Model Checking as a Tool Used by Parallelizing Compilers*

Teodor Rus and Eric Van Wyk
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242 USA
rus,vanwyk@cs.uiowa.edu

Abstract

In this paper we describe the usage of temporal logic and model checking in a parallelizing compiler to analyze the structure of a source program and locate opportunities for optimization and parallelization. The source program is represented as a process graph *in which the nodes are sequential processes and the edges are control and data dependence relationships between the computations at the nodes.*

By labeling the nodes and edges with descriptive atomic propositions and by specifying the conditions necessary for optimizations and parallelizations as temporal logic formulas, we can use a model checker to locate nodes of the process graph where particular optimizations can be made. To discover opportunities for new optimizations or modify existing ones in this parallelizing compiler, we need only specify their conditions as temporal logic formulas. We do not need to add or modify the code of the compiler. This greatly simplifies the process of locating optimization and parallelization opportunities in the source program and makes it easier to experiment with complex optimizations. Hence, this methodology provides a convenient, concise, and formal framework to carry out program optimizations by compilers.

1 Introduction

There are many known optimizing and parallelizing program transformations [ABC+88, ASU86, CFS90, CHH89, HAM95, HMPT94, KP79, PEH95, Pol88, Wol96, ZC90] that can be applied by a compiler to significantly decrease the execution time of the program. These transformations depend on both the peculiarities of the machine architecture and on the semantic properties of the language constructs that undergo such transformations. However, there is no systematic way to determine the context in which these transformations can be performed while preserving the computation expressed by the transformed constructs. Hence, discovering where in the program such transformations can be made is a difficult problem. In most instances, the compiler represents the program as a graph which is searched for opportunities to make these transformations. Neither the graph nor the search are based on the source language specification rules and therefore they cannot lead to a systematic approach to discovering parallelism opportunities. Usually this searching process is performed by code written by the compiler implementer and is performed in various phases of the compilation process. Writing such code is time consuming and error prone and thus makes adding and modifying optimization and parallelization transformations difficult and may affect program portability. These difficulties prevent compiler developers from fully exploring the range of optimization and parallelization transformations.

To encourage compiler designers to experiment with varied and complex transformations, a methodology is needed that allows the properties of optimization and parallelization transformations of source language constructs to be expressed by formulas. These formulas are compositional semantic macro-operations expressing properties of the computation denoted by the source language construct in terms of the properties of the construct components. The semantic properties we use characterize optimization conditions and are parameterized temporal logic formulas where parameters are properties of the construct components. This methodology is obtained by (1) developing a logic that allows the expression of optimization properties and (2) attaching formulas expressing these properties to the language specification rules. In this proposal we describe such a methodology which allow the com-

*This work was partially supported by the grant NGT-51321 from the NASA Jet Propulsion Laboratory.

piler developer to write temporal logic formulas instead of computer code to find optimization and parallelization opportunities. This encourages more experimentation with advanced optimizations and parallelizations as the compiler developer is not burdened with modifying code which locates optimization opportunities, but must only modify the formulas which specify the conditions necessary for the optimization. This methodology also provides the foundation for an interactive dialog between the compiler and the programmer which allows the programmer to control the granularity of the sequential processes in the program and thus tune the performance of the parallel program.

The methodology presented in this paper relies on an extension to the branching time temporal logic called Computation Tree Logic, CTL, and its model checking algorithms [CES86]. Model checking is a formal verification technique often used to check the correctness of real-time and concurrent programs by writing the correctness specification as a temporal logic formula and using a model checker to determine which states in a graph representation of the program, called a model, satisfy the formula. Here, the compiler developer can specify the necessary conditions for a particular optimization or parallelization as a temporal logic formula and a model checker is then used to find, in a graph representation of the source program, the locations in the program which satisfy the formula and are thus candidates for the optimization.

The graph representation of the source program used in our compiler is a directed graph called a *process graph*. When the process graph is labeled with atomic propositions, it is referred to as a *process model*. The nodes of the model represent sequential processes discovered in the source program. These processes are the *units of computation* which are executed sequentially and are the components of a parallel program. The programmer can control the granularity of the sequential processes by specifying as the units of computation those types of language constructs desired to be executed sequentially. These nodes are labeled by atomic propositions representing fundamental properties of the computations at the nodes. The edges represent the control and data dependency relationship between the nodes. Both nodes and edges are labeled with descriptive propositions. We thus extend the temporal logic CTL and its satisfaction rules to allow formulas which reference these edge propositions. Temporal logic formulas written over these propositions allow the compiler developer to describe the conditions necessary for particular optimizations. Thus, during compilation, a model checking algorithm can be invoked to find the computations in the program that satisfy the conditions described by the optimization formula.

Specifying optimizations formally in a temporal logic provides a formal foundation for program optimizations. In addition, this technique promotes quick implementation of new program optimizations because a compiler is obtained by automatic integration of stand alone algorithms (scanner, parser, semantic generator, code generator) [Rus97] that are determined by the specification rules and allow independent development of the language lexicon, constructs, and types. The compiler developer writes formulas, not programs, to find optimizations and attaches them to the rules specifying the constructs involved in the optimization targeted by these formulas. Consequently this technique does not freeze the notation a compiler can handle thus allowing a dynamic development of the compiler within a dynamic problem domain. Therefore this technique can be used to improve the quality of the compiler, adapting the compiler dynamically to language changes while experimenting with new optimization strategies.

The fundamental property on which much of our research is developed results as a consequence of a different way of solving problems with a computer which departs from the Von Neumann program execution model with its *fetch, analyze, execute,* and *store* operation cycle. Rather than transforming a problem solving algorithm into a sequence of fetch, analyze, execute operation cycles we use universal constructs of algebra to develop universal algorithms over problem domains that map *problem expressions* into *problem solutions*. An example is the universal algorithm that evaluates an expression by extending a function defined on the free generators of the expression (taking values in the types of the generators) to a homomorphism that maps from the problem (the term algebra that contains the expression as a valid element) to the solution (the algebra of values where the term algebra is interpreted). Assuming that the term algebra is freely generated and finitely specified this universal algorithm can be parameterized by the specification rules of the problem expression algebra and can be adapted to all kinds of problems that involve expression evaluation, such as language translation, theorem proving, text-rewriting, etc. The essential ingredient in this methodology is the universal construct of algebra known as the *unique extension lemma* [BL69, Rus91], which is further used as follows:

- Use structural properties of the specification rules to identify valid stand-alone components of the objects (programs, statements, expressions, etc.) specified by these rules. We employ structural properties of BNF specification rules of the form

$A_0 = t_0 A_1 \ldots t_{n-1} A_n t_n$ where $n \geq 0$, $t_0, t_1, \ldots, t_n$ are fixed strings called terminals, and $A_1, \ldots, A_n$ are variables called non-terminals that stand for valid constructs specified by BNF rules of the form $A_i = t_0^i A_1^i \ldots t_{n_i-1}^i A_{n_i}^i t_{n_i}^i$, [Rus87, RH94].

