Integrating Temporal Logics and Model Checking Algorithms*

Teodor Rus and Eric Van Wyk

Department of Computer Science
The University of Iowa
Iowa City, Iowa USA

Abstract. Temporal logic and model checking algorithms are often used for checking system properties in various environments. The diversity of systems and environments implies a diversity of logics and algorithms. But there are no tools to aid the logician or practitioner in the experimentation with different varieties of temporal logics and model checkers. Such tools could give users the ability to modify and extend a temporal logic and model checker as their problem domain changes. We have developed a set of tools that provide these capabilities by placing the model checking problem in an algebraic framework. These tools provide a temporal logic test bed that allows for quick prototyping and easy extension to logics and model checkers. Here we discuss the usage of these tools to generate model checker algorithms as algebraic mappings (i.e., embeddings of one algebra into another algebra by derived operations) with the temporal logic as the source algebra and the sets of nodes of a model as the target algebra. We demonstrate these tools by extending CTL and its model checker by introducing formulas that quantify the paths over which the satisfaction of the temporal operators is defined. This is made possible by permitting propositions to label the edges as well as the nodes in the model. We use this logic and its model checker to analyze program process graphs during the parallelization phase of an algebraic compiler.

1 Introduction

There is currently much research in the area of temporal logic and model checking. It seems only reasonable that logicians and practitioners should have tools that allow them to experiment with various temporal logic notations and semantics. However, there are no tools that provide this capability. We propose a methodology which permits users to easily implement and modify temporal logics and model checking algorithms thus allowing them to change and extend the notation and semantics of their own temporal logics. This methodology is conducive to generating tools that foster an interactive experimentation with temporal logic. We demonstrate this by extending CTL and its model checker

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

by introducing formulas that express the quantification of paths for the temporal operators based on propositions labeling the edges in the model. This model checker is used by a parallelizing algebraic compiler to analyze program process graphs of the parallelized program.

Model checking is a formal technique used to verify that a system, be it a concurrent or real-time program or representation of a physical system, satisfies a given specification. The system is represented by a *model* that describes how the system's state changes over time. This model, (usually a Kripke structure [Kri63]), is a directed graph whose nodes are labeled with logical propositions describing various properties satisfied by the nodes of the system and whose edges describe the possible transitions between nodes. The specification of a property to be verified against a given model is written as a temporal logic formula over the propositions labeling the nodes of the model. Model checking is the problem of determining which nodes in a model $\mathcal{M}$ satisfy a temporal logic formula f. In this paper we examine the Computational Tree Logic, CTL, a branching time temporal logic [CES86]. We also extend this logic to CTL^e, a logic whose formulas are checked against models with propositions labeling both the edges and the nodes in the model.

The traditional implementation of a model checking algorithm is not easily extended to other temporal logics because the code needs to be re-designed and re-written. The model checker algorithm implemented as an algebraic embedding of the source algebra of CTL formulas into the target algebra of sets of nodes in the model is completely defined by the finite specifications of these algebras. Therefore, we are able to easily implement and extend temporal logics and model checkers simply by changing the specifications of the source and target algebras and allowing the tools to generate the model checker implementation from these specifications. To facilitate the understanding of how this is performed we first introduce our usage of the algebraic methodology for problem solving.

In the algebraic methodology, a problem is formulated as an expression in a *problem algebra* and its solution may be obtained by mapping this expression into the *solution* in a *solution algebra*. In this approach the problem algebra and the solution algebra must be *similar*, i.e., for each operation of arity n in the problem algebra, there is an operation of arity n in the solution algebra. Then the process of solving a problem can be viewed as simply calculating the homomorphic image of the problem given in the problem or *source* algebra into the solution or *target* algebra. For example, a problem stated as finding the value of the expression $x + y * z$ in the algebra of arithmetic expressions over variables, where x=6, y=3, and z=9, can be solved by mapping this expression into the number in the algebra of integers with operations addition and multiplication that is precisely the value of this expression. However, in most situations the problem algebra and the solution algebras are not similar. In this case, in order for this process to work one needs to derive the signature of the problem algebra into the solution algebra[Coh81]. For each operation in the signature of the problem algebra, the derivation process consists of generating a derived operation (called a macro-operation in programming jargon) in the solution algebra that evaluates the

portion of the problem constructed by the source operation. The algebra defined by these derived operations is a subalgebra of the solution algebra which is similar to the problem algebra as required by this approach. A homomorphism mapping the source algebra into the subalgebra of the target algebra defined by the derived operations is an embedding of the source algebra into the target algebra. In the above example of the evaluation of the expression $x + y * z$ in the algebra of natural numbers, if this algebra has only the successor operation, successor(n) = n+1, and the test for equality, the derivation process requires us to develop derived operations that express the addition and multiplication of natural numbers in terms of successor and test for equality operations only.

Assuming that the signature of the source algebra has been derived into the target algebra, the solution of a problem formulated as a term of the source algebra is computed by identifying the source algebra operations which were used to construct the problem expression and performing the appropriate derived operations in the target algebra which will construct the solution expression. A derived operation is chosen such that given target images of the source operation arguments it constructs the target image of the result of the source operation applied on those source arguments. The mapping of an expression from the source to the target algebra is then computed by identifying the free generators and operations used to create the source expression and building the target expression by employing the derived target operations associated with each free generator and operation used in the source expression. Note, the essential property of an algorithm developed on this principle is that it depends only on the (free) generators and the signature of the source algebra.

For example, the expression $x + y * z$ in the algebra of expressions, E, is constructed from three free generators, or nullary operations: $x{:}\,\emptyset \to E$, $y{:}\,\emptyset \to E$, $z{:}\,\emptyset \to E$, by two binary operations: $*{:}\,E \times E \to E$, and $+{:}\,E \times E \to E$. Thus, to create the source to target algebra mapping, we must construct derived target operations that implement the source operations in the target algebra. The assignment x=6, y=3, z=9 tells us to create constant derived target operations that generate the values 6, 3, and 9 for the three free-generators "x", "y", and "z". The source operations $+{:}\,E \times E \to E$ and $*{:}\,E \times E \to E$ are implemented in the target algebra by derived operations performing integer addition and multiplication. Thus, the embedding morphism identifies the generators x, y, and z, in the source expression and computes their corresponding target images to be $6, 3$, and 9. Then the embedding morphism identifies the source operation $*$ that creates the expression $y * z$. The target derived operation associated with $*$ is computed, taking as operands the images of y and z (i.e., 3 and 9) and generating 27. Finally, the last source operation used to construct $x + y * z$ is $+$ in the context $x + (y * z)$. The associated target derived operation is integer addition and is computed on the image of the arguments, i.e., $6 + 27$, giving 33. Thus, by associating each free generator and operation in the source algebra with an appropriate derived operation in the target algebra, we can construct the problem solving homomorphism.

