Algebraic Implementation of Model Checking Algorithms

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

We describe an algebraic methodology for implementing model checking algorithms. In this methodology temporal logic formulas are seen as phrases of a source language L_s and the sets of states of a as elements of an algebra of sets called the target language, L_t. Thus, the model checker becomes an algebraic compiler $\mathcal{C} : L_s \rightarrow L_t$ which maps temporal logic formulas in L_s into the sets of states of the model in L_t which satisfy these formulas. Since algebraic compilers can be automatically generated from algebraic specifications of the source and target algebras this methodology enjoys the advantage of the automatic generation of model checking algorithms from the algebraic specification of the temporal logics and their associated models. Also, since algebraic compilers implement translation via a homomorphism between the source and target algebras, which is a naturally parallel computation, the model checkers thus implemented are naturally parallel algorithms.

1 Introduction

An algebraic compiler $\mathcal{C} : L_s \rightarrow L_t$ is a *language-to-language* translator that uses an algorithm for homomorphism computation to embed a source language L_s into a target language L_t. We have developed a methodology and its supporting tools that allow us to automatically generate such compilers from algebraic specifications of the source and target languages [1]. Although algebraic compilers are typically used for program translation, many other computations can be easily implemented within this framework. In this paper, we describe the application of this methodology to generate algebraic implementations of model checking algorithms.

Model checking is a formal technique [2] used to verify the correctness of concurrent and distributed programs according to some correctness specification. Programs are represented as labeled finite state transition systems called *Kripke models* [3] or simply *models*, and correctness properties are described by formulas written in a temporal logic, which acts as a correctness specification language. In this paper, we use CTL, Computational Tree Logic [2], a propositional, branching-time temporal logic. A model checking algorithm determines which states in a model satisfy a given temporal logic formula. For example, the behavior of two concurrent processes competing for access to a critical section can be represented as a model and the mutual exclusion and absence of

starvation requirements can be expressed as temporal logic formulas. Given a model describing the processes and a formula describing mutual exclusion, the model checker can then determine which states of the model satisfy the mutual exclusion property. If all states in the model satisfy the formula, then the program satisfies the mutual exclusion property.

Formally, a model M is a tuple $M = \langle S, E, P{:}\, T \rightarrow 2^S \rangle$, where S is a finite set of states, $S = \{s_1, s_2, \ldots, s_m\}$, and E is a binary relation on S, $E \subseteq S \times S$, such that $\forall s \in S, \exists t \in S, (s,t) \in E$, that is, every state in the graph of M has a successor. For each $s \in S$ we use the notation $successors(s) = \{s' \in S | (s,s') \in E\}$. A *path* is an infinite sequence of states $(s_0, s_1, s_2, \ldots)$ such that $\forall i, i \geq 0, (s_i, s_{i+1}) \in E$. T is a finite set of *atomic propositions*, $T = \{p_1, p_2, \ldots, p_n\}$, and is a subset of AP, the set of all possible atomic propositions. P is a proposition labeling function that maps an *atomic proposition* in T to the set of states in S on which that proposition is *true*. Figure 1 shows a model [2]

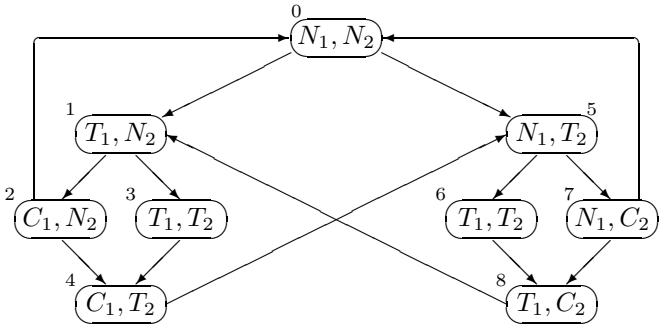

Figure 1: Model Example

for two processes competing for entrance into a critical section. The atomic propositions T_i, N_i, and C_i denote process $i, 1 \leq i \leq 2$, trying to enter the critical section, not trying to enter the critical section, and executing in the critical section, respectively.

The set of well-formed CTL formulas is described by the rules [2]:

1. The logical constants, *true* and $false$ are CTL formulas.
2. Every atomic proposition, $p \in AP$, is a CTL formula.
3. If f_1 and f_2 are CTL formulas, then so are $\neg f_1$, $f_1 \wedge f_2$, AXf_1, EXf_1, $A[f_1 U f_2]$, and $E[f_1 U f_2]$.

As in [2], we define the satisfaction relation, $\models$, of a formula $f \in CTL$ on a state s in the model M, denoted $s \models f$ or $M, s \models f$ and read "s satisfies f", as follows:

$$
\begin{array}{rcl}
s \models p & \text{iff} & s \in P(p) \\
s \models \neg f & \text{iff} & \textit{not } s \models f \\
s \models f_1 \wedge f_2 & \text{iff} & s \models f_1 \text{ and } s \models f_2 \\
s \models AX\ f_1 & \text{iff} & \forall (s,t) \in E, t \models f_1 \\
s \models EX\ f_1 & \text{iff} & \exists (s,t) \in E, t \models f_1 \\
s \models A[f_1\ U\ f_2] & \text{iff} & \forall\ \textit{paths}\ (s_0, s_1, s_2, \ldots),\ s = s_0 \text{ and} \\
 & & \exists i[i \geq 0 \wedge s_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow s_j \models f_1]] \\
s \models E[f_1\ U\ f_2] & \text{iff} & \exists\ a\ \textit{path}\ (s_0, s_1, s_2, \ldots),\ s = s_0 \text{ and} \\
 & & \exists i[i \geq 0 \wedge s_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow s_j \models f_1]]
\end{array}
$$

The set of states $\{s \in S \mid M, s \models f\}$ is called the *satisfiability set* of the formula f for model M. For the model in Figure 1, we can express the mutual exclusion property that both processes should not be in the critical section at the same time by the CTL formula $\neg(C_1 \wedge C_2)$. The absence of starvation property, which states that if a process is trying to enter the critical section it will eventually be able to do so, is described for process i by the formula $\neg T_i \vee A[true\ U\ C_i]$. The model checker would verify that both of these properties hold on all states in the model; thus, the program satisfies both properties.

Clarke, Emerson, and Sistla [2] developed a model checking algorithm that, when given a CTL formula f and a model M, labels each state of M with all sub-formulas of f which the state satisfies. This is accomplished by converting the CTL formula to a prefix form and then working from the end of the prefix formula. During each step towards the front of the formula, all sub-formulas of the sub-formula being checked have been labeled on the states on which they are true. Thus, upon completion, the formula f holds on a particular state, s, if the state s is labeled with the formula f.