- Provide evaluation mechanisms that when given the value of the objects w_i, $1 \leq i \leq n$, specified by the rules $A_i = t_0^i A_1^i \ldots t_{n_i-1}^i A_{n_i}^i t_{n_i}^i$, $1 \leq i \leq n$, construct the value of the object w specified by rule $A_0 = t_0 A_1 \ldots t_{n-1} A_n t_n$ using w_i, $1 \leq i \leq n$, as components. We use semantic macro-operations [RH84, Lee90, RVW96] for this purpose.

- Provide proof rules that allow formal correctness proofs of the synthesis process of the value of the object specified by the rule $A_0 = t_0 A_1 \ldots t_{n-1} A_n t_n$ from its components specified by the rules $A_i = t_0^i A_1^i \ldots t_{n_i-1}^i A_{n_i}^i t_{n_i}^i$, $1 \leq i \leq n$. We use homomorphisms, derived operations, and embeddings [BL69, Coh81, Rus91] for this purpose.

We use this methodology for the construction of algebraic compilers [Rus95], and for the automatic generation of model checking algorithms [RVW96, RVW97b]. Here we illustrate this principle by integrating a model checker algorithm into a compiler as a program parallelization tool. For that we develop a temporal logic that allows us to express properties of control flow and data dependencies of a program and propose a formal method of specifying locations where program optimizations can be made.

Section 2 of the paper describes the temporal logic we use to express program properties checked during program transformation by the compiler. Section 3 is devoted to the Kripke model we use as our program representation graph called process model. Here we specify the atomic propositions that we use to construct temporal logic formulas expressing program optimization and parallelization properties. Section 4 discusses, with examples, the mechanism of program parallelization by the compiler using temporal logic and model checking. Finally, section 5 discusses some previous results and introduces the reader to some of the future developments we are currently pursuing.

2 Temporal logic and model checking

Model checking is a formal verification technique used to validate the correctness of some system, be it a concurrent or real-time program or representation of a physical system. The system is represented by a model that describes how the state of the system changes in time. A Kripke structure or *model* $\mathcal{M} = <N, E, P : AP \rightarrow 2^N>$ is a directed graph with a finite set of nodes N, a finite set of edges E, and a proposition labeling function P which maps atomic propositions from the set AP to the set of nodes in N on which those propositions are true. A path in $\mathcal{M}$ is a sequence of nodes $n_0, n_1, \ldots$ such that $\forall i \geq 0$, $(n_i, n_{i+1}) \in E$.

A specification of the correctness of the system is written as a temporal logic formula over the propositions labeling the nodes of the model. Model checking is the problem of finding on which nodes, n, in a model $\mathcal{M}$ a temporal logic formula f is satisfied. In this paper we describe CTL, Computational Tree Logic, a branching time temporal logic, developed by Emerson, Clarke and Sistla in [CES86] and an extension called CTL^e [RVW97b].

CTL formulas are defined by the following rules:

1. $true$, $false$ and any atomic proposition $ap \in AP$ are CTL formulas.

2. if f_1 and f_2 are CTL formulas then $\neg f_1$, $f_1 \vee f_2$, and $f_1 \wedge f_2$ are CTL formulas.

3. if f_1 and f_2 are CTL formulas then AXf_1, EXf_1, $A[f_1 U f_2]$, and $E[f_1 U f_2]$ are CTL formulas.

The temporal operator X (next-time) is used in the formula AXf_1 (respectively, EXf_1) which is satisfied on a node if all (respectively, on one or more) successors satisfy f_1. The temporal operator U (until) is used in the formula $A[f_1 U f_2]$ (respectively, $E[f_1 U f_2]$) which is satisfied on a node if on all (respectively, on one or more) paths beginning on this node there is a node on which f_2 holds and f_1 hold on all intermediate nodes.

The formal rules that determine if a node n in a model $\mathcal{M}$ satisfies a formula f, denoted $\mathcal{M}, n \models f$ or $n \models f$ if $\mathcal{M}$ is assumed, are given below:

$n \models ap$ iff $n \in P(ap)$
$n \models \neg f$ iff not $n \models f$
$n \models f_1 \wedge f_2$ iff $n \models f_1$ and $n \models f_2$
$n \models f_1 \vee f_2$ iff $n \models f_1$ or $n \models f_2$
$n \models EXf_1$ iff $\exists m \in N[(n,m) \in E \wedge m \models f_1]$
$n \models AXf_1$ iff $\forall m \in N[(n,m) \in E \Rightarrow m \models f_1]$
$n \models A[f_1\ U\ f_2]$ iff $\forall$ *paths* $(n_0, n_1, \ldots)$ with $n = n_0$
$[\ \exists i\ [i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow n_j \models f_1]\ ]\ ]$
$n \models E[f_1\ U\ f_2]$ iff $\exists$ *a path* $(n_0, n_1, \ldots)$ with $n = n_0$
$[\ \exists i\ [i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow n_j \models f_1]\ ]\ ]$

We extend CTL so that propositions labeling the *edges* of the model can be used to quantify the paths examined in determining the satisfaction of temporal

logic formulas. This extension of CTL with edge-formulas, which we call CTL^e, has important applications in reasoning about dynamic systems (such as processes in parallel programs). We use this logic to reason about the parallel processes discovered in a sequential program by a parallelizing compiler.

To label the edges of a model with propositions, we extend the model to $\mathcal{M} = <N, E, P_n : AP_n \rightarrow 2^N, P_e : AP_e \rightarrow 2^E>$, where N is a finite set of nodes (as before), E is a finite set of edges, P_n maps node atomic propositions in AP_n to the set of nodes on which they hold, and P_e maps atomic edge propositions in AP_e to the set of edges on which they hold. Here, the model may be a multi-graph, that is, there may be more than one edge between the same two nodes. This is required in our application since there may be more that one dependency between the same two computations in the source program. If these computations are represented by distinct nodes, then the propositions describing these distinct dependencies must be kept on separate edges. An edge e from node s to node t, cannot be uniquely identified by the ordered pair (s,t), thus, we introduce the notation $\sigma(e)$ and $\tau(e)$ to identify respectively, the source and target of an edge e. For this reason, a path is redefined to be a sequence of nodes and edges $n_0, e_0, n_1, e_1, n_2, e_2, \ldots$ where $\forall i \geq 0, n_i \in N \wedge e_i \in E \wedge n_i = \sigma(e_i) \wedge n_{i+1} = \tau(e_i)$.