We use the same methodology to implement CTL and CTL^e model checkers.

We specify the logics and models as algebras and construct the set of derived operations that will implement the model checker as an embedding of the algebra of formulas into the algebra of subsets of nodes of the model. As source algebra operations that were used to create the subformulas of a CTL formula are discovered the associated derived operations are used to compute the sets of nodes in the model that satisfy these subformulas. Consequently, in the same manner in which the expression $x+y*z$ was computed by using target operations to compute the solution to the component subexpressions, the set of nodes that satisfy a temporal logic formula is computed by the derived target operations from the sets of nodes that satisfy its subformulas.

The algebraic tools in this setting are implementations of a generic algorithm that computes the homomorphic image of a term of a source algebra in a target algebra when the generators and operations of the source algebra have been derived in the target algebra. This methodology has numerous advantages. First, the model checker is generated from its specifications, thus there is no traditional programming which makes the process easier, faster, and less error-prone. Second, the correctness of the algorithm thus generated is mathematically established. Third, this algorithm adapts to a new source logic by simply changing the signature of the source algebra and its derivation in the target algebra. This is a consequence of the fact that the algorithm computing a homomorphism depends only upon the signatures of the algebras involved. Hence, for each new specification we automatically get a new model checker. Moreover, since computing a homomorphism is a naturally parallel algorithm, a parallel implementation of the model checking algorithm can be generated [Kna94].

The remainder of this paper describes how this methodology is applied to temporal logic model checking. Section 2 gives the algebraic specification of CTL. Section 3 presents the algebraic temporal logic and model checker integration techniques using CTL as an example. Section 4 shows how, in this algebraic framework, CTL and its model checker can be extended to CTL^e. Section 5 shows how CTL^e formulas can provide insight into the structure of parallel programs. Section 6 gives some comments and conclusions.

2 Algebraic specification of CTL

We have developed a set of tools that allow the easy specification and implementation of temporal logics and model checkers. These tools implement a model checker by specifying the temporal logic formulas and the sets of nodes of the model as algebras, and the model checker as an embedding homomorphism that maps a temporal logic formula into the set of nodes in the model on which the formula is satisfied. We primarily use this methodology to implement algebraic compilers where source and target languages are algebras, and the compiler is an embedding morphism of the source language algebra into the target language algebra [Rus91]. In this paper, we describe how this methodology can also be applied to temporal logic and model checking.

Formally, a model [CES86] is a directed graph $\mathcal{M} = <N, E, P : AP \rightarrow 2^N>$

with a finite set of nodes N, a finite set of directed edges E, and a proposition labeling function P which labels each node with logical propositions describing that node. This function maps atomic propositions from the set AP to the set of nodes in N on which those propositions are true. We use the notation $succ(n), n \in N$ to denote the set of successors of n in N. A path $n_0, n_1, ..., n_m, ...$ is a sequence of nodes such that $\forall i \geq 0, n_i \in N \wedge n_{i+1} \in succ(n_i)$. The example

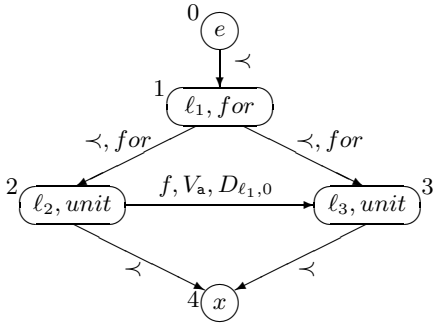

Fig. 1. Model Example

of a model in Fig.1 represents a program abstraction derived from a sequential source program by an algebraic parallelizing compiler. The nodes of this model represent computation units performed during program execution. The set of atomic propositions is $AP = \{e, x, unit, for, \ell_1, \ell_2, \ell_3\}$. The meaning of these propositions is not important for this example; they are however explained in Section 5. On this model we have $P(e) = \{0\}$, $P(x) = \{4\}$, $P(unit) = \{2, 3\}$, $P(for) = \{1\}$, $P(\ell_1) = \{1\}$, $P(\ell_2) = \{2\}$, $P(\ell_3) = \{3\}$. The successor functions $succ$ is defined by $succ(0) = \{1\}$, $succ(1) = \{2, 3\}$, $succ(2) = \{3, 4\}$ $succ(3) = \{4\}$, $succ(4) = \{\}$.

The CTL formulas used to express properties to be verified against the model are propositional logic formulas extended with temporal operators AX, EX, AU, and EU and 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, so are $\neg f_1$, $f_1 \vee f_2$, and $f_1 \wedge f_2$.
3. if f_1 and f_2 are CTL formulas, so are AXf_1, EXf_1, $A[f_1 U f_2]$, and $E[f_1 U f_2]$.

The formula AXf_1 (respectively EXf_1) is satisfied on a node if all (one or more) successors satisfy f_1. The formula $A[f_1 U f_2]$ (respectively $E[f_1 U f_2]$) is satisfied on a node n if on all (one or more) paths beginning on this node there is a node n' on which f_2 holds and f_1 holds on all nodes of the path between n and n'.

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 $n \models f_2$
$n \models AXf_1$ iff $\forall m \in N[(n,m) \in E \Rightarrow m \models f_1]$
$n \models EXf_1$ iff $\exists m \in N[(n,m) \in E \wedge m \models f_1]$
$n \models A[f_1\ U\ f_2]$ iff $\forall\ paths\ (n_0, n_1, n_2, \ldots)\ [n = n_0$ and
$\exists i[i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow n_j \models f_1]\ ]\ ]$
$n \models E[f_1\ U\ f_2]$ iff $\exists\ a\ path\ (n_0, n_1, n_2, \ldots)\ [n = n_0$ and
$\exists i[i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow n_j \models f_1]\ ]\ ]$

For example, in Fig. 1, since $1 \in P(\ell_1)$, i.e., node 1 is labeled with the proposition ℓ_1, it satisfies the formula ℓ_1, that is, $1 \models \ell_1$; since $succ(1) = \{2, 3\}$ and $\{2, 3\} \subseteq P(unit)$, i.e., both successors of node 1 satisfy the formula $unit$, node 1 satisfies the formula $AX\ unit$, that is $1 \models AX\ unit$.