We present in this paper a simple algorithm that implements a model checker based on the algorithm for homomorphism computation used by an algebraic compiler [1]. To implement a model checker as an algebraic compiler $\mathcal{C} : L_s \rightarrow L_t$ we take the source language L_s to be the language of CTL formulas for a given model M and the target language L_t to be a language describing the subsets of the states of the model M. The algebraic compiler $\mathcal{C}$ translates a CTL formula f, to the set of states, S', on which the formula f holds. That is, $\mathcal{C}(f) = S'$ where $S' = \{s \in S | M, s \models f\}$. A significant advantage of using this methodology is that this algorithm is automatically generated from its specifications by our existing TICS (Technology for Implementing Computer Software) tools [4]. Hence, the implementation of our algorithm does not require any programming activity other than the algebraic specification of the CTL language, the structuring of the model M as an algebra, and the definition of appropriate macro operations associated with the specification rules of the CTL language that embed CTL into the states of M. Consequently, this algo-

rithm does not require any preprocessing of the formula and provides a clean simple algebraic solution to the model checking problem. Another significant advantage of our approach is that since the homomorphism computation used by an algebraic compiler is a parallel algorithm, the generated model checker is also a parallel algorithm.

The paper is structured as follows: Section 2 describes the algebraic structure of the CTL logic and the model M. Section 3 describes the algebraic algorithm implementing the model checker and gives the the algebraic specification used to generate the model checker program. Section 4 provides some conclusions. An Appendix contains the complete specification of the algebraic CTL model checker.

2 Algebraic specification of CTL

The source and target languages of an algebraic compiler are defined as Ω-languages [1]. This formalism defines a language as a tuple $L =< Sem, Syn, \mathcal{L} : Sem \rightarrow Syn\rangle$ where Sem is the language semantics specified by a universal algebra of a given class of similarity, Syn is the language syntax specified by the word algebra of that same class of similarity, and $\mathcal{L}$ is a partial mapping, called the language learning function. There also exists a homomorphism $\mathcal{E} : Syn \rightarrow Sem$, called the language evaluation function, such that $\mathcal{E}(\mathcal{L}(\alpha)) = \alpha$ whenever $\mathcal{L}(\alpha)$ is defined. That is, $\mathcal{L}$ maps computations, or semantics, in the semantics algebra to their expressions in the syntax algebra and $\mathcal{E}$ maps expressions in the syntax algebra to their computations, or semantics, in the semantics algebra. $\mathcal{L}$ and $\mathcal{E}$ are related by a Galois connection [5]. In this section, we describe the CTL formulas and the satisfiability sets of a model M as Ω-languages.

Let us consider first the class of similarity $C(\Omega^9)$ defined by the signature $\Omega^9 = \langle 0, 0, 1, 2, 2, 1, 1, 2, 2\rangle$ and an unspecified set of axioms. We define CTL as the Ω^9-language $L_{ctl} = \langle \mathcal{A}_M, \mathcal{A}^w_{ctl}, \mathcal{L}_{ctl} : \mathcal{A}_M \rightarrow \mathcal{A}^w_{ctl}\rangle$. Here $\mathcal{A}^w_{ctl}$ is the word (term) algebra of the class $C(\Omega^9)$ generated by the operator scheme $\Omega^9_{ctl} = \{true,\ false,\ \neg,\ \wedge,\ \vee,\ AX,\ EX,\ AU,\ EU\}$ and a finite set of variables from AP. $\mathcal{A}_M$ is a CTL semantic algebra of type Ω^9 defined on the satisfiability sets of the CTL formulas and determined by some model M. In other words, since the meaning of a CTL formula is dependent upon the model M, the CTL semantic algebra is also dependent upon the model M. $\mathcal{L}_{ctl}$ is a mapping that associates satisfiability sets in $\mathcal{A}_M$ with the CTL expressions in $\mathcal{A}^w_{ctl}$ that satisfy them, and $\mathcal{E}_{ctl}$ is a homomorphism that evaluates CTL expressions in $\mathcal{A}^w_{ctl}$ to their satisfiability sets in $\mathcal{A}_M$.

The word algebra $\mathcal{A}^w_{ctl}$ is unique, up to homomorphism, in the class $C(\Omega^9)$

and is independent of any model. The carrier set F of this algebra is the collection of expressions, also called *terms* or *words*, built by the juxtaposition of operator symbols in Ω^9_{ctl} and variables in AP by the usual rules shown in Fig-

Operator	D		R	Description
$true:$	$\emptyset$	$\rightarrow$	F	$true \in F$
$false:$	$\emptyset$	$\rightarrow$	F	$false \in F$
$not:$	F	$\rightarrow$	F	$if\ f \in F\ then\ \neg f \in F$
$and:$	$F \times F$	$\rightarrow$	F	$if\ f_1, f_2 \in F\ then\ f_1 \wedge f_2 \in F$
$or:$	$F \times F$	$\rightarrow$	F	$if\ f_1, f_2 \in F\ then\ f_1 \vee f_2 \in F$
$AX:$	F	$\rightarrow$	F	$if\ f \in F\ then\ AX\ f \in F$
$EX:$	F	$\rightarrow$	F	$if\ f \in F\ then\ EX\ f \in F$
$AU:$	$F \times F$	$\rightarrow$	F	$if\ f_1, f_2 \in F\ then\ A[f_1\ U\ f_2] \in F$
$EU:$	$F \times F$	$\rightarrow$	F	$if\ f_1, f_2 \in F\ then\ E[f_1\ U\ f_2] \in F$

Figure 2: The operator scheme of $\mathcal{A}^w_{ctl}$

ure 2, where "D" stands for domain, and "R" stands for range. The constants $true$ and $false$ form the set of nullary operators, $\Omega_{ctl_0} = \{true, false\}$, and thus are the free generators of the ground terms of $\mathcal{A}^w_{ctl}$. In other words, the carrier set F of the word algebra $\mathcal{A}^w_{ctl}$ is constructed in the usual way by the following rules:

(i) if $f \in AP\ \cup\ \Omega^9_{ctl_0}$ then $f \in F$

(ii) if $f_1, \ldots, f_n \in F$ and $\omega \in \Omega^9_{ctl_n}$, then the "string" $\omega(f_1, \ldots, f_n) \in F$

where $\Omega^9_{ctl_n}$ is the set of operators in Ω^9_{ctl} of arity n [6,7]. Note, however, that the CTL formulas are usually written using an infix notation instead of the prefix notation given above.