To allow path quantification, we extend the syntax and semantics of CTL to create CTL^e where we define edge formulas over the edge propositions constructed by the following rules:

1^e. $true$, $false$ and any atomic edge proposition $ap_e \in AP_e$ are CTL^e edge formulas.

2^e. if f_1 and f_2 are CTL^e edge formulas, so are $\neg f_1$, $f_1 \vee f_2$, and $f_1 \wedge f_2$.

If an edge $e \in E$ satisfies an edge formula f for a model $\mathcal{M}$ we write $\mathcal{M}, e \models f$ or $e \models f$. The formal rules that determine the satisfaction of edge formulas are:

$e \models ap_e$ iff $e \in P_e(ap_e)$
$e \models \neg f$ iff not $e \models f$
$e \models f_1 \wedge f_2$ iff $e \models f_1$ and $e \models f_2$
$e \models f_1 \vee f_2$ iff $e \models f_1$ or $e \models f_2$

To construct CTL^e formulas we use the same logical and temporal operators used in CTL. However, the temporal operators in CTL^e have as an additional argument the edge formula which describes the conditions that must be met on the edges of the paths traversed in determining the satisfaction of CTL^e formulas. To preserve the familiar notation of the CTL temporal operators, the edge formulas in CTL^e are written as subscripts to the CTL temporal operators rather than as the second arguments of these operators. Thus, although the arity of the CTL temporal operators changes when they are employed in CTL^e formulas, a familiar notation can still be used. CTL^e formulas are defined by the following rules:

1′. $true$, $false$ and any atomic node proposition $ap \in AP_n$ are CTL^e formulas.

2′. if f_1 and f_2 are CTL^e formulas then $\neg f_1$, $f_1 \vee f_2$, and $f_1 \wedge f_2$ are CTL^e formulas.

3′. if f_1 and f_2 are CTL^e formulas, and f_e is a CTL^e edge formula, then $AX_{\{f_e\}}f_1$, $EX_{\{f_e\}}f_1$, $A[f_1 U_{\{f_e\}} f_2]$, and $E[f_1 U_{\{f_e\}} f_2]$ are also CTL^e formulas.

The formula $AX_{\{f_e\}}f_1$ (respectively, $EX_{\{f_e\}}f_1$) is satisfied on a node if all (respectively, one or more) successors satisfy f_1 and the edges to these successors satisfy the edge formula f_e. The formula $A[f_1 U_{\{f_e\}} f_2]$ (respectively, $E[f_1 U_{\{f_e\}} f_2]$) is satisfied on a node if on all (respectively, on one or more) paths beginning on this node there is a node on which f_2 holds, f_1 holds on all nodes before this node, and each edge in the path before the node on which f_2 holds satisfies f_e.

The formal rules defining the satisfaction of the CTL^e formulas are:

$n \models ap$ iff $n \in P(ap)$
$n \models \neg f$ iff not $n \models f$
$n \models f_1 \wedge f_2$ iff $n \models f_1$ and $n \models f_2$
$n \models f_1 \vee f_2$ iff $n \models f_1$ or $n \models f_2$
$n \models EX_{\{f_e\}}f_1$ iff $\exists e \in E, n = \sigma(e) \wedge (e \models f_e \wedge \tau(e) \models f_1)$
$n \models AX_{\{f_e\}}f_1$ iff $\forall e \in E, n = \sigma(e) \Rightarrow (e \models f_e \wedge \tau(e) \models f_1)$
$n \models A[f_1\ U_{\{f_e\}}\ f_2]$ iff $\forall$ *paths* $(n_0, e_0, n_1, e_1, \ldots)$, $n = n_0$ and $\exists i[i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow (n_j \models f_1 \wedge e_j \models f_e)]\ ]$
$n \models E[f_1\ U_{\{f_e\}}\ f_2]$ iff $\exists$ *a path* $(n_0, e_0, n_1, e_1, \ldots)$, $n = n_0$ and $\exists i[i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow (n_j \models f_1 \wedge e_j \models f_e)]\ ]$

As an example, Figure 1 shows a model with six nodes, 1, 2, 3, 4, 5, 6, labeled with node propositions P, Q, and R and with edge propositions a, b, and c. State 1 satisfies $AX\ P$ but not $AX_{\{b\}}P$ since although all successors satisfy P, edge (1,4) does not satisfy b. State 1 also satisfies $E[P\ U_{\{b\}}R]$ by the path 1,3,6. State 2 satisfies $A[P\ U_{\{a\}}R]$ since edges in both paths 2,5,6 and 2,6 satisfy a, nodes 2 and 5 satisfy P and node 6 satisfies R.

Theorem: CTL and CTL^e have the same expressive power.

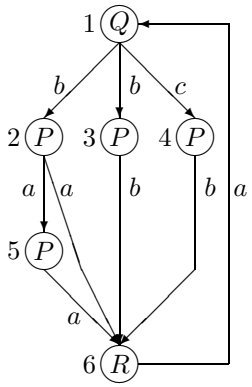

Figure 1. Example of a CTL^e model

Proof: Any CTL^e formula f and edge labeled model $\mathcal{M}$ can be restructured, respectively as a CTL formula f' and traditional model $\mathcal{M}'$ such that the set of nodes in $\mathcal{M}$ which satisfy f is the same as the set of nodes in $\mathcal{M}'$ which satisfy f'. The model $\mathcal{M}'$ is constructed from the model $\mathcal{M}$ by the procedure:

1. For each node $n \in N$ we construct a node $n' \in N'$ and label n' with the propositions labeling n.

2. For each edge $e_i \in E$ construct two edges, $e'_i, e''_i \in E'$ and a node $n'_i \in N'$ such that the source of e'_i is the source e_i, the target of e'_i is n'_i, the source of e''_i is n'_i, and the target of e''_i is the target of e_i. That is, $\sigma(e'_i) = \sigma(e_i)$, $\tau(e'_i) = n'_i$, $\sigma(e''_i) = n'_i$, and $\tau(e''_i) = \tau(e_i)$. Node n'_i is labeled with the propositions which label e_i and with the new proposition *edge*, to indicate that this node replaced an edge in the edge labeled model.