To use the algebraic methodology for temporal logic model checking we must give an algebraic specification of the CTL formulas and the sets of nodes of the model. This requires the specification of the operators and the carrier set for the CTL algebra $\mathcal{A}_{CTL_\mathcal{M}}$ and the sets algebra $\mathcal{A}_{Sets_\mathcal{M}}$. For a given model $\mathcal{M}$, the carrier set of $\mathcal{A}_{CTL_\mathcal{M}}$, denoted by $F_\mathcal{M}$, is the set of CTL formulas generated from the atomic propositions which appear on the nodes of $\mathcal{M}$ using the temporal logic operators. Thus, the free generators, or nullary operations, for $\mathcal{A}_{CTL_\mathcal{M}}$ are $true$, $false$, ap_1, ap_2,..., ap_m which generate the CTL formulas denoted by their operator names. The other operations for creating CTL formulas from given CTL formulas are$\neg$, $\wedge$, $\vee$, AX, EX, AU, and EU. For example, the generator ap_3 generates the CTL formula ap_3 and from CTL formulas $ap_1 \wedge ap_5$ and $AXap_4$ the operator AU generates the formula $A[ap_1 \wedge ap_5 U AXap_4]$. Thus, the algebra $\mathcal{A}_{CTL_\mathcal{M}}$ of CTL formulas is the following term algebra:

$$\mathcal{A}_{CTL_\mathcal{M}} = <F_\mathcal{M}, true, false, ap_i, \neg, \wedge, \vee, AX, EX, AU, EU>.$$

Similarly, we specify the language of subsets of the sets of nodes of a model $\mathcal{M} = <N, E, P{:}\, AP \rightarrow 2^N>$ as a universal algebra $\mathcal{A}_{Sets_\mathcal{M}}$ with the carrier set $S_\mathcal{M} = 2^N$, the nullary operators are $\emptyset$, N, $n_i \in N$, $succ(n_i)$, $P(ap_i)$, and the binary operators are $\cap$, $\cup$, and $\setminus$ (set difference). Thus,

$$\mathcal{A}_{Sets_\mathcal{M}} = <S_\mathcal{M}, \emptyset, N, n_i, succ(n_i), P(ap_i), \cap, \cup, \setminus>.$$

A homomorphic mapping between two algebras requires that they be similar. Since the two algebras $\mathcal{A}_{CTL_\mathcal{M}}$ and $\mathcal{A}_{Sets_\mathcal{M}}$ are not similar, for each operation in the $\mathcal{A}_{CTL_\mathcal{M}}$ we construct a derived operation in $\mathcal{A}_{Sets_\mathcal{M}}$ of the same arity, thus defining in $\mathcal{A}_{Sets_\mathcal{M}}$ a subalgebra $D(\mathcal{A}_{CTL_\mathcal{M}})$ that is similar to $\mathcal{A}_{CTL_\mathcal{M}}$. Then we can implement the model checker as a homomorphism $H_\mathcal{M}{:}\, \mathcal{A}_{CTL_\mathcal{M}} \rightarrow D(\mathcal{A}_{CTL_\mathcal{M}})$ which is an embedding of $\mathcal{A}_{CTL_\mathcal{M}}$ into $\mathcal{A}_{Sets_\mathcal{M}}$ [Rus91].

¿From the programming viewpoint the derived operations required by the algebraic implementation of a model checker are parameterized expressions (called macro-operations in programming jargon) in the target algebra $\mathcal{A}_{Sets_\mathcal{M}}$ that implement the operations of the source algebra in the target algebra [Rus91]. For

example, the source operation $\wedge$ can be implemented by the target operation $\cap$ (set intersection). Given two CTL formulas f_1 and f_2 and the set of nodes on which they hold, $@_1$ and $@_2$, the set of nodes on which the formula $f_1 \wedge f_2$ holds is the set $@_1 \cap @_2$. Here, the source operation is implemented by a single target operation. However, the temporal CTL operators can not be implemented by a single target operation of the algebra $\mathcal{A}_{Sets_{\mathcal{M}}}$. For each temporal operator we need to create a special expression customized to implement the meaning of that temporal operator in the target in terms of the meaning of its operands, according to the satisfaction relations. Following the algebraic conventions [Coh81] we call these expressions derived operations. For example, if we denote by $@_1$ the set of nodes on which a formula f holds then the formula $AX\ f$ will hold on the set $\{n \in N | succ(n) \subseteq @_1\}$. That is, the unary temporal operator AX is implemented by the set-expression $\{n \in N | succ(n) \subseteq @_1\}$. Since this expression depends only on the variable $@_1$ it defines a unary derived operation in $\mathcal{A}_{Sets_{\mathcal{M}}}$ that implements the operation AX. In the next section, we discuss the specification of the target derived operations as a programming language over sets and show how the specification is used to generate the model checker implementation.

3 Model checker generation from specifications

Here we discuss the general embedding algorithm and show the specifications of the source and target algebras that are used by the tools to automatically generate this algorithm customized to the specified algebras. For that we consider $\mathcal{W}_1 = <\mathcal{T}_1, \Sigma_0^1, \Sigma_1^1, \ldots, \Sigma_{n_1}^1>$ to be the source term algebra (in our case $\mathcal{W}_1$ is $\mathcal{A}_{CTL_{\mathcal{M}}}$) to be embedded by a homomorphism into the target term algebra $\mathcal{W}_2 = <\mathcal{T}_2, \Sigma_0^2, \Sigma_1^2, \ldots, \Sigma_{n_2}^2>$ (in our case $\mathcal{W}_2$ is $\mathcal{A}_{Sets_{\mathcal{M}}}$) by derived operations. Σ_j^i is the set of operations of arity j in term algebra W_i.