The CTL semantic algebra $\mathcal{A}_M$ has the same signature Ω^9, but uses the operator scheme $\Omega^9_{sem} = \{S, \emptyset, \mathcal{C}\ \cap, \cup, Next_{all}, Next_{some}, lfp_{all}, lfp_{some}\}$. For a model M the carrier set of $\mathcal{A}_M$ is $S_M = 2^S$, the set of subsets of the states S in the model M. These are the satisfiability sets of the formulas in $\mathcal{A}^w_{ctl}$. Since $\mathcal{A}^w_{ctl}$ and $\mathcal{A}_M$ are similar, each operator in Ω^9_{ctl} corresponds to an operator of the same arity in Ω^9_{sem}. However, while the operators in Ω^9_{ctl} are used to construct well-formed CTL expressions, the operators in Ω^9_{sem} are used to construct satisfiability sets of well-formed CTL expressions. Therefore we use different symbols to denote these operators. That is, the operators $true, false, \neg, \wedge, \vee, AX, EX, AU$, and EU in Ω^9_{ctl} correspond, respectively, to the operators $S, \emptyset, \mathcal{C}, \cap, \cup, Next_{all}, Next_{some}, lfp_{all}$, and lfp_{some} in Ω^9_{sem} whose actions in S_M are defined as follows:

- S is the constant set of all states in M and $\emptyset$ is the constant empty set.
- $\mathcal{C}$ is the unary operator that produces the complement in S of its argument.
- $\cap$ and $\cup$ are the binary set union and intersection operators.
- For $\alpha \in S_M$ the unary operators $Next_{all}(\alpha)$ and $Next_{some}(\alpha)$ are defined by the equalities $Next_{all}(\alpha) = \{s \in S | successors(s) \subseteq \alpha\}$ and, $Next_{some}(\alpha) = \{s \in S | successors(s) \cap \alpha \neq \emptyset\}$, respectively, where $successors(s)$ denotes the successors of the state s in the model M.
- lfp_{all} and lfp_{some} are inspired by the Y operator for fixed point construction [8]. For $\alpha, \beta \in 2^S$, $lfp_{all}(\alpha, \beta)$ computes the least fixed point of the equation $Z = \beta \cup (\alpha \cap \{s \in S | successors(s) \subseteq (\alpha \cap Z)\})$ and $lfp_{some}(\alpha, \beta)$ computes the least fixed point of the equation $Z = \beta \cup (\alpha \cap \{s \in S | (successors(s) \cap \alpha \cap Z) \neq \emptyset\})$ [2].

Although the algebra $\mathcal{A}_M$ exists, it is not used directly in the model checking process. It is only used to explain CTL as an Ω-language.

The CTL model checker defined as an algebraic compiler maps expressions in the CTL syntax algebra $\mathcal{A}^w_{ctl}$ into *set expressions* of a *set expression language* determined by the model M while preserving the satisfiability semantics of the CTL expressions. Thus, we need to organize the model as an Ω-language whose syntax is the set of *set expressions* and whose semantics is 2^S where S is the set of states of the model. For that we consider the class of algebras $C(\Omega^4)$ defined by the signature $\Omega^4 = \langle 0, 2, 2, 2\rangle$ and an unspecified set of axioms. Further, we define the model M as the Ω^4-language $L_M = \langle \mathcal{A}_{sets}, \mathcal{A}^w_{sets}, \mathcal{L}_{sets} : \mathcal{A}_{sets} \rightarrow \mathcal{A}^w_{sets}\rangle$. Here $\mathcal{A}^w_{sets}$ is the word (term) algebra of the class $C(\Omega^4)$ generated by the operator scheme $\Omega^4_{sets} = \{\emptyset, \cup, \cap, \setminus\}$ and a finite set of variables. The ground terms of $\mathcal{A}^w_{sets}$ are the set expressions generated without any variables, i.e., set expressions which evaluate to the empty set. It is only with variables specified by some model that meaningful set expressions can be written. Therefore the set of variables for this word algebra includes the set of atomic propositions AP, the states in S, and the variable S. In a set expression containing these variables, a variable p in AP represents the set of states which satisfy p, a variable s in S represents the singleton set $\{s\}$, and the variable S represents the full set of states of a model M. Given this interpretation, the evaluation function $\mathcal{E}_{sets}$ can evaluate any expression in $\mathcal{A}^w_{sets}$ to produce the set of states in $\mathcal{A}_{sets}$ which the expression describes. This underscores the fact that the meaning of a CTL formula can only be defined for a specified model. The *set expressions*, i.e. the elements of

A^w_{sets}, are created by the same rules forming the expressions in $\mathcal{A}^w_{ctl}$ with Ω^4_{sets} replacing Ω^9_{ctl}.

$\mathcal{A}_{sets}$ is a set algebra of the class of similarity Ω^4 such that there is the homomorphism $\mathcal{E}_{sets}{:}\mathcal{A}^w_{sets} \rightarrow \mathcal{A}_{sets}$ and $\mathcal{E}_{sets}(\mathcal{L}_{sets}(\alpha)) = \alpha$ whenever $\mathcal{L}_{sets}(\alpha)$ is defined. The operator scheme for $\mathcal{A}_{sets}$ is given in Figure 3.

Operator	D		R	Description
$\emptyset:$	$\emptyset$	$\rightarrow$	S_M	$\emptyset \in S_M$
$\cap:$	$S_M \times S_M$	$\rightarrow$	S_M	$if\ S_1, S_2 \in S_M\ then\ S_1 \cap S_2 \in S_M$
$\cup:$	$S_M \times S_M$	$\rightarrow$	S_M	$if\ S_1, S_2 \in S_M\ then\ S_1 \cup S_2 \in S_M$
$\setminus:$	$S_M \times S_M$	$\rightarrow$	S_M	$if\ S_1, S_2 \in S_M\ then\ S_1 \setminus S_2 \in S_M$

Figure 3: The operator scheme of $\mathcal{A}_{sets}$ and $\mathcal{A}^w_{sets}$

Now we can define the CTL model checker as an algebraic compiler $\mathcal{MC}_M$: $L_{ctl} \rightarrow L_M$, by an embedding morphism H_{MC} from A^w_{ctl} to A^w_{sets}, which maps from the word algebra of CTL formulas $\mathcal{A}^w_{ctl}$ to the word algebra of sets $\mathcal{A}^w_{sets}$. This morphism will map a CTL formula f to the set expression that evaluates to the set of states in the model M which satisfy the formula f. In other words, the CTL model checker can be defined as a tuple $\mathcal{MC}_M = \langle \mathcal{I}, H_{MC} \rangle$, where $\mathcal{I} : 2^S \rightarrow 2^S$ is the identity map on the carrier sets of the algebras $\mathcal{A}_M$ and $\mathcal{A}_{sets}$, and $H_{MC} : \mathcal{A}^w_{ctl} \rightarrow \mathcal{A}^w_{sets}$ is an embedding of $\mathcal{A}^w_{ctl}$ into $\mathcal{A}^w_{sets}$ such that the diagram in Figure 4 is commutative. The commutativity of this