In the new model, the formula *edge* holds on all new nodes and the formula $\neg edge$ holds on all original nodes. A CTL^e formula f on the model $\mathcal{M}$ is an abbreviations for the CTL formula f' on the model $\mathcal{M}'$ obtained by the following rules:

1. $AX_{\{f_e\}}f$ is the abbreviation of $AX((f_e \wedge edge) \wedge AX(f \wedge \neg edge))$

2. $EX_{\{f_e\}}f$ is the abbreviation of $EX((f_e \wedge edge) \wedge EX(f \wedge \neg edge))$

3. $A[f_1 U_{\{f_e\}} f_2]$ is the abbreviation of $A[((f_1 \wedge \neg edge) \vee (f_e \wedge edge))U f_2 \wedge \neg edge)]$

4. $E[f_1 U_{\{f_e\}} f_2]$ is the abbreviation of $E[((f_1 \wedge \neg edge) \vee (f_e \wedge edge))U(f_2 \wedge \neg edge)]$

It is easy to show that $\mathcal{M}, n \models f$ iff $\mathcal{M}', n \models f'$, where f is the abbreviation of f', q.e.d.

One of the major goals in our compiler development methodology is to make the process of compiler development as simple as possible. Hence, despite the fact that CTL and CTL^e have the same expressive power, CTL^e is required due to the complexity of the CTL models and formulas expressing paths properties. In addition, CTL^e formulas express more naturally the optimization and parallelization properties. This motivates our use of CTL^e and its model checker in the compiler.

3 A program abstraction model

The *process graph* is the intermediate representation of the source program used by our parallelizing compiler [RVW97a]. When atomic propositions label the nodes and the edges, we view this graph as a Kripke structure, and call it the *process model*. Thus, we can use a model checker and CTL^e formulas to analyze the source program. Here we describe the methodology we use to construct the process model we use to identify optimization and parallelization opportunities.

A process graph is a directed graph in which nodes represent units of computation found in a source program by a parallelizing compiler and edges represent control and data dependency relationships between these computations. The process graph is not a traditional "control flow graph" whose edges direct the flow of control from one computation to the next. The nodes of a process graph represent sequential processes, i.e., stand alone computations; the edges represent the minimal restrictions on the execution order of the computations represented by the nodes required to ensure a correct execution of the computation represented by the graph.

To determine how a source program is broken into the pieces that are represented by the nodes of a process graph we define *units of computation* to be types of computations that the programmer chooses to be executed sequentially. Thus, constructs in the source program that are not units of computation are either too lightweight to stand as individual processes and must be combined with other program constructs, or have as their components units of computation and are therefore represented as *graphs* whose nodes are units of computation and edges are control and data dependencies between the nodes of their component graphs. For example, if *assignment statements* are units of computation, then an assignment statement in a source program will be represented as a node, whereas the expression on its right hand side is too lightweight to be a process; a sequence of assignment statements is represented as a collection of nodes each representing the

individual assignment statements and edges representing the control and data dependencies between them. Since computations are represented by language constructs, the rules specifying valid language constructs allow us to provide the following formal definition of the unit of computation: *a unit of computation is any valid construct recognized by one of the rules marked by the programmer as specifying units of computation.* That is, the programmer specifies which rules generate units of computation, and the compiler in turn generates sequential code from any construct recognized by these rules. By allowing the programmer to specify which constructs will be defined as units of computation the programmer can control the granularity of the processes executing the parallel program generated by the compiler.

Once a computation from the source program has been identified as a unit of computation, it is represented as a node in the process graph. A node, n, is a tuple $<Types, State, Transition>$ where:

- $Types$ is the set of types of the variables and constants used in the computation;
- $State$ is the tuple $<V, \upsilon: V \rightarrow Types>$, where V is the set of variables in the computation. Since during the compilation process the values of the variables in V may not be known, υ maps a variable to its type instead of mapping it to its value. This provides an approximation to the variable value which used in the abstract interpretation of the program [CC77].
- $Transition: \upsilon \rightarrow \upsilon'$ implements the computation by mapping υ, the values of the the variables V before the computation, to υ', the values of the variables in V after the computation.

By $V_W(n)$ and $V_R(n)$ we denote the sets of variables which are written (i.e., defined) and read (i.e., used) in the computation $\upsilon \rightarrow \upsilon'$ on node n, respectively. Thus, if *assignment statement* has been specified as a unit of computation, an assignment statement `x = i * y` where `x` and `y` are real variables and `i` is an integer variable would be represented by the node

$$<\{Real, Integer\}, <\{x, y, i\}, \upsilon = \{x \rightarrow Real, y \rightarrow Real, i \rightarrow Integer\}>, \{x' = i * y, y' = y, i' = i\}>$$

where x', y' and i' represent the new values of the variables after the computation. The value of υ is specified at compilation time using universal constants (that can be chosen to be the type names) and is preserved by program execution.

Each time a graph is constructed, two special nodes, denoted e and x, are created which perform no computation but stand for the *entry* and the *exit* points of the computation represented by the graph. Hence, when a unit of computation is discovered and its computation is represented by a single node n, the graph for this computation contains three nodes, e, n, x, and two edges, $e \rightarrow n$ and $n \rightarrow x$, i.e., this graph has the shape $e \rightarrow n \rightarrow x$. The $e \rightarrow n$ and $n \rightarrow x$ edges are *control* dependency edges since e initiates the execution of n and n must complete before x.

Each node in the process graph is also labeled with descriptive atomic propositions to create the process model. All unit of computation nodes are labeled by the proposition *unit*, the entry node is labeled with proposition e, and the exit node is labeled with proposition x.

In the construction of graphs from nodes and other graphs we restrict our presentation in this paper to four graph compositions, *functional*, *branching*, *enumerated repetition*, and *conditional repetition*. These, respectively, correspond to the source language composition operations *sequential composition*, *if-then-else*, *do loop* and *while loop* of the sample source language in this paper. A graph composition of component graphs is performed by a parallelizing compiler as the corresponding source language construct is recognized by a parser. For example, when a do loop is discovered by the parser in the source program, a graph is created with the enumerated repetition graph composition operator using as components the graph previously constructed for the loop body and information from the loop used to label a loop header node. The composition is expressed by setting precedence relationships (control and data dependencies) between the nodes in the component graphs and is constructed as *labeled edges between the nodes*. Each source language composition may define a relationship, $\prec$, between two nodes n_1 and n_2, denoted, $n_1 \prec n_2$, that indicates that n_1 would execute before n_2 in a sequential execution of the program. Assuming that in a particular program with nodes n_1 and n_2 an instance of n_1 may execute before an instance of n_2, then we have:

1. The computation represented by the node n_2 is *data flow dependent* on the computation represented by the node n_1, $n_1 \stackrel{f}{\rightarrow} n_2$, if $V_W(n_1) \cap V_R(n_2) \neq \emptyset$.
2. The computation represented by the node n_2 is *data anti dependent* on the computation represented by the node n_1, $n_1 \stackrel{a}{\rightarrow} n_2$, if $V_R(n_1) \cap V_W(n_2) \neq \emptyset$.
3. The computation represented by the node n_2 is *data output dependent* on the computation represented by the node n_1, $n_1 \stackrel{o}{\rightarrow} n_2$, if $V_W(n_1) \cap V_W(n_2) \neq \emptyset$.