To construct the general algorithm for the homomorphic embedding of $\mathcal{W}_1$ into $\mathcal{W}_2$ we use the following result [HR76]: *for each operation $r \in \Sigma$ of a term algebra $\mathcal{W} = <\mathcal{T}, \Sigma>$ there is a rewriting rule α_r in the production set of a context-free grammar $\mathcal{G}_W$ such that $\mathcal{T}$ is the same as the language generated by $\mathcal{G}_W$*. According to this result, each source operation $r \in \Sigma_0^1 \cup \Sigma_1^1 \cup \ldots \cup \Sigma_{n_1}^1$ can be seen as an equation of the form $lhs(r) ::= rhs(r)$. The generators are specified by the nullary rules $r \in \Sigma_0^1$, hence in this case $rhs(r)$ is a lexical term and $lhs(r)$ is the name of a syntax category such as variable or constant. For example, in a programming language algebra, $true$ and $false$ are boolean constants, C, thus their specification rules should be $C ::= true$ and $C ::= false$. The other kinds of constructs are specified by rules $r \in \Sigma_1^1 \cup \ldots \cup \Sigma_{n_1}^1$ where $rhs(r)$ is interpreted as a pattern specifying constructs in terms of their construct components while $lhs(r)$ is a syntax category, such as factor, term, expression, statement, etc. For example, assignment statements, $Assign$, may be specified by rules of the form $Assign ::= Var := Expr$. In addition, assume that for each $r \in \Sigma_0^1 \cup \Sigma_1^1 \cup \ldots \cup \Sigma_{n_1}^1$, $d(r)$ is a parameterized expression in $\mathcal{W}_2$ where the parameters are the target algebra expressions corresponding to the construct components of the source algebra expression specified by rule r. These parameters are typically

labeled $@_1, @_2, \ldots, @_n$ for the n components in rule r. For example, if the target algebra is an assembly language, $d(Assign ::= Var := Expr)$ may be $@_2$; `STO R` $@_1$, where $@_2$ is the assembly language target code for $Expr$, `STO R` $@_1$ is the assembly language statement which stores the result `R` of the expression in the target variable $@_1$. The general algorithm that constructs the homomorphic embedding of $\mathcal{W}_1$ into $\mathcal{W}_2$ is:

1. Let w be a term of the algebra $\mathcal{W}_1$, to be embedded in the algebra $\mathcal{W}_2$.
2. For each $r \in \Sigma_0^1 \cup \Sigma_1^1 \cup \ldots \cup \Sigma_{n_1}^1$ search for an occurrence of $rhs(r)$ in w, i.e, $w = w_1\ rhs(r)\ w_2$ and expand $d(r)$ into the term $target(r)$ of the target algebra $\mathcal{W}_2$ using as parameters the target expressions associated with the language symbols in $rhs(r)$ in w; if $r \in \Sigma_0^1$, $d(r)$ has no parameters.
3. Substitute the tuple $(lhs(r), target(r))$ for the occurrence of $rhs(r)$ in w discovered in (2), i.e., map w into $w_1\ (lhs(r), target(r))\ w_2$ and make sure that only the component $lhs(r)$ of the tuple $(lhs(r), target(r))$ is used in (2) when searching in w for occurrences of $rhs(r'), r' \in \Sigma_0^1 \cup \Sigma_1^1 \cup \ldots \cup \Sigma_{n_1}^1$.
4. Continue applying rules (2) and (3) until no occurrence of $rhs(r)$ is found in w for any $r \in \Sigma_0^1 \cup \Sigma_1^1 \cup \ldots \cup \Sigma_{n_1}^1$. If w reduces to $w' = (lhs(r), target(r))$ then it is a syntactically correct term of syntax category $lhs(r)$ in $\mathcal{W}_1$ and its image in $\mathcal{W}_2$ is $target(r)$; otherwise w was, in fact, not a term in $\mathcal{W}_1$.

The term algebra $\mathcal{W}_1$ can be viewed as a language and the rules r as the BNF specification rules of this language. Hence, this embedding algorithm can be implemented by a source language recognizer (a bottom-up parser) which is compositional (i.e., recognizes source language constructs in terms of their construct components) and a target code generator that is compositional (i.e., generates target language constructs in terms of their target construct components). Problems raised by such an implementation and their solutions are discussed in [Rus91]. This is precisely the algorithm performed by an algebraic compiler [Rus91, KR93]. Here we will assume that this algorithm is given and will focus on its customization to the languages $\mathcal{A}_{CTL_\mathcal{M}}$ and $\mathcal{A}_{Sets_\mathcal{M}}$.

In customizing the embedding morphism algorithm to the model checker application, we first derive grammar rules from the signatures of $\mathcal{A}_{CTL_\mathcal{M}}$ algebra operations. The signature $\wedge\colon F_M \times F_M \rightarrow F_M$ of the $\mathcal{A}_{CTL_\mathcal{M}}$ operation $\wedge$ generates the grammar rule $F_M ::= F_M \wedge F_M$. Similarly, $ap_i\colon \emptyset \rightarrow F_M$ generates the BNF rule $F_M ::= ap_i$ and $AU\colon F_M \times F_M \rightarrow F_M$ yields $F_M ::= A\ [\ F_M\ U\ F_M\ ]$. The BNF rules for the other algebraic operations are generated in the same manner. Here, we have specified one syntax category, F_M, the set of formulas over the propositions in the model $\mathcal{M}$. These rules create an ambiguous grammar which must be altered to remove the ambiguities. We can disambiguate this grammar by partitioning the carrier set F_M into 4 levels $F_{M_1}, F_{M_2}, F_{M_3}$, and F_{M_4}. This method is commonly used in compiler construction. By partitioning the carrier in this manner, the ambiguities are removed and the precedence of operations is ensured. The complete grammar of the CTL language is discussed below.