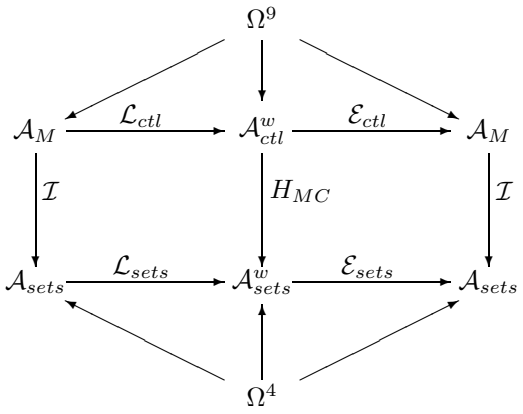

Figure 4: Ω-languages L_{ctl}, and L_M, and the model checker $\mathcal{MC}_M = \mathcal{E}_{sets} \circ H_{MC}$

diagram ensures that the mapping H_{MC} preserves the meaning of formulas in $\mathcal{A}^w_{ctl}$ when mapping them to set expressions in $\mathcal{A}^w_{sets}$. Since the model checker should evaluate CTL formulas to produce sets, not set expressions, the model checker, $\mathcal{MC}_M$ is implemented as $H_{MC} \circ \mathcal{E}_{sets}$, the composition of H_{MC} and the set language evaluation function $\mathcal{E}_{sets}$. Thus, for a CTL formula f in $\mathcal{A}^w_{ctl}$, $\mathcal{MC}(f) = (H_{MC} \circ \mathcal{E}_{sets})(f) = \mathcal{E}_{sets}(H_{MC}(f)) = \{s \in S \mid M, s \models f\}$.

Since $\mathcal{A}^w_{ctl}$ and $\mathcal{A}^w_{sets}$ are not similar, the mapping H_{MC}: $\mathcal{A}^w_{ctl} \rightarrow \mathcal{A}^w_{sets}$ is not a homomorphism, rather it is an embedding morphism. This embedding is implemented by creating a subalgebra $\mathcal{A}^w_{sctl}$ of $\mathcal{A}^w_{sets}$ similar to the algebra $\mathcal{A}^w_{ctl}$ and then constructing the monomorphism $H_{MC} : \mathcal{A}^w_{ctl} \rightarrow A^w_{sctl}$. The subalgebra $\mathcal{A}^w_{sctl}$ has the same carrier set as the algebra $\mathcal{A}^w_{sets}$. Its operations are however those of similarity class Ω^9 and are defined as derived operations ^{9,7,1} in terms of the existing operations in Ω^4_{sets}. Hence, on the one hand this subalgebra will be similar to $\mathcal{A}^w_{ctl}$ and thus $H_{MC} : \mathcal{A}^w_{ctl} \rightarrow \mathcal{A}^w_{sctl}$ can be implemented as a homomorphism, and on the other hand $\mathcal{A}^w_{sctl}$ is a subalgebra of $\mathcal{A}^w_{sets}$ and thus the images of elements in $\mathcal{A}^w_{ctl}$ by H_{MC} are computable set-expressions whose values are sets of states of the model M. The commutativity of diagram in Figure 4 ensures that the values of these set-expressions are precisely the satisfiability sets of the CTL expressions taken as arguments by H_{MC}.

In order to define the derived operations that implement the operations in Ω^9_{ctl} in the algebra $\mathcal{A}^w_{sets}$ we need some *meta-variables*, that run over the carrier set of $\mathcal{A}^w_{set}$. That is, the values taken by these meta variables are set-expressions. Further, we consider for each operation scheme $\omega \in \Omega^9_{ctl}$ a parameterized macro-operations denoted by $d(\omega)$, whose name is ω, whose parameters are meta variables denoted by $@_i$, $1 \leq i \leq arity(\omega)$, and whose bodies are defined as well-formed words in the word algebra over operations in Ω^4_{sets} and meta variables $@_i$. The variable $@_i$ used as a parameter in the macro-operation $d(\omega)$ take as values set-expressions in $\mathcal{A}^w_{sets}$ that are images of argument i of the CTL expressions constructed by the operator ω. In other words, if $arity(\omega) = n$ and ω is the function ω: $A_1 \times A_2 \times \ldots \times A_n \rightarrow A_0$, which defines CTL expressions of syntax category A_0 in terms of CTL expressions of syntax categories A_i, $1 \leq i \leq n$ then $@_i$ is the meta variable that take as values set-expressions that are images of the CTL expressions of syntax category A_i, $1 \leq i \leq n$.

Now the morphism H_{MC} can be constructed by the following rules:

1. Define the macro-operations for the generators of the $\mathcal{A}^w_{ctl}$. This is done by setting $d(true) = S$, $d(false) = \emptyset$, and $d(p) = p$ for each $p \in AP$.

2. Embed the generators of the algebra $\mathcal{A}^w_{ctl}$ in the algebra $\mathcal{A}^w_{sets}$ by the

function: $H_{MC}(true) = d(true)$, $H_{MC}(false) = d(false)$, and $\forall p \in AP.H_{MC}(p) = d(p)$

3. Extend the function defined at (2) above to the entire algebra $\mathcal{A}^w_{sets}$ by the equality: for each $w \in \mathcal{A}^w_{ctl}$ such that $w = \omega(f_1, f_2, \ldots, f_n)$ define $H_{MC}(w) = d(\omega)(H_{MC}(f_1), H_{MC}(f_2), \ldots, H_{MC}(f_n))$

Since H_{MC} is the unique extension of the function (2) to a homomorphism H_{MC} is well defined.

To improve the efficiency of the model checker it is possible to interpret the derived operations, $d(\omega), \omega \in \Omega^9_{ctl}$ as operations over sets in $\mathcal{A}_{sets}$ instead of as operations over set expressions in $\mathcal{A}^w_{sets}$. Thus, instead of mapping a formula in $\mathcal{A}^w_{ctl}$ to a set expression in $\mathcal{A}^w_{sets}$ and then evaluating the set expression to a set in $\mathcal{A}_{sets}$, we can map the formula directly into the set algebra $\mathcal{A}_{sets}$. This is accomplished by modifying the *macro processor* portion of the algebraic compiler and is discussed in the following section.

3 Algebraic implementation of a model checker

The behavior of a model checking algorithm consists of identifying the set of states of the model M that satisfy each sub-formula of a given CTL formula f and constructing from these sets, the set of states that satisfy the formula f. This is precisely the behavior of the algorithm for homomorphism computation performed by an algebraic compiler; it evaluates an expression by repeatedly identifying its generating sub-expressions and replacing them with their values. In the case of a model checker, the sub-expressions are CTL sub-formulas and their values are the sets of states in the model which satisfy the sub-formulas. Hence, to understand the algebraic implementation of the model checking algorithm, we describe first the structure of an algebraic compiler and then show its relationship with a model checker.