The consistency of the computations at n_1 and n_2 requires that if $n_1 \xrightarrow{d} n_2$, $d \in \{f, a, o\}$, then an instance of the computation at n_1 executes before an instance of the computation at n_2.

The data dependency edges are labeled also by propositions describing the *distance* of the data dependency [Ban88]. The distance of a data dependency between two nodes n_1 and n_2 (which may be the same node) is defined for each loop ℓ enclosing both nodes as the number of iterations of loop ℓ between the execution of n_1 and the subsequent execution of n_2 causing the data dependency. That is, if during iteration i of loop ℓ, n_1 writes a value to a variable which is read by n_2 in iteration j, $j \geq i$, then the data dependency distance over loop ℓ is $j - i$. For example, if during the iteration i of ℓ, n_1 writes a value to an array element that is read in n_2 on the iteration $i + 1$ of ℓ, the data dependency distance is 1 since $i + 1 - i = 1$, i.e., there was one iteration between the instances of n_1 and n_2 that caused the dependency. In the general case, the distance of a data dependency may not be a constant value, but may change for different instances of n_1 and n_2; this is a *non-constant distance* data dependency. For example, in a loop with index variable I, a writing array reference $A[2 * I]$ on node n_1 and a reading array reference $A[I]$ on n_2 (both enclosed in ℓ) causes a non-constance data dependency since a data flow dependency exists between iterations 1 and 2, 2 and 4, 3 and 6, etc. Data dependencies that may have a distance of 1 or more for an enclosing loop ℓ are called *loop carried* dependencies or dependencies *carried by loop* ℓ. The data dependency distances are computed by solving a set of linear equations derived from the index expressions used in the loop body ℓ [Ban88]. Data dependencies between nodes that are not enclosed in the same loop structure are called *loop crossing* data dependencies, because they cross the boundary of a loop.

In the process model data dependency edges are labeled with propositions describing the data dependency distance for each enclosing *enumeration loop*. Each proposition is indexed by the enclosing loop label and by a distance category. That is, for a loop ℓ, a data dependency is labeled with $D_{\{\ell,0\}}$, $D_{\{\ell,+\}}$, $D_{\{\ell,*\}}$, or $D_{\{\ell,\times\}}$ to indicate respectively a zero distance data dependency, positive distance data dependency, non-constant or unknown distance data dependency, and loop crossing data dependency.

The simplest graph construction results from the functional composition of two unit of computation nodes, n_1 and n_2. This graph will consist of four nodes: e, n_1, n_2, and x, and control dependency edges $e \rightarrow n_1$, $e \rightarrow n_2$, $n_1 \rightarrow x$, and $n_2 \rightarrow x$. We refer to this as functional composition since the computation computed by the graph is the functional composition of the transition functions on nodes n_1 and n_2. Formally, if g_1 and g_2 are process graphs then $g_1 \oplus g_2$ is a the graph representing the functional composition of the computations represented by g_1 and g_2 and has the shape in Figure 2. If there is a data dependency (flow, anti, or output) from a node n_1 of g_1 to a node n_2 of g_2 requiring the sequential execution of some component nodes, a data dependency edge is added, $n_1 \xrightarrow{d} n_2$, $d \in \{f, a, o\}$, which ensures the correct execution order of the computations at the nodes. If there were no such data dependencies, there would be no edges between g_1 and g_2, indicating that they could be executed in parallel. The graph representing the functional composition $g_1 \oplus g_2$ has no control flow edges from g_1 to g_2 since they are not necessary. The data dependency edges ensure the correct execution order or there is no data dependency and the computations at the nodes of g_1 and g_2 can be executed concurrently. This allows us to keep a minimal set of edges representing restrictions on the process execution order.

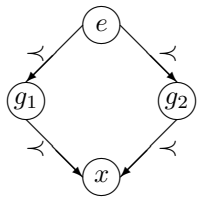

Figure 2. Functional composition

The branch composition of two graphs g_1 and g_2 and a predicate p is denoted by $Br(p, g_1, g_2)$ and has the shape in Figure 3, where *true* and *false* are control dependencies.

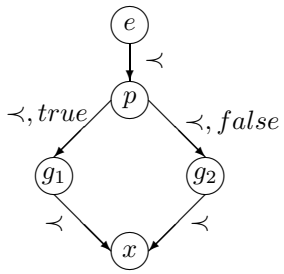

Figure 3. Branch composition

An enumeration loop represents the repeated execution of a computation body with different values associated with a variable called the loop index. The

operator that constructs a loop from a loop index and a loop body is denoted by $Lp(h[i], g[i])$ where i is the loop index, $h[i]$ is the graph representing the loop index range, and $g[i]$ is the computation performed by the body. A loop can also be seen as the repeated functional composition of the loop body, each repetition executing with a new value of the loop index. That is, $Lp(h[i], g[i]) = (h[l] \oplus g[l]) \oplus \ldots \oplus (h[u] \oplus g[u])$ where $i \in [l..u]$ is the range of values taken by i. The copies of the nodes in the body of the loop could then be labeled with their value of the loop index variable. We could then add all the appropriate data dependency edges between nodes in the loop body instantiations to ensure the correct execution order of the nodes and allow the loop header node to initiate the execution of all iterations at once. Thus all copies of the loop body could execute in parallel with the appropriate restrictions resulting from the data dependency between them. However, representing loops in this way is clearly not practical because the size of the process graph grows exponentially in the depth of the largest loop nest. In addition, because the range of the loop index may not be known at compile time, this representation is not always possible. Therefore, the graph $Lp(h[i], g[i])$ has the shape in Figure 4 where $h \xrightarrow{i:l..u} g$ represents the instantiation of *all* iterations of the loop body. This requires that we add data dependency edges to ensure correct execution order of the loop iterations. Note that the lack of control dependency edges between nodes of g is reminiscent of the lack of control dependency edges between nodes in a functional composition. In both cases they are not necessary.

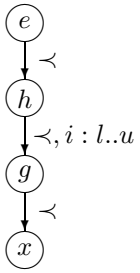

Figure 4. Enumerated repetition composition

The *conditional repetition* composition corresponds to the while loop construct. Its operator $CondLp(p, g)$ has as components a predicate p, and a loop body graph g. The resulting graph has the shape shown in Figure 5. The predicate and edges emanating from it are similar to those in the branch composition and are labeled with $true$ and $false$ indicating the conditional control dependencies. We also add control dependencies $\prec$ from each predecessor of the exit node of g to the predicate node of the conditional loop. These ensure a sequential execution of the while loop. Because of the sequential nature of the conditional loop and the $\prec$ edges, we do not concern ourselves with data dependencies carried by a conditional loop. In many cases it is possible to transform a conditional loop into an enumerated loop, although these are not examined in this paper.