¿From the grammar specification of the source algebra, the tools generate a pattern matching parser [Kna94] which parses a CTL formula in a bottom-up fashion to discover which source algebra operations were used to create the

CTL formula. This corresponds to the searching phase of step (2) in the general algorithm above. For each source operation discovered, the parser initiates the macro-processor to expand the associated derived target operation to build the image of the source construct in the target algebra. This corresponds to the mapping of $d(r)$ into $target(r)$ in step (2) of the general algorithm above. This parser is significantly different from most parsers in that it scans the source text in any order to locate patterns that match the right hand sides of the specifying grammar rules. Upon finding a match, in the proper context, the parser replaces the right hand side pattern with the symbol on the left hand side, expands the appropriate derived target operation, and repeats the process. Since the right hand sides of more than one rule may match overlapping portions of the source text, each rule is associated with a set of context and noncontext pairs to ensure that the proper rule is used when the right hand side pattern is found [RH94]. Moreover, various relationships may exists among the right-hand sides of specification rule. For example, there may be rules r_1, r_2 such that $rhs(r_1) = w_1\ rhs(r_2)\ w_2$. Obviously in this case not all occurrences of $rhs(r_2)$ in a string w specify constructs of syntax category $lhs(r_2)$. Contexts and noncontexts [Rus88] have been introduced to handle such situations. A context pair (x, y) associated with a specification rule r describes the context in which the $rhs(r)$ specifies the $lhs(r)$. Here x is the string of symbols which may appear to the left of the matched pattern and y is the string of symbols which may appear to the right of the pattern. Hence, if the context of the pattern in the source text matches one of the context pairs associated with the rule, then that rule should be used for the reduction. A noncontext pair takes the same form and describes the context in which $rhs(r)$ does not specify $lhs(r)$, hence the rule cannot be used to make the reduction. For example, if the source text has been reduced to $F_{M_4} \vee F_{M_3} \wedge F_{M_2}$, should the rule $F_{M_3} ::= F_{M_3} \wedge F_{M_2}$ be matched, or should the rule $F_{M_4} ::= F_{M_4} \vee F_{M_3}$ be matched? The pair $(\vee, \$)$ (where \$ represents the beginning or end of the source text, depending on the context) is a context pair for the rule $F_{M_3} ::= F_{M_3} \wedge F_{M_2}$ indicating that this rule should be used. Whereas the pair $(\$, \wedge)$ is a noncontext pair for the rule $F_{M_4} ::= F_{M_4} \vee F_{M_3}$, indicating that this rule should not be used. It is the use of this concept of a *viable context* as opposed to the traditional *viable prefix* that guarantees the correctness of this pattern matching parser and allows its parallelization.

The derived operations associated with the source specification rules describe the target image of the computation contents of the source constructs specified by these rules. Hence, the terms in the target algebra specified by derived operations are called target images of the source constructs. Target images implement source computations in the target algebra but do not actually perform these computations. However, we are interested in the *actual* set of nodes in the model which satisfy a formula, not a term which tells *how* to calculate that set of nodes. Thus, in our case the derived operations need to be evaluated by the compiler and therefore are written in a pseudo-programming language for sets. This language allow us to specify the sequence of set operations used to build the set of nodes on which a CTL formula is satisfied when we know the set of nodes

on which the components of this formula are satisfied. Thus, when the pattern matching parser finds $rhs(r)$ in the source text and replaces it with $lhs(r)$, it initiates the execution of the derived operation, $d(r)$. The expected operations of $\cap$, $\cup$, and $\setminus$ as well as some conditional set description notations are included. We also allow set assignment statements and while-loop constructs.

Source specification rules are expressed by BNF notation, i.e., are constructs of the form $r : A_0 ::= t_0 A_1 t_1 \ldots t_{n-1} A_n t_n$ where t_i, $0 \leq i \leq n$, are fixed (terminals) and A_i, $0 \leq i \leq n$, are variables (non-terminals). Variables represent components of the constructs (CTL formulas) specified by the rules r and are identified by their indices $i = 0, 1, \ldots, n$. The target images of these components are denoted by $@_i$, $i = 0, 1, \ldots, n$, respectively, and are used as parameters in the construction of the derived operations $d(r)$ associated with the rules r. This convention is used here as follows: (1) the parameter $@_0$, which often appears on the left hand side of an assignment statement, represents the target image of the construct specified by r and thus is associated with the $lhs(r)$ for the derived operation $d(r)$; (2) parameters $@_1, @_2$, and $@_3$ are used as the names for the target images of the expressions associated with the first, second, and third parameters on the right hand side of the grammar rule. These target images are generated by the derived operations associated with the specification rules that recognize the components of the construct specified by r. Below are BNF grammar rules and their associated derived target operations.

The source operation $true$ whose signature is $true: \emptyset \rightarrow F_{M_1}$ provides the BNF rule $F_{M_1} ::= true$ and the derived operation $@_0 = N$. Since the CTL formula $true$ holds on all nodes in the model, the target expression associated with "F_{M_1}" is N, the set of all nodes in the model. Similarly, since the formula $false$ is not satisfied on any node in the model, the operation $false$ provides the BNF rule $F_{M_1} ::= false$ and the derived operation $@_0 = \emptyset$. The set of nodes associated with the atomic proposition ap is $P(ap)$, the set of nodes on which ap holds. That is, the operation ap provides the BNF rule $F_{M_1} ::= ap$ and the derived operation $@_0 = P(ap)$.

For the CTL operation $\neg$, with signature $\neg: F_{M_1} \rightarrow F_{M_2}$, we have the BNF rule $F_{M_2} ::= \neg F_{M_1}$ and the derived operation $@_0 = N \setminus @_1$. For the operation $\wedge$, with signature $\wedge: F_{M_3} \times F_{M_2} \rightarrow F_{M_3}$ we have the BNF rule $F_{M_3} ::= F_{M_3} \wedge F_{M_2}$ and the derived operation $@_0 = @_1 \cap @_2$. That is, the set of nodes on which $f_1 \wedge f_2$ is satisfied is the intersection of the set of nodes on which f_1 is satisfied and the set of nodes on which f_2 is satisfied. In a similar manner, the CTL operation of $\vee$ gives the BNF rule $F_{M_4} ::= F_{M_4} \vee F_{M_3}$ and the derived operation $@_0 = @_1 \cup @_2$.