3.1 Structure of an algebraic compiler

The syntax of the source language of an algebraic compiler is specified by a finite set, R, of BNF specification rules. Each rule $r \in R$ corresponds to an operation in the source language syntax algebra and is an equation of the form $A_0 ::= t_0 A_1 t_2 \ldots t_{n-1} A_n t_n$ where, for each $i, 0 \leq i \leq n$, t_i is a string (possibly empty) of *terminal symbols* and each A_i is a variable called a *nonterminal symbol*. We use the notation $lhs(r)$ to denote the left-hand side of the rule r, that is, $lhs(r) = A_0$, and $rhs(r)$ to denote the right-hand side of the rule r, that is, $rhs(r) = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n$. As an example, in an algebraic model

checker, the $\mathcal{A}^w_{ctl}$ operation $\wedge\colon F \times F \rightarrow F$ could be represented as the BNF rule $F ::= F$ and F.

An algebraic compiler is specified by associating each source language specification rule $r \in R$, r: $A_0 ::= t_0 A_1 t_2 \ldots t_{n-1} A_n t_n$, with a target macro operation, $macro(r)$. This forms a *compiler specification* $CS = \{\langle r, macro(r)\rangle \mid r \in R\}$. Typically, the macro operation $macro(r)$ is a parameterized target language representation of the computation expressed by the source language construct specified by r. In a Pascal to C language translator for example, the macro operation associated with a rule specifying a Pascal `for` loop would describe, in C, the equivalent C `for` loop. That is, $macro(r)$ is defined by the compiler implementor by expressing in the target language the meaning of the source language computation specified by r in terms of the target language images of the components of the source expression. The components of the source language construct specified by $r : A_0 ::= t_0 A_1 t_1 \ldots t_{n-1} A_n t_n$ are source language constructs of syntax categories A_1,A_2,..., A_n. Hence, the formal parameters of $macro(r)$ are the nonterminals $A_1, A_2, \ldots, A_n$ and the actual parameters are target language images of the source language constructs of syntax categories A_1, A_2,..., A_n. The body of $macro(r)$ expresses the computation denoted by the constructs specified by r as a valid target construct in the target language called the target image; this target image is composed from the target images of the components of the construct specified by r. As in Section 2 we use the symbol @, with indices, to denote the target images of the components of constructs specified by r. That is, the actual parameters of the macro operation $macro(r)$ are the target images, denoted $@_1, @_2, \ldots, @_n$, of the source language constructs of syntax categories $A_1, A_2, \ldots, A_n$. For example, in the algebraic specification of a model checker, the macro operation associated with the rule $F ::= F$ and F must express, in the target language of sets, the set of states on which a formula specified by this rule holds. This set of states is the intersection of the two sets of states which satisfy the two sub-formulas of any formula specified by this rule. This macro operation implements the $\wedge$ operator in the source language by the $\cap$ operator in the target language. This operation may be defined by the expression $@_1 \cap @_2$ which would be associated with the rule $F ::= F$ and F in the compiler specification.

An algebraic compiler is implemented by three components $\mathcal{R}, \mathcal{G}$, and $\mathcal{M}$. $\mathcal{R}$ is a pattern-matching parser recognizing valid constructs in the source language. $\mathcal{M}$ is a target language *macro processor* which expands target language macro operations associated with the specification rules used by $\mathcal{R}$ to produce valid target language constructs called *images*. $\mathcal{G}$ provides an interface between $\mathcal{R}$ and $\mathcal{M}$. Let $r : A_0 = t_0 A_1 t_1 \ldots t_{n-1} A_n t_n$ be a specification rule, in R, used by $\mathcal{R}$ to recognize valid constructs specified by r in the input text. When $\mathcal{R}$

recognizes a portion of the input text specified by r, it calls $\mathcal{G}$, giving it as parameters the rule r and the portion of input text matched by the right hand side of rule r. $\mathcal{G}$ identifies the macro-operation $macro(r)$ associated with r and uses the text matched by $rhs(r)$ to identify the previously computed target images $@_1$, $@_2$,..., $@_n$ of the components of the construct recognized by $\mathcal{R}$. These target images belong respectively to syntax categories A_1, A_2, ..., A_n. $\mathcal{G}$ packages these target images into the list $macro(r)$, $(@_1, @_2, \ldots, @_n)$ which it passes to the the macro processor $\mathcal{M}$. The macro processor $\mathcal{M}$ expands the macro-operation $macro(r)$ taking $@_1$, $@_2$, ..., $@_n$ as parameters thus constructing the target image $@_0$ of syntax category A_0 of the portion of input text matched by $rhs(r)$. This target image is passed back to $\mathcal{G}$ which associates it with the left hand side of the rule r, $lhs(r)$, i.e., with A_0, thus constructing the tuple $(lhs(r), @_0)$ which replaces the portion of the text matched by $rhs(r)$ in the input. This process is illustrated in Figure 5. In the case where an alge-

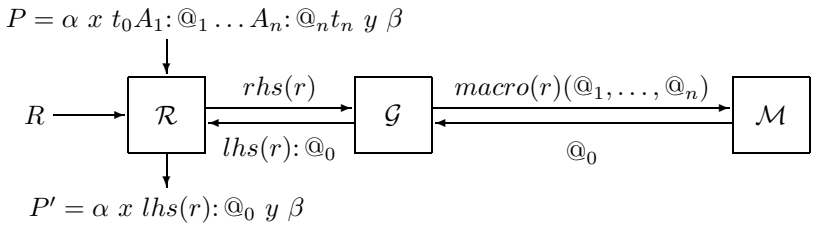

Figure 5: The integration of the components of an algebraic compiler

braic compiler $\langle \mathcal{R}, \mathcal{G}, \mathcal{M} \rangle$ implements a model checker, the source language is the language of CTL formulas and the target language is a language of sets of a model. Thus, the source language constructs recognized by $\mathcal{R}$ are sub-formulas of CTL formulas, the target images produced by $\mathcal{M}$ are the satisfiability sets of the sub-formulas recognized by $\mathcal{R}$, and $\mathcal{G}$ is the interface between them. The components $\mathcal{R}$, $\mathcal{G}$, and $\mathcal{M}$ are generated by the TICS compiler generation tools from specifications of the source and target languages.