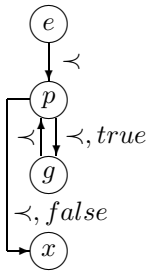

Figure 5. Conditional repetition composition

Our process graphs and process models are significantly different from most program flow graphs found in optimizing and parallelizing compilers [GP92, Pol88, PW86, SA88, Sar89, SG91, SMD+89]. First, nodes do not represent *basic blocks* - segments of target code that contain no branching statements. The nodes of a process graph represent source language constructs and their creation is controlled by the programmer by defining the set of *units of computation*. This allows the programmer to control the granularity of the sequential processes executing the parallel program. We also provide descriptive data dependency edges so that the control dependencies implicitly provided by the textual layout of the computation are irrelevant to the execution order of the statements in the program. Process graphs are, however, very similar to the Kripke structures used in model checking. By representing some of the information labeling the nodes and edges in the process graph as atomic propositions, the projections of our graphs containing the nodes, edges, and atomic propositions, which we call process models, are Kripke structures. Thus, many of the questions that optimizing and parallelizing compilers ask about programs can be represented as temporal logic formulas. These *questions* can then be *answered* by a model checking algorithm.

The propositions which label nodes and their meanings are listed below:

ℓ_n	- unique label for node n
e	- *entry* nodes
x	- *exit* nodes

$unit$	- unit of computation nodes
$func$	- functional composition nodes
$branch$	- branch condition nodes
$enum$	- enumerated loop header nodes
$cond$	- conditional loop header nodes

The propositions which label edges and their meanings are listed below:

$\prec$	- control dependency
$true$	- $true$ branch control dependency
$false$	- $false$ branch control dependency
$enum$	- $enum$ control dependency
d	- data dependency
f	- data flow dependency
a	- data anti dependency
o	- data output dependency
$V_{\texttt{var}}$	- data dependency on variable `var`
$D_{\ell,0}$	- constant distance data dependency with distance 0 for $enum$ loop node ℓ
$D_{\ell,+}$	- constant distance data dependency with positive distance for $enum$ loop node ℓ
$D_{\ell,*}$	- non constant or unknown distance data dependency for $enum$ loop ℓ
$D_{\ell,\times}$	- data dependency that crosses loop boundary ℓ

4 Discovering optimization opportunities

During construction of the process graph and process model we use a model checker to answer questions about the source program structure posed in the form of a CTL^e formula. As the parser recognizes source language constructs, it invokes a graph macro processor which constructs the process graph and process model which correspond to the source language construct discovered. This graph is constructed from the process graphs corresponding to the components of the source language construct. During the process of compilation, the model checker is used to answer questions about the construct discovered by the parser. For example, when a *do* loop is discovered, the compiler may ask if the loop iterations are independent and can thus be executed concurrently. This is done by constructing the process model for the loop and checking if the loop header node satisfies a CTL^e formula that describes the conditions necessary for the concurrent execution of the iterations of the loop body. We illustrate this technique by testing for loop iteration independence and scalar expansion with a few simple examples including the Dirichlet example [CT92].

One fundamental question a parallelizing compiler will ask is if the iterations of an enumerated loop (do loop) are independent and can thus be executed concurrently. We consider this question first for the simple loop in Figure 6. An enumerated loop has independent

```
ℓ1: do i = 1, 100
ℓ2:     a[i] = b[i] * c[i]
ℓ3:     d[i] = a[i] * d[i]
    end do
```

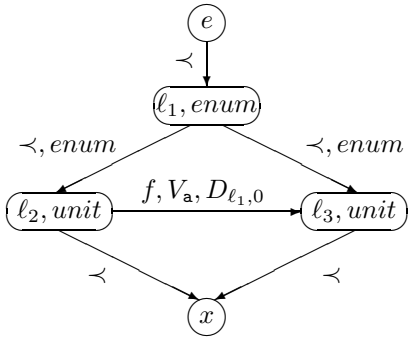

Figure 6. Process graph of a Fortran do loop

iterations if there are no loop carried data dependencies between the nodes representing the computations of the loop body. An enumerated loop, with loop header node labeled ℓ whose loop body does not contain any loops or branching statements has independent iterations if the node ℓ is an enumerated loop header, i.e., labeled by $enum$, and every successor reachable by an $enum$ labeled edge does not have any successors reachable by a loop carried data dependency edge labeled with $D_{\ell,+}$ or $D_{\ell,*}$. These requirements can be stated as a CTL^e formula allowing us to determine that loop ℓ_1 has independent iterations if

$$\ell_1 \models enum \wedge AX_{\{enum\}} \neg EX_{\{D_{\ell,+} \vee D_{\ell,*}\}} true$$

The model checker would report that the node ℓ_1 does in fact satisfy this CTL^e formula. Even though there is a data flow dependency from ℓ_2 to ℓ_3, it is not loop carried. On the other hand, if we were to replace the assignment statement ℓ_3 with `d[i+1] = a[i] * d[i]` a loop carried data dependency edge would be added from ℓ_3 to ℓ_3 with propositions $f, V_{\texttt{d}}, D_{\ell_1,+}$ which would cause the iterations of ℓ_1 to not be independent and ℓ_1 to not satisfy the formula.

With the original assignment statement on ℓ_3 we would still need to verify that this loop does not contain any loop or branching statements. Knowing that only

unit of computation nodes are labeled with *unit*, we can infer that this is the case if

$$\ell_1 \models AX\{enum\}\ unit.$$

We can combine these two conditions into a single CTL^e formula such that if ℓ_1 satisfies

$$enum \wedge AX_{\{enum\}}(unit\ \wedge\ \neg EX_{\{D_{\ell_1,+}\ \vee\ D_{\ell_1,*}\}} true)$$

then ℓ_1 represents an enumerated loop with independent iterations.

We use in this discussion (and will continue to do so) temporal logic formulas with atomic propositions specific to the examples we examine. However, in the compiler, the temporal logic formulas must be specified so that they are applicable for all language constructs that can be potential subject to optimization transformations. Therefore, attached to the specification rules are temporal logic macro-operations which are expanded by the compiler into the actual formulas that are checked against the model. The expansion of these macro-operations takes place when the source language constructs (loops in this case) are recognized by the parser using the corresponding specification rules.