For the temporal operator AX we have the BNF rule $F_{M_2} ::= AX\ F_{M_1}$ and the derived operation $@_0 = \{n \in N | succ(n) \subseteq @_1\}$. This derived operation conditionally uses the target operation $\cup$ over singleton sets $\{n\}$ to generate the set of all nodes, n, such that all successors of n are in the set $@_1$. Similarly, for EX, we have the BNF rule $F_{M_2} ::= EX\ F_{M_1}$ and the derived operation $@_0 = \{n \in N | succ(n) \cap @_1 \neq \emptyset\}$. Here, the derived operation finds all nodes r which have at least one successor in $@_1$.

The derived target operations for the temporal operators AU and EU require a more complicated use of the $\cup$ target operation. Here we see the use of assignment statements and loops to control the application of $\cup$. For AU we have, on the left below, the derived operation which implements the least fixed point solution to the equation $@_0 = lfp(@_2 \cup (@_1 \cap \{n \in N | succ(n) \subseteq @_0\}))$. Similarly, for EU we have, on the right below, the derived operation which implements the least fixed point solution to the equation $@_0 = lfp(@_2 \cup (@_1 \cap \{n \in N | succ(n) \cap @_0 \neq \emptyset\}))$ [CGL94].

r: $F_{M_1} ::= A\ [\ F_{M_4}\ U\ F_{M_4}\ ]$
$d(r)$: let Z, Z' be sets;
$Z = \emptyset$; $Z' = @_2$;
while ($Z \neq Z'$) do
$Z = Z'$;
$Z' = Z' \cup \{n \in N | n \in @_1 \wedge succ(n) \subseteq Z\}$;
$@_0 = Z$;

r: $F_{M_1} ::= E\ [\ F_{M_4}\ U\ F_{M_4}\ ]$
$d(r)$: let Z, Z' be sets;
$Z = \emptyset$; $Z' = @_2$;
while ($Z \neq Z'$) do
$Z = Z'$;
$Z' = Z' \cup \{n \in N | n \in @_1 \wedge succ(n) \cap Z \neq \emptyset\}$;
$@_0 = Z$;

Due to the splitting of the carrier set F_M into partitions $F_{M_1}, F_{M_2}, F_{M_3}$, and F_{M_4}, we have 4 additional rules, $F_{M_2} ::= F_{M_1}$, $F_{M_3} ::= F_{M_2}$, $F_{M_4} ::= F_{M_3}$, $F_{M_1} ::= (\ F_{M_4}\ )$, with the same derived operation $@_0 = @_1$. These rules and derived operations are the complete set of specifications required by the tools to generated the model checking algorithm.

4 Extending the temporal logic and model checker

As stated earlier, the algebraic tools provide a framework which allows users of temporal logic to modify and experiment with temporal logics and model checkers in various forms. We demonstrate how easily this can be accomplished when the temporal logic and model checker are specified and implemented within the framework of this algebraic methodology by extending CTL to CTL^e, a logic written over propositions labeling the edges as well as the nodes of a model.

To label the edges of a model with propositions, we extend the definition of a model to $\mathcal{M} = <N, E, P_n : AP_n \rightarrow 2^N, P_e : AP_e \rightarrow 2^E>$, where N and E are as before, P_n maps atomic node 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. Since we allow $\mathcal{M}$ to be a multi-graph, edges may not be uniquely identified by their source and target. When referring to an edge e, $\sigma(e)$ denotes the source of edge e and $\tau(e)$ denotes the target of e. Also, a path $n_0, e_0, n_1, e_1, ...$ is the sequence of nodes n_i and edges e_i such that $\forall i \geq 0, n_i \in N \wedge e_i \in E \wedge n_i = \sigma(e_i) \wedge n_{i+1} = \tau(e_i)$. In the example in Fig. 1, the set of atomic edge propositions AP_e is $\{\prec, for, f, V_{\mathtt{a}}, D_{\ell_1,0}\}$. Again, the meaning of these propositions is explained in the next section and is not important here. In this example, the edges (1,2) and (1,3) are labeled with the proposition for, thus $P_e(for) = \{(1,2), (1,3)\}$.

To create CTL^e we define edge formulas to be non-temporal formulas over the edge propositions constructed by the following rules:

1. $true$, $false$ and all edge propositions $ap_e \in AP_e$ are CTL^e edge formulas.
2. 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 a edge formula f for a model $\mathcal{M}$ we write $\mathcal{M}, e \models f$ or $e \models f$. Satisfaction of edge formulas is defined below:

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

As in CTL, CTL^e uses the following rules to define formulas:

3. $true$, $false$ and all atomic node propositions $ap_n \in AP_n$ are CTL^e formulas.
4. if f_1 and f_2 are CTL^e formulas, so are $\neg f_1$, $f_1 \vee f_2$, and $f_1 \wedge f_2$.

In this extension, we also allow the temporal operators to be subscripted with edge formulas to specify the conditions which must be met by the edges of the paths. Thus we add the following CTL^e syntax rule:

5. 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 CTL^e formulas.

The formula $AX_{\{f_e\}}f_1$ (respectively $EX_{\{f_e\}}f_1$) is satisfied on a node if all (one or more) successors satisfy f_1 and the edge 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 (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 f_e holds on all edges before this node. Hence, the satisfaction rules of these CTL^e formulas are given by:

$$n \models EX_{\{f_e\}}f_1 \text{ iff } \exists e \in E[n = \sigma(e) \wedge e \models f_e \wedge \tau(e) \models f_1]$$
$$n \models AX_{\{f_e\}}f_1 \text{ iff } \forall e \in E[\sigma(e) = n \Rightarrow (e \models f_e \wedge \tau(e) \models f_1]$$
$$n \models A[f_1\ U_{\{f_e\}}\ f_2] \text{ iff } \forall\ paths\ (n_0, e_0, n_1, e_1, \ldots)\ [n = n_0 \text{ and } \exists i[i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow (n_j \models f_1 \wedge e_j \models f_e)]\ ]\ ]$$
$$n \models E[f_1\ U_{\{f_e\}}\ f_2] \text{ iff } \exists\ a\ path\ (n_0, e_0, n_1, e_1, \ldots)\ [n = n_0 \text{ and } \exists i[i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow (n_j \models f_1 \wedge e_j \models f_e)]\ ]\ ]$$