The compilation process performed by the algebraic compiler is a sequence of transformations of the input text during which source language constructs specified by the rules $r \in R$ are discovered in the input text, their target images $@_{lhs(r)}$ are constructed by expanding their associated macros, $macro(r)$, and the portions of the input text representing such constructs are replaced by records of the form $lhs(r){:}\,@_{lhs(r)}$. Suppose that after a number of transformations the input text has the form

$$P = \alpha\ x\ t_0\ A_1{:}\,@_1\ t_1 \ldots A_n{:}\,@_n\ t_n\ y\ \beta$$

where α, β, x, and y are text strings, and the tuple $A_i{:}\,@_i$, $1 \leq i \leq n$, shows that a source language construct of syntax category A_i has been discovered by $\mathcal{R}$ as a valid component of the input and its target image constructed by $\mathcal{M}$ is $@_i$. The next transformation of P by the algebraic compiler is performed by the following three steps.

(1) For each $\langle r, macro(r)\rangle \in CS$, $\mathcal{R}$ interprets $rhs(r) = t_0 A_1 t_1 \ldots A_n t_n$ as a pattern to search for in P. $\mathcal{R}$ ignores the target images embedded in P. When an occurrence of the $rhs(r)$ is discovered in P by $\mathcal{R}$ the context [10] (x, y) is checked against the pre-computed *context* and *non-context* sets associated with r. If (x, y) is in the context set of rule r then the $rhs(r)$ can be replaced by the $lhs(r)$ preserving the syntactic validity of P, i.e., P is transformed into $P' = \alpha\ x\ lhs(r)\ y\ \beta$. If (x, y) is in the non-context set of rule r then the $rhs(r)$ can not be replaced by the $lhs(r)$ because r was not used to generate the text specified by $rhs(r)$.

(2) For each $\langle r, macro(r)\rangle \in CS$, $\mathcal{G}$ interprets $rhs(r) = t_0 A_1 t_1 \ldots A_n t_n$ as the name of the macro operation $macro(r)$. Therefore, when $\mathcal{R}$ determines that a portion of the input can be replaced by the $lhs(r)$, $\mathcal{G}$ identifies the macro-operation $macro(r)$, extracts the actual parameters $@_1, \ldots, @_n$ from the portion of the input $t_0\ A_1{:}\,@_1\ t_1 \ldots\ A_n{:}\,@_n\ t_n$ matched by $\mathcal{R}$ and calls the macro processor $\mathcal{M}$ to expand the macro operation $macro(r)$ with parameters $(@_1, \ldots, @_n)$. $@_0$ denotes the construct thus generated by $\mathcal{M}$. Then $\mathcal{G}$ associates the parameter $@_0$ with the $lhs(r)$ creating the record $lhs(r){:}\,@_0$.

(3) When $\mathcal{G}$ calls the macro processor $\mathcal{M}$ and passes it the parameters $macro(r)$ and $@_1, \ldots, @_n$, $\mathcal{M}$ builds a target image $@_0$ from the component target images $@_1, \ldots, @_n$ according to the macro $macro(r)$. The macro specifies how the macro processor will build the target image $@_0$ from the components $@_1, \ldots, @_n$. The relationship between components $\mathcal{R}$, $\mathcal{G}$, and $\mathcal{M}$ of the algebraic compiler while performing a transformation of the input text is shown in Figure 5.[11]

3.2 The macro processor generating satisfiability sets

Although the components $\mathcal{R}$ and $\mathcal{G}$ of an algebraic compiler depend only on the specification rules R, $\mathcal{M}$ depends on the macro-operations and the target language in which these macro-operations are expanded. Hence, algebraic compilers with different target languages use different macro processors to generate target images. A macro processor used for generating assembly language programs is not appropriate for generating the sets of states in a model which satisfy CTL formulas since the macro processor used by the CTL model checker generates the sets of states which satisfy CTL formulas. It builds the target

image set $@_0$ for a CTL formula, f, from target image sets $@_1, \ldots, @_n$ which satisfy the sub-formulas of f. The macros processed by this macro processor specify how to construct the set $@_0$ from the parameter sets $@_1, \ldots, @_n$. Each macro processor works in a different target algebra and thus defines an appropriate language in which the macros it processes are written. While both assembly macro processors and CTL macro processors construct target images from component target images as instructed by macro operations, the macro operations are specific to the target language. Thus, when specifying an algebraic compiler, we must be sure that an appropriate macro processor exists for the target language. To adapt the general algebraic compiler methodology to model checking, a new macro processor, named $\mathcal{M}_{\mathcal{A}_{sets}}$, must be created to compute the sets of states in the model.

The macro operations discussed in Section 2 provide a mechanism for specifying *parameterized* set expressions which define operations in the word algebra $\mathcal{A}^w_{ctl}$. These macro operations take as parameters valid set expressions and generate valid set expressions. For example, if $@_i \cup @_j$ is a parameterized set expression and $@_i$ and $@_j$ are valid set expressions then, $@_i \cup @_j$ evaluates to a valid set expression whenever $@_i$ and $@_j$ are substituted into $@_i \cup @_j$. These same parameterized set expressions can also define operations in the set algebra $\mathcal{A}_{sets}$ which take sets as parameters and evaluate to sets. Considering the nature of our problem, we design a macro processor which implements the macro operations in the set domain of $\mathcal{A}_{sets}$ instead of the set expressions domain of $\mathcal{A}^w_{sets}$.

The language in which the macro operations processed by $\mathcal{M}_{\mathcal{A}_{sets}}$ are written is a simple imperative language that allows us to construct satisfiability sets using set operations $(\cap, \cup, \setminus)$ over the given generator sets of $\mathcal{A}_{sets}$ and set variables over the carrier set of $\mathcal{A}_{sets}$, denoted by $@i$, $i = 0, 1, \ldots, n$. Specifically, since our macros expand into the satisfiability sets of CTL formulas specified by BNF rules, r, of the form $A_0 ::= t_0 A_1 t_1 \ldots t_{n-1} A_n t_n$, the variables $@i$, $1 \leq i \leq n$, stand for the satisfiability sets of the CTL components of syntax categories A_i, $1 \leq i \leq n$, and $@_0$ stands for the satisfiability set of the CTL formula matched by rule r. This language has been extended with the conditional set construction operator denoted by $\{s \in S| < \textit{condition on } s >\}$. There are also macro assignment statements and while loops which are used to control the application of the target language algebra operations. Boolean expressions over sets using the subset and equivalence relations are also available. These provide a convenient mechanism to write macros to express complex derived operations in the target language algebra of sets. The implementation of this macro processor, $\mathcal{M}_{\mathcal{A}_{sets}}$ is discussed below in 3.4.

3.3 Generating a model checker program

Here we develop the implementation of $\mathcal{MC}_M$ using the methodology of the algebraic compiler. The essential element here is the embedding of the source algebra $\mathcal{A}^w_{ctl}$ into the target algebra $\mathcal{A}_{sets}$ by derived (macro) operations. In this section we show the BNF rules, R, which specify $\mathcal{A}^w_{ctl}$ and the macro operations that map the CTL formulas into $\mathcal{A}_{sets}$ thus embedding $\mathcal{A}^w_{ctl}$ into $\mathcal{A}_{sets}$. The model checking program is automatically generated from these specifications.