The test above for loop iteration independence is of course too restrictive in that if it was our only test for iteration independence our compiler would not parallelize many loops. As a case study in this paper, we

```
ℓ1 :while (not converged(S))
ℓ2 :  do i = 1, N
ℓ3 :   do i = 1, N
ℓ4 :    sum=S[i,j-1]+S[i+1,j]
            +S[i,j+1]+S[i-1,j]
ℓ5 :    Next_S[i,j] = sum / 4
       end do
      end do
ℓ6 :  do i = 1, N
ℓ7 :   do i = 1, N
ℓ8 :    S[i,j] = Next_S[i,j]
       end do
      end do
    end while
```

Figure 7. Dirichlet Problem Code

examine the Dirichlet problem. A sample implementation of the problem is given in a simple imperative programming language in Figure 7. The corresponding process model, using the atomic propositions described above, is shown in Figure 8.

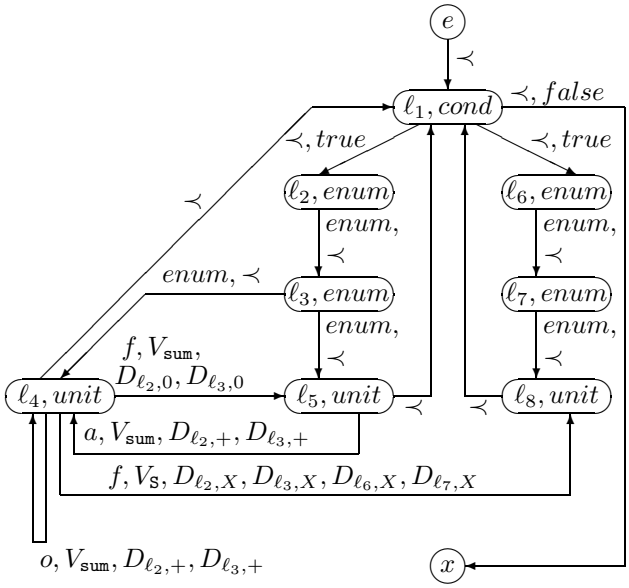

Figure 8. Dirichlet Process Model

To develop a more general iteration independence test, we must handle loops that contain other loops and branches. In the Dirichlet code, the enumerated loop labeled ℓ_6 has no loop carried dependencies and thus could execute its iterations concurrently, however, it fails our previous iteration independence test. If we re-examine this test, we realize that we really need to consider possible loop carried data dependencies from any unit of computation node in the graph of the loop body, not just those that are immediate successors of the loop header node as we did in the first version of the independence test. Since any unit of computation inside a loop or branch is reachable from the construct header or predicate node by a path of edges labeled by control dependency propositions, we can rewrite the formula to test if loop ℓ has independent iterations as

$$\ell \models A[true\ U_{\{\prec\}}(unit\ \wedge\ \neg EX_{\{D_{\ell,+}\ \vee\ D_{\ell,*}\}} true)]$$

This formula holds on node ℓ if all paths from ℓ are labeled by control dependency propositions $\prec$ and there is eventually a *unit* node from which there are no loop carried data dependencies.

Loops at nodes ℓ_6 and ℓ_7 satisfy this formula and thus have independent iterations. The loops at ℓ_2 and ℓ_3 do not satisfy this formula because of the scalar variable `sum` and the loop carried anti and output dependencies caused by `sum`. While this scalar is clearly unnecessary and would be removed by most sequential optimizers, we leave it in for demonstration purposes.

A scalar variable can often be *expanded* into an array such that the output of the program does not change

but data dependencies on the scalar disappear. In general, if a scalar, `s`, is written before it is read in all iterations of an enclosing loop ℓ with index variable `i`, it can be expanded into an array indexed by the loop index variable, i.e., `s[i]`. We say that the variable `s` is *expanded over loop* ℓ. The semantics of the loop do not change, and the loop carried data dependencies disappear since each iteration of the loop works on its own array element which replaces the scalar. Scalar expansion over a loop ℓ is possible if there are no loop ℓ carried data *flow* dependencies on the scalar. (For simplicity, we assume the scalar is not used elsewhere in the code. This assumption can of course be removed by modifying the CTL^e formula below.) The presence of a loop ℓ carried data flow dependency would indicate that the scalar is written in one iteration and read in a later iteration thus preventing the scalar expansion. We can again write these requirements as a CTL^e formula. That is, a scalar `s` can be expanded over loop ℓ if

$$\ell \models A[true\ U_{\{\prec\}}\ (unit \wedge \neg EX_{\{f \wedge V_{\mathtt{s}} \wedge (D_{\ell,+} \vee D_{\ell,*}\}} true)]$$

This formula is similar in structure to the previous iteration independence test and simply ensures that each *unit* node in the body of the loop ℓ does not have an ℓ loop carried data flow dependency on variable `s`.

Returning to our example, we notice that both nodes ℓ_2 and ℓ_3 satisfy the formula

$$\ell \models A[true\ U_{\{\prec\}}\ (unit \wedge \neg EX_{\{f \wedge V_{\mathtt{sum}} \wedge (D_{\ell,+} \vee D_{\ell,*}\}} true)]$$

where ℓ is either ℓ_2 or ℓ_3. Thus, the scalar variable could be expanded over either loop. If we first expand `sum` over the inner loop ℓ_3, we will remove the data dependencies for this loop, but not for the enclosing loop ℓ_2. `sum` would be expanded to `sum[j]` and thus there would be no data dependencies carried by loop ℓ_3, but ℓ_2 data dependencies still exist. There is no reason to consider only expanding scalar variables and not array variables. We can, in fact, expand `sum[j]` over loop ℓ_2 to get `sum[i,j]`. To test for array expansion we can use the same CTL^e formula we used for scalar expansion. Thus, after expanding `sum` over loops ℓ_3 and then ℓ_2, there would be no loop carried dependencies in ℓ_2 or ℓ_3, and both would pass the above iteration independence test. The only remaining data dependency is the loop crossing data anti dependency on `S` from node ℓ_4 to ℓ_8. Therefore the loops ℓ_2 and ℓ_6 can not execute concurrently with each other.

By using these CTL^e tests, we have verified that loops ℓ_6 and ℓ_7 can be parallelized and we have modified loops ℓ_2 and ℓ_3 so that they to can also be parallelized. Each analysis requires no special programming; just writing the CTL^e formulas defining the necessary conditions for the optimization. Thus, the compiler developer has much more freedom to experiment with different optimization and parallelization requirements. While the formulas we have written here are certainly too restrictive in that they may reject loops for parallelization that could be parallelized, it is the process of writing formulas to detect optimization and parallelization opportunities that we want to emphasize. To find more parallelizable loops, we need only modify the CTL^e formulas that detect them. We do not modify code to detect them.

5 Conclusions and comments

We have successfully integrated a model checker into our parallelizing compiler to locate opportunities for optimization and parallelization in programs written in a simple imperative language. The success of this project has prompted us to begin work on the general technology to be incorporated into a Fortran 90 compiler.