In specifying the CTL^e algebra $\mathcal{A}_{CTL^e_{\mathcal{M}}}$, we must add operations for the edge formulas. That is, new generators for the edge propositions and operators to create edge formulas must be added. The edge formula free generators are $true_e$, $false_e$, ap_{e_1}, ap_{e_2}, ..., and ap_{e_m} corresponding to $true$ and $false$ edge formulas and the edge propositions $ap_{e_j}, 1 \leq j \leq m$, that label the edges of the model $\mathcal{M}$. We also add edge versions of the boolean operators $\vee_e$, $\wedge_e$, and $\neg_e$ and subscript them with an "e" to distinguish them from the CTL^e boolean operators $\vee$, $\wedge$, and $\neg$. In the actual implementation the subscripts are dropped since the parser can determine which logical operation is being employed by the context of the operator. Here we split the carrier set into partitions EF_M (CTL^e edge formulas) and F_M (CTL^e formulas). Thus, the signatures of the new operators above specify the edge formula partition EF_M. In addition to the new operators, we modify the temporal operators AX, EX, AU, and EU to

allow path quantification. The non path quantified operations are removed since they can be implemented by the path quantifying operations by using $true$ as the edge formula. EX and AX must be modified from their previous unary form to become binary operators which create a CTL^e formula from an edge formula and a CTL^e formula, that is $AX, EX: EF_M \times F_M \rightarrow F_M$. Also, AU and EU become ternary operators with the same signature $AU, EU: F_M \times EF_M \times F_M \rightarrow F_M$, to create a CTL^e formula from a CTL^e edge formula and two CTL^e formulas. In conclusion we have

$$\mathcal{A}_{CTL^e_{\mathcal{M}}} = <<F_{\mathcal{M}} EF_{\mathcal{M}}>, true, false, ap_i, true_e, false_e, ap_{e_j} \\ \neg, \wedge, \vee, \neg_e, \wedge_e, \vee_e, AX, EX, AU, EU>.$$

We also extend the target algebra to accommodate edge propositions. Thus, the carrier set of $\mathcal{A}_{Sets_{\mathcal{M}}}$ is extended to a split carrier set of sets of nodes and sets of edges. We add free generators for the empty set of edges $\emptyset_e$, for the entire set of edges E, for each edge $e_j \in E$, and for the set of edges labeled by each edge proposition $P(ap_{e_j})$ for each $ap_{e_j} \in AP_e$. We also add the generators $\sigma(e)$ and $\tau(e)$ for each edge $e \in E$. The extended algebra can be specified as:

$$\mathcal{A}_{Sets_{\mathcal{M}}} = <<S_{\mathcal{M}}, E_{\mathcal{M}}>, \emptyset, N, n_i, succ(n_i), P(ap_i), \sigma(e_i), \tau(e_i), \\ \emptyset_e, E, e_i, P(ap_{e_i}), \cap, \cup, \setminus>.$$

With the source and target algebras appropriately extended, we can deduce the source BNF specification rules and their derived operations that specify the new model checker. In specifying the BNF rules from the signatures of the source algebra operations, we again generate an ambiguous grammar. We take the same approach as before and split F_M into F_{M_1}, F_{M_2}, F_{M_3}, and F_{M_4} and split EF_M into EF_{M_1}, EF_{M_2}, EF_{M_3}, and EF_{M_4}.

The non-temporal grammar rules and their derived operations from the CTL specification are also used in the specification for CTL^e. The source specification rules and their derived target operations for the new operation generators are given as the following tuples: $(EF_{M_1} ::= true_e;\ @_0 = E)$, $(EF_{M_1} ::= false_e;\ @_0 = \emptyset_e)$, $(EF_{M_1} ::= ap_{e_j};\ @_0 = P(ap_{e_j}))$, $(EF_{M_2} ::= \neg EF_{M_1};\ @_0 = E \setminus @_1)$, $(EF_{M_3} ::= EF_{M_3} \wedge EF_{M_2};\ @_0 = @_1 \cap @_2)$, and $(EF_{M_4} ::= EF_{M_4} \vee EF_{M_3};\ @_0 = @_1 \cup @_2)$. These are similar to previous derived operations.

The derived operation for the path quantifier AX operator will find all nodes n such that all successors, n', of n, via an edge e, satisfy the formula specified by F_{M_4} and e satisfies the edge formula specified by EF_{M_4}. Thus,

r: $\quad F_{M_1} ::= AX\{EF_{M_4}\}F_{M_4}$;
$d(r)$: $\quad @_0 = \{n \in N | \forall e \in E[\sigma(e) = n \Rightarrow (e \in @_1 \wedge \tau(e) \in @_2)\ ]\}$

The derived operation below for the path quantified EX operator will find all nodes n such that $\exists e \in E$, $\sigma(e) = n$, which satisfies the edge formula specified by EF_{M_4}, and whose target satisfies the formula specified by F_{M_4}. Thus,

r: $\quad F_{M_1} ::= EX\{EF_{M_4}\}F_{M_4}$;
$d(r)$: $\quad @_0 = \{n \in N | \exists\ e \in E[\sigma(e) = n \wedge e \in @_1 \wedge \tau(e) \in @_2]\}$

For AU and EU we have the following specification rules and derived operations which solve, respectively, the equations $@_0 = lfp(@_3 \cup (@_1 \cap \{r \in N | \forall e \in E, \sigma(e) = r \Rightarrow e \in @_2\} \cap \{r \in N | succ(r) \subseteq @_0\}))$ and $@_0 = lfp(@_3 \cup (@_1 \cap \{r \in N | \exists e \in E, \sigma(e) = r \wedge e \in @_2 \wedge \tau(e) \in @_2\}))$:

r:	$F_{M_1} ::= A[F_{M_4} U \{EF_{M_4}\} F_{M_4}]$	r:	$F_{M_1} ::= E[F_{M_4} U \{EF_{M_4}\} F_{M_4}]$
$d(r)$:	let Z, Z' be sets;	$d(r)$:	let Z, Z' be sets;
	$Z = \emptyset$; $Z' = @_3$;		$Z = \emptyset$; $Z' = @_3$;
	while ($Z \neq Z'$) do		while ($Z \neq Z'$) do
	$Z = Z'$;		$Z = Z'$;
	$Z' = Z' \cup \{n \in N \mid n \in @_1$		$Z' = Z' \cup \{n \in N \mid n \in @_1$
	$\wedge \forall e \in E, \sigma(e) = n \Rightarrow$		$\wedge \exists e \in E, \sigma(e) = n \wedge$
	$(e \in @_2 \wedge \tau(e) \in @_3)\}$;		$e \in @_2 \wedge \tau(e) \in @_3)\}$;
	$@_0 = Z$;		$@_0 = Z$;

The algebraic tools then use this specification to generate the model checker for the extended logic CTL^e.