The set of BNF rules that directly specify the source syntax algebra may be ambiguous, which is the case with $\mathcal{A}^w_{ctl}$. Therefore, we split the carrier set F of $\mathcal{A}^w_{ctl}$ on the layers of generation $Factor$, $Term$, and $Expression$ and write BNF rules that provide a non-ambiguous specification of $\mathcal{A}^w_{ctl}$. This yields the three syntax categories F_f, F_t, and F_e. The complete specification, with the associated macro operations, can be seen in the Appendix. To increase the readability of this specification we discuss a few of the rules and their macro operations and show an example compilation below. In

r: $F_t ::= F_t$ and F_f ;
$macro(r)$: $@_0 := @_1 \cap @_2$;

as described above, the macro operation implements the source algebra $\wedge$ operation as the target algebra $\cap$ operation. In the following,

r: $F_f ::=$ p ;
$macro(r)$: $@_0 := P(p)$;

the generator $P(p)$ gives the set of states on which the proposition p holds. The macro operation in

r: $F_f ::=$ ax F_f ;
$macro(r)$: $@_0 := \{s \in S | successors(s) \subseteq @_1\}$;

finds all states s such that all successors of s are in the set $@_1$. In

r: $F_f ::=$ a [F_e u F_e] ;
$macro(r)$: let Z, Z' be sets;
$Z := \emptyset$; $Z' := @_2$;
while ($Z \neq Z'$) do
$Z := Z'$;
$Z' := Z' \cup \{s \in S | s \in @_1 \wedge successors(s) \subseteq Z\}$;
end while
$@_0 := Z$;

the macro operation uses set variables Z and Z' to implement a least fixed point solution to the equation $Z = (@_2 \cup (@_1 \cap \{s \in S | successors(s) \subseteq (@_1 \cap Z)\}))$.

By applying the TICS compiler generation tools to the complete specification in the Appendix we obtain a program that implements a CTL model

checking algorithm. As described in Section 3.2, when this program is run on a CTL formula and a model M it gives as the result the set of states of M which satisfy the given formula. We should also note that although we have stated that the model M dictates the creation of the algebras $\mathcal{A}^w_{sets}$ and $\mathcal{A}_{sets}$, the same model checker program can be used to find the satisfiability set of any CTL formula on any given model M. When M changes, the model checker algorithm works correctly by simply updating the generator sets of the new $\mathcal{A}^w_{ctl}$ and $\mathcal{A}_{sets}$ algebras.

As an example, consider the CTL formula *not* (C_1 *and* C_2) checked against the model in Figure 1. The $rhs(r)$ of the rule F_f ::= p will match atomic propositions C_1 and C_2 since "p" matches names of atomic propositions. The macro processor is called, once for each proposition matched, to execute the macro $@_0 := P(p)$, generating, respectively, the sets $\{2, 4\}$ and $\{6, 8\}$ since $2 \models C_1$, $4 \models C_1$, $6 \models C_2$, and $8 \models C_2$. The *lhs* symbol, F_f, and the target set images replace the respective propositions in the CTL formula to yield

$$\text{not (} F_f\text{:}\{2, 4\} \text{ and } F_f\text{:}\{6, 8\} \text{)}$$

Next, the rule F_t ::= F_f will match the first sub-formula. (The second sub-formula textually matches $rhs(r)$, but the context set of the rule does not include the tuple $\langle$ *and*, $\rangle$, so this occurrence is not matched. This parsing method is explained fully in [1,12,13]). The macro associated with the rule F_t ::= F_f is $@_0 := @_1$ and consequently only copies the target image from the first parameter yielding

$$\text{not (} F_t\text{:}\{2, 4\} \text{ and } F_f\text{:}\{6, 8\} \text{)}$$

Now the *rhs* of rule F_t ::= F_t and F_f will match the pattern "F_t and F_f." The associated macro, $@_0 := @_1 \cap @_2$, will take the set intersection of the parameter sets, which in this case results in the empty set. The empty set is associated with F_t and placed in the formula to yield

$$\text{not (} F_t\text{:}\emptyset \text{)}$$

After application of the copy rules F_e ::= F_t and F_f ::= (F_e), we are left with

$$\text{not } F_f\text{:}\emptyset$$

The *rhs* of the rule F_f ::= not F_f now matches the text and the associated macro $@_0 := S \setminus @_1$ is applied with $@_1 = \emptyset$ to yield

$$F_f\text{:}\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$$

After the application of the copy rules F_t ::= F_f and F_e ::= F_t we are left with the final form

$$F_e\text{:}\{0, 1, 2, 3, 4, 5, 6, 7, 8\}.$$

Since no more rules in R have a $rhs(r)$ which match any portion of the CTL formula text, the process is complete. Thus, this process shows that the formula $not(C_1 and C_2)$ is satisfied on all states in the model and thus mutual exclusion is assured.

3.4 Implementing the macro processor $\mathcal{M}_{\mathcal{A}_{sets}}$

The implementation of the macro processor $\mathcal{M}_{\mathcal{A}_{sets}}$ is necessary to generate model checking algorithms from their algebraic specifications. As stated before, the macro operations are written in a macro language for sets which specifies the target algebra operations to be performed to build the satisfiability sets for the CTL formulas constructed by the BNF rules. The macro processor $\mathcal{M}_{\mathcal{A}_{sets}}$ interprets the macro operations as collections of operations on sets. At the time when the model checker program is generated from its specifications, the macro operations, which are essentially code fragments written in the macro language, are translated into C language code fragments which are compiled and linked to the rest of the model checker program. These C language statements perform the operations specified by the macro operations. This translation is done by a macro *pre-processor* which translates each macro operation into a C language function which is executed when a CTL formula is being verified to construct the satisfiability set specified by the macro operation. For example, the macro operation $@_0 := @_1 \cap @_2$ associated with the rule $F_e ::= F_e$ and F_t is translated by the macro pre-processor into the C language function shown in Figure 6. (Since this rule appears as the third rule in the specification the function

```
void macro_3 ( image_struct_ptr LHS, image_struct_ptr RHS[2] )
  { set_copy ( LHS->image,
      set_intersection ( RHS[1]->image, RHS[2]->image ) ) ; }
```

Figure 6: C language function implementing a macro operation.

name is `macro_3`.) The functions `set_copy` and `set_intersection` belong to a library of set functions which are also linked to the model checker program.

The macro pre-processor which translates the macros to C language functions is itself implemented as an algebraic compiler generated by the same TICS tools used to generate the model checker. This compiler has the macro language as its source language and C as its target language. Like any algebraic compiler, it is automatically built from the algebraic specification of its source and target languages. Thus, building the macro pre-processor is not a difficult task.