We developed the CTL and CTL^e model checkers used in the parallelizing compiler in the framework of an algebraic compiler which implements the model checker as a *language translator*, $\mathcal{MC}$, whose source language is the language of the CTL or CTL^e formulas, and the target language is the language of sets of nodes of the model [RVW96, RVW97b]. The model checker then maps the source expression of a CTL formula f, into the set of nodes on which f is satisfied. That is, $\mathcal{MC}(f) = \{n | n \in N \wedge \mathcal{M}, n \models f\}$. A significant advantage of this approach is that the model checker is automatically generated from the algebraic specifications of the source and target languages, thus, its correctness is assured. Therefore the generated model checkers are applicable to the verification of critical systems. Also, since the algebraic compiler methodology is based on homomorphism computation, the generated model checker algorithm is naturally parallel [RVW96], [Kna94].

By specifying a model checker in this algebraic framework, it is also very easy to extend the temporal logic and model checker as the problem domain expands. We found that it was necessary to label the edges of the process model with propositions to make decisions about optimization and parallelization opportunities. Thus, we extended the CTL logic to include edge formulas. Since the model checker was implemented in this algebraic framework, we needed to only extend the specification of the source and target algebras to generate a new model checker [RVW97b]. Hence, we can view the logic and model

checker as evolving dynamic systems. These results are also described on our World Wide Web page at `http://www.cs.uiowa.edu/~vanwyk/TICS/`.

Our results so far are promising, but there is still much to be done in this area. Besides the Fortran90 compiler, we are also experimenting with different methods of expressing the temporal logic formulas to make them easier to read and write.

There is also a continual effort to improve the formulas to find and exploit more parallelism in the source program. Formulas which identify induction variables and conditional loops which may be transformed into enumerated loop can be written. Of course new problems may require the extension of the temporal logic. In anticipation of this possibility, we have investigated extending the logic to include the *past* temporal operators. Again, given that the model checkers are developed in the algebraic framework, this extension is straight forward.

In closing we observe that there is much research to be done in this area, and we feel that there are many possibilities to improve the methodology and quality of parallelizing compilers through the use of these formal methods.

References

[ABC+88] F.E. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the ptran analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(October):617–640, 1988.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Ban88] U Banerjee. An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, 2(2):133–150, 1988.

[BL69] R.M. Burstall and P.J. Landin. Pprograms and their proofs: an algebraic approach. *Machine Intelligence*, 4:17–43, 1969.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proceedings of the 4th ACM Symposium on Programming Languages, POPL*, pages 238–252, January 1977.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CFS90] R. Cytron, J. Ferrante, and V. Sarkar. Experiences using control dependence in ptran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[CHH89] R. Cytron, M. Hind, and W. Hsieh. Automatic generation of dag parallelism. *ACM SIGPLAN Notices*, 25(6):54–64, June 1989. Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation.

[Coh81] P.M. Cohn. *Universal Algebra*. Reidel, London, 1981.

[CT92] K.M. Chandy and S. Taylor. *Introduction to Parallel Programming*. Jones and Bartlett Publishers, Boston, 1992.

[GP92] M. Girkar and C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, 1992.

[HAM95] S. Haridi, K. Ali, and P. Magnuson, editors. *EURO–PAR'95 Parallel Processing*, volume 966 of *Lecture Notes in Computer Science*. Springer–Verlag, 1995.

[HMPT94] C. Halatsis, D. Maritsas, G. Philokyprou, and S Theodoridis, editors. *PARLE'94 Parallel Architectures and Languages Europe*, volume 817 of *Lecture Notes in Computer Science*. Springer–Verlag, 1994.

[Kna94] J.L. Knaack. *An Algebraic Approach to Language Translation*. PhD thesis, The University of Iowa, Department of Computer Science, Iowa City, IA 52242, December 1994.

[KP79] D.J. Kuck and D.A. Padua. High-speed multiprocessors and their compilers. In *Proceedings of the 1979 International Conference on Parallel Processing*, pages 5–16, August 1979.

[Lee90] J.C. Lee. Macro-processors as compiler code generators. Master's thesis, The University of Iowa, Department of Computer Science, Iowa City, IA 52242, 1990.

[PEH95] D.A. Padua, R. Eigenmann, and J.P. Hoeflinger. Automatic program restructuring for parallel computing and the polaris fortran translator. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 647–649, San Francisco, Ca, February 15-17 1995.

[Pol88] C.D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[RH84] T. Rus and F. Her. An algebraic directed compiler generator. Technical Report 84-02, The University of Iowa, Department of Computer Science, Iowa City, Iowa 52242, 1984.

[RH94] T. Rus and T. Halverson. Algebraic tools for language processing. *Computer Languages*, 20(4):213–238, 1994.

[Rus87] T. Rus. An algebraic model for programming languages. *Computer Languages*, 12(3/4):173–195, 1987.

[Rus91] T. Rus. Algebraic construction of compilers. *Theoretical Computer Science*, 90:271–308, 1991.

[Rus95] T. Rus. Algebraic processing of programming languages. In A. Nijholt, G. Scollo, and R. Steetskamp, editors, *Twente Workshop on Language Technology*, pages 1–42, University of Twente, Enschede, The Netherlands, 1995.

[Rus97] T. Rus. Algebraic processing of programming languages. *Theoretical Computer Science*, 1997. Paper accepted for publication.

[RVW96] T. Rus and E. Van Wyk. Algebraic implementation of model checking algorithms. In *Third AMAST Workshop on Real-Time Systems, Proceedings*, pages 267–279, March 6 1996. Available at http://www.cs.uiowa.edu/~rus.

[RVW97a] T. Rus and E. Van Wyk. A formal approach to parallelizing compilers. In *SIAM Conference on Parallel Processing for Scientific Computation, Proceedings*, March 14 1997. Paper available at http://www.cs.uiowa.edu/~rus.

[RVW97b] T. Rus and E. Van Wyk. Integrating temporal logics and model checking algorithms. In *Fourth AMAST Workshop on Real-Time Systems, Proceedings*, May 21 1997. Paper available at http://www.cs.uiowa.edu/~rus.

[SA88] K. Smith and W.F. Appelbe. Pat–an interactive fortran parallelizing assistant tool. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 58–62, 1988.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989.

[SG91] B. Shei and D. Gannon. Sigmacs: A programmable programming environment. In A. Nicolau, D. Gelernter, T. Gross, and D. D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 88–108. The MIT Press, 1991.

[SMD+89] K. Sridharan, M. McShea, C. Denton, B. Eventoff, J.C. Browne, P. Newton, M. Ellis, D. Grossbard, T. Wise, and D. Clemmer. An environment for parallel structuring of fortran programs. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 2, pages 98–105, 1989.

[Wol96] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, California, 1996.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.