

[37] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Verifying Concurrent C Programs with VCC. A draft of VCC tutorial, 2010. http://research.microsoft.com:8081/en-us/um/people/moskal/pdf/vcc-tutorial-1col.pdf.

[38] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[39] Leslie Lamport. win and sin: predicate transformers for concurrency. *ACM Trans. Program. Lang. Syst.*, 12(3):396–428, 1990.

[40] Theodore S. Norvell. The Static Semantics of HARPO/L. [Draft], 2008.

[41] Jayadev Misra. A logic for concurrent programming. Technical report, Formal Aspects of Computing, 1994.