5 A program abstraction model

Part of our motivation for this research is our own need for custom temporal logics and model checkers which we use in algebraic compilers. By representing the structure of a program as a program abstraction model, labeling the nodes and edges with descriptive propositions, and describing optimization opportunities as temporal logic formulas, we can use a model checker to locate the parts of the program where optimizations can be applied [RV97]. Program abstraction models are not traditional control flow graphs; they have the fundamental property that all computations at their nodes can execute concurrently. To ensure the correctness of the computation the execution order is restricted by dependency (flow and data) relationships between nodes. The structure of the code fragment below is represented in the model in Fig. 1.

```
ℓ1 for i := 1 to 100 do
ℓ2     a[i] := b[i] ;
ℓ3     c[i] := a[i] * d[i];
   endfor
```

Nodes in the model represent computation *units* and control structures in the program, in this case, the three statements above, and are labeled by properties identifying the computation contents they represent, in this case the labels ℓ_1, ℓ_2, ℓ_3, for, and $unit$. Node 1, labeled ℓ_1, for, represents the `for` loop at ℓ_1, node 2 labeled by ℓ_2, $unit$, and node 3, labeled by ℓ_3, $unit$, are the computation units which represent the assignment statements at ℓ_2 and ℓ_3, respectively. The two additional nodes labeled e and x indicate, respectively, the entry and exit points of the computation represented by the model. Edges in the model are also labeled with propositions that represent data and control flow dependencies between the computations at the nodes. The proposition $\prec$ represents a control

flow dependency indicating that the computation at the source of the edge must complete before the computation at the target of the edge starts. Thus, the entry node 0, completes before the loop at node 1 begins; computation at node 1 completes before the assignment statements at nodes 2 and 3 begin. The edges labeled by the proposition for are control dependencies associated with a `for` loop and represent the concurrent initiation of all iterations of the computations in the loop. That is, the edge from 1 to 2 represents the initiation of all 100 instances of the computation at node 2. Thus, the necessary data dependency edges must be added to ensure that the computation will execute correctly. Since ℓ_2 writes a value in `a[i]` which is used by ℓ_3 during the same iteration, a data dependency edge from node 2 to node 3 is added. This edge is labeled by three propositions, $f, V_{\mathtt{a}}$, and $D_{\ell_1,0}$ where f indicates that this is a data *flow* dependency, $V_{\mathtt{a}}$ indicates that the variable `a` caused the dependency, and $D_{\ell_1,0}$ indicates that the data dependency is over loop ℓ_1 with a data dependency distance of 0. The data dependency distance is the number of loop iterations crossed between the computations causing the data dependency[Ban88]. Hence, every instance of the computation at node 3 must wait for the completion of the computation at node 2 from the same iteration of the loop ℓ_1. Data dependencies that have a positive or unknown distance are labeled by the proposition $D_{\ell,+}$ and $D_{\ell,?}$ respectively.

¿From this model of the program structure, the compiler designer can instruct the compiler to pose questions about the computations it processes in the form of CTL^e formulas. The answers to these questions are sets of nodes in the model that satisfy the CTL^e formulas. One question often asked is whether the iterations of a loop can be executed in parallel. The iterations of a loop can execute in parallel if they are *independent*. For example, if there are no data dependencies with positive or unknown distances between the iterations performed by the loop ℓ_1 then all iterations of this loop can be executed in parallel. This condition is stated by the CTL^e formula:

$$1 \models \ell_1 \wedge (AX_{\{for\}}\ (\neg(EX_{\{D_{\ell_1,+} \vee D_{\ell_1,?}\}}(true)\ )\ )\ )$$

This formula states that on all successors reachable by a for labeled edge, there does not exist an out going edge labeled by propositions for a positive or unknown data dependency distance. Note that this formula assumes that there are no nested loops or branch statements in the `for` loop. However, more general formulas can be written for the general case.

6 Conclusions and comments

We use the general algorithm that constructs the homomorphic embedding of an algebra $\mathcal{A}_1$ into an algebra $\mathcal{A}_2$ to implement temporal logic model checkers. For that we specify the temporal logic formulas and their model as algebras. This methodology has a number of advantages. First, by generating the model checker from a set of specifications, the user has great flexibility in the notation and semantics of the logic. So, by extending and modifying the specifications,

the user can generate new model checkers for different temporal logics. Second, since there is no traditional programming, and only specifications are written, there are no programming errors created by the user. Assuming that the specifications are correct, a correct model checker is generated. Third, since the homomorphism algorithm is naturally parallel, parallel model checkers can be generated, thus increasing the size of problems which can be solved with model checking. However, while these tools are designed so that users can easily extend temporal logics and model checkers to meet their specialized needs, the user still needs to be concerned with issues of logic soundness and the correctness of the model checker specifications.

Demonstrations of these tools and of the automatically generated model checkers can be found on the web page `http://www.cs.uiowa.edu/~rus/TICS`.

References

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic, Boston, 1988.

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

[CGL94] E. Clarke, O. Grumberg, and D. Long. Model checking. In *Proceedings of the International Summer School on Deductive Program Design*, 1994.

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

[HR76] W. Hatcher and T. Rus. Context-free algebra. *Journal of Cybernetics*, 6:65–77, 1976.

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

[KR93] J. Knaack and T. Rus. The environment of an algebraic compiler. Technical Report 93–04, Department of Computer Science, The University of Iowa, Iowa City, IA 52242, April 1993.

[Kri63] S. Kripke. Semantical analysis of modal logic i: Normal modal propositional calculi. *Zeitschrift f. Math. Logik und Grundlagen d. Math.*, 9, 1963.

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

[Rus88] T. Rus. Parsing languages by pattern matching. *IEEE Transactions on Software Engineering*, 14(4):498–510, 1988.

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

[RV97] 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.

This article was processed using the LaTeX macro package with LLNCS style