4 Conclusions

The complexity of the original model checker algorithm presented by Clarke, Emerson, and Sistla [2] is $\mathcal{O}(length(f) \times (card(S) + card(E)))$. Having in view that the recognizer $\mathcal{R}$ is linear in the length of the input text f, and that macro

expansion could be polynomial in the size of the model, the worst case behavior of the model checker algorithm presented in this paper is $\mathcal{O}(length(f) \times (card(S)^2 + card(E))$. However, this is not inherent in the use of the algebraic methodology and one can write target language macros that are linear, therefore obtaining the same complexity. But the distinct advantage of using a homomorphism computation is that the generated model checker can be easily implemented by a parallel algorithm. The process performed by $\mathcal{R}$, $\mathcal{G}$, and $\mathcal{M}$ can be replicated to work on different parts of the input text at the same time, thus executing in parallel.

Another significant advantage of our methodology is that all components $\mathcal{R}$, $\mathcal{G}$, and $\mathcal{M}$ of the model checker are automatically generated from their specifications. Thus, human programming errors are avoided. If the specifications are correct, then the generated program is correct. This ensures the correctness of the model checker. This does require the implementor to understand the model checker in the algebraic framework presented above and be able to specify it as a homomorphism between algebras. However, the universality of the homomorphism computation makes various algebraic frameworks implementable by this same approach. There is very little, if any, traditional programming required by this implementation framework; the program is automatically generated. The re-usability of previous work allowing the extension of the algorithm is another advantage of using an algebraic framework. By extending the algebraic specification the implementor can change the generated program such that its behavior fits new requirements [14].

Finally, we want to observe that the application of the algebraic model checker expands beyond the usual field of interest. We are involved in a project to use CTL model checking to identify and exploit implicit parallelism in sequential programs [15,16]. The program text is analyzed by a usual recognizer and a macro-processor generates a model describing its flow of data and control. A CTL model checker uses this model and CTL formulas describing opportunities for parallelism or parallelism properties to identify states where implicit parallelism exists or to verify that an appropriate amount of parallelism in the original program has been exploited. Other CTL formulas can verify that certain optimality conditions have been satisfied.

Information about the current implementation of the CTL model checker, and various algebraic extensions are available on the World Wide Web at URL `http://www.cs.uiowa.edu/~tics/ModelChecker`.

Acknowledgments

We thank the NASA Jet Propulsion Laboratory for supporting this research.

Appendix - Algebraic Specification

Below is the complete algebraic specification used to generate an implementation of the model checker algorithm.

r_1: $F_e ::= F_e$ or F_t ;	$macro(r_1)$: $@_0 := @_1 \cup @_2$;
r_2: $F_e ::= F_t$;	$macro(r_2)$: $@_0 := @_1$;
r_3: $F_t ::= F_t$ and F_f;	$macro(r_3)$: $@_0 := @_1 \cap @_2$;
r_4: $F_t ::= F_f$;	$macro(r_4)$: $@_0 := @_1$;
r_5: $F_f ::=$ not F_e ;	$macro(r_5)$: $@_0 := S \setminus @_1$;
r_6: $F_f ::=$ (F_e) ;	$macro(r_6)$: $@_0 := @_1$;
r_7: $F_f ::=$ p ;	$macro(r_7)$: $@_0 := P(p)$;
r_8: $F_f ::=$ true ;	$macro(r_8)$: $@_0 := S$;
r_9: $F_f ::=$ false ;	$macro(r_9)$: $@_0 := \emptyset$;

r_{10}: $F_f ::=$ ax F_f ;
$macro(r_{10})$: $@_0 := \{s \in S | successors(s) \subseteq @_1\}$;

r_{11}: $F_f ::=$ ex F_f ;
$macro(r_{11})$: $@_0 = \{s \in S | successors(s) \cap @_1 \not\subseteq \emptyset\}$;

r_{12}: $F_f ::=$ a [F_e u F_e] ;
$macro(r_{12})$:
let Z, Z' be sets;
$Z := \emptyset$; $Z' := @_2$;
while ($Z \neq Z'$) do
    $Z := Z'$;
    $Z' := Z' \cup \{s \in S | s \in @_1 \wedge successors(s) \subseteq Z\}$;
end while
$@_0 := Z$;

r_{13}: $F_f ::=$ e [F_e u F_e] ;
$macro(r_{13})$:
let Z, Z' be sets;
$Z := \emptyset$; $Z' := @_2$;
while ($Z \neq Z'$) do
    $Z := Z'$;
    $Z' := Z' \cup \{s \in S | s \in @_1 \wedge successors(s) \cap Z \neq \emptyset\}$;
end while
$@_0 := Z$;

References

1. T. Rus. Algebraic construction of compilers. *Theoretical Computer Science*, 90:271–308, 1991.
2. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, 1986.
3. S. Kripke. Semantical analysis of modal logic i: Normal modal propositional calculi. *Zeitschrift f. Math. Logik und Grundlagen d. Math.*, 9, 1963.
4. T. Rus, T. Halverson, E. Van Wyk, and R. Kooima. An algebraic language processing environment. In *Sixth International Conference on Algebraic Methodology and Software Technology, AMAST '97, Proceedings*, Syney, Australia, Dec. 13–17 1997.
5. S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, New York Heidelberg Berlin, 1971.
6. S. Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Graduate Texts in Mathematics, 78. Springer-Verlag, New York, 1980.
7. P.M. Cohn. *Universal Algebra*. Reidel, London, 1981.
8. M. Gordon. *Programming Language Theory and its Implementation*. Prentice Hall, 1988.
9. R.M. Burstall and P.J. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4:17–43, 1969.
10. T. Rus and T. Halverson. Algebraic tools for language processing. *Computer Languages*, 20(4):213–238, 1994.
11. T. Rus and S. Pemmaraju. Using graph coloring in an algebraic compiler. *Acta Informatica*, 34(3):191–209, 1997.
12. T. Rus. Algebraic construction of a compiler. Technical Report 90–01, The University of Iowa, Dept. of Computer Science, 1990.
13. J.L. Knaack. *An Algebraic Approach to Language Translation*. PhD thesis, The University of Iowa, Dept. of Computer Science, Dec. 1994.
14. T. Rus and E. Van Wyk. Integrating temporal logics and model checking algorithms. In *Fourth AMAST Workshop on Real-Time Systems, Proceedings, LNCS, 1231*. Springer-Verlag, May 21 1997.
15. 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.
16. T. Rus and E. Van Wyk. Model checking tools for parallelizing compilers. In *Second International Workshop on Formal Methods for Parallel Programming: Theory and Applications, Proceedings*, April 1 1997.