Meta Languages in Algebraic Compilers [⋆]

Eric Van Wyk

Oxford University Computing Laboratory

Abstract. Algebraic compilers provide a powerful and convenient mechanism for specifying language translators. With each source language operation one associates a computation for constructing its target language image; these associated computations, called *derived operations*, are expressed in terms of operations from the target language. Sometimes the target language operations are not powerful enough to specify the required computations and one may then need to extend the target language algebras with more computationally expressive operations. A better solution is to package them in a *meta language* which can be automatically composed with the target language operations to ensure that all operations needed or desired for performing a translation are provided. In this paper, we show how imperative and functional meta languages can be composed with a target language in an example which implements a temporal logic model checker as an algebraic compiler and show how meta languages can be seen as components to be combined with a source and target language to generate an algebraic compiler.

1 Introduction

Attribute grammars [7, 1] and algebraic compilers [9] provide powerful and convenient mechanisms for specifying language translators. In both, one associates with each operation in the source language computations for constructing the target language images of source constructs created by the operation. The complexity of these computations contributes to the complexity of the entire language translator specification, and we are thus interested in means of reducing the specification's complexity by writing these computations in languages appropriate to the translation task at hand. These languages must be computationally expressive enough to perform the necessary computations, and should provide convenient programming constructs which simplify the specification process for the translator implementor. Since algebraic compilers provide solid mathematical framework which provide a clear distinction between the target language and the language used to specify the translation, they provide a better context in which to explore the issues of *meta languages*.

An algebraic compiler $\mathcal{C} : L_S \to L_T$ 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. The computations associated with each source language operation that define an algebraic compiler are written in terms using the

[⋆] This work is funded by Microsoft Research.

operations from the target language and are called *derived operations*. In some cases, the operations provided by the target language are not expressive enough to correctly specify the translation or exist at such a low level of abstraction, with respect to the source language, that the specification is excessively difficult to read and write. In such cases, the target language is extended with additional operations to make the translation possible or more easily specifiable. In this paper, we explore how different *meta languages* can be used in conjunction with operations of the target language, to correctly and conveniently specify translators implemented as algebraic compilers without extending the target language.

As an example, we develop a model checker for the temporal logic CTL (computation tree logic) [3] as an algebraic compiler which maps the source language CTL into a target language of satisfiability sets. Since the operations in the target language of sets are not powerful enough to specify general computations, we must use a meta language to provide a more computationally expressive language in which to specify this translation. We show how both functional and imperative style meta languages can be used in the specification, thus giving the language implementor some choice in choosing an appropriate meta language.

Section 2 describes CTL and model checking. In Section 3 we define algebraic languages and compilers and show how CTL and models can be specified as algebraic languages. Section 4 discusses meta languages in algebraic compilers and specifically the meta languages used to implement a model checker as an algebraic compiler. Section 5 provides the specification of the model checker as an algebraic compiler using both a functional and an imperative meta language. Section 6 contains some comments on meta languages in attribute grammars, domain specific meta languages, and future work.

2 Model Checking

We present the problem of model checking a temporal logic as a language translation problem and implement two solutions as generalized homomorphisms using different meta languages. Model checking [3] is a formal technique used to verify the correctness of a system according to a given correctness specification. Systems are represented as labeled finite state transition systems called *Kripke models* [8] or simply *models*. Correctness properties are defined by formulas written in a temporal logic. In this paper, we use CTL [3], a propositional, branching-time temporal logic as our example. A model checking algorithm determines which states in a model satisfy a given temporal logic formula, and can thus seen as a language translator which maps formulas in the temporal logic language to sets in a set language defined by the model.

A model M is a tuple $M = \langle S, E, P\colon AP \rightarrow 2^S \rangle$, where S is a finite set of states, $S = \{s_1, s_2, \ldots, s_m\}$, and E defines directed edges between states and 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 has a successor. For each $s \in S$ we use the notation $succ(s) = \{s' \in S | (s, s') \in E\}$. A *path* is an infinite sequence of states $(s_0, s_1, s_2, \ldots)$ such that $\forall i, i \geq 0, (s_i, s_{i+1}) \in E$. AP is a finite set of *atomic propositions*,

$AP = \{p_1, p_2, \ldots, p_n\}$, P is a proposition labeling function that maps an *atomic proposition* in AP to the set of states in S on which that proposition is *true*. Figure 1 shows a model [3] for two processes competing for entrance into a critical

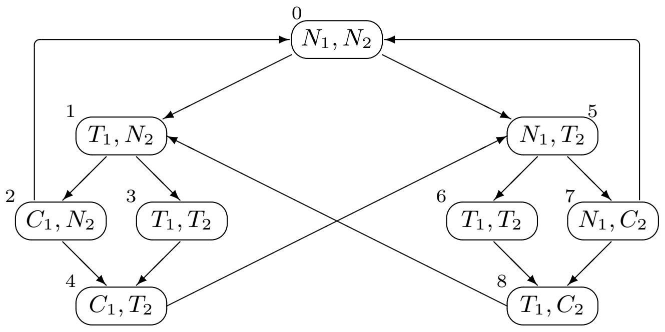

Fig. 1. Model Example

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 [3]:

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

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

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 verifies that these properties hold on all states in the model.

We present both a functional and imperative version of a CTL model checker implemented as an algebraic compiler [9] $\mathcal{MC} : L_S \to L_T$ where the source language L_S is CTL and the target language L_T is a language describing the subsets of the states of the model M. The algebraic compiler $\mathcal{MC}$ translates a CTL formula f, to the set of states, S', on which the formula f holds. That is, $\mathcal{MC}(f) = S'$ where $S' = \{s \in S | M, s \models f\}$.

3 Algebraic compilers

3.1 *Σ*–algebras and *Σ*–languages

An *operator scheme* is a tuple $\Sigma = \langle S, Op, \sigma\rangle$ where S is a set of sorts, Op is a set of operator names, and σ is a mapping defining the signatures of the operator names in Op over the sorts in S. That is, $\sigma\colon Op \to S^* \times S$ such that if, for example, s_0, s_1, and s_2 are sorts S and op is an operator name in Op which stands for operations which take an element of sort s_1 and an element of sort s_2 and generates an element of sort s_0, then $\sigma(op) = s_1 \times s_2 \to s_0$.

A Σ–algebra is a family of non–empty sets, called the *carrier sets*, indexed by the sorts S of Σ and a set of Op named operations over the elements of these sets whose signatures are given by σ. There may be many different algebras for the same operator scheme Σ. These algebras are called *similar* and are members of the same *class of similarity*, denoted $\mathcal{C}(\Sigma)$. An interesting member of $\mathcal{C}(\Sigma)$ is the *word* or *term* algebra for Σ. This algebra is parameterized by a set of variables $V = \{V_s\}_{s\in S}$ and is denoted $W_\Sigma(V)$. Its carrier sets contain words formed from the variables of V and operator names of Op and its operators construct such words according to the operations signatures defined by σ [4]. Variables in V are called *generators* and V is thus said to *generate* $W_\Sigma(V)$.

A Σ-language [9] L is defined as the tuple $\langle \mathcal{A}^{sem}, \mathcal{A}^{syn}, \mathcal{L}\colon \mathcal{A}^{sem} \to \mathcal{A}^{syn}\rangle$ where $\mathcal{A}^{sem}$ is a Σ-algebra which is the language semantics, $\mathcal{A}^{syn}$ is a Σ word algebra which is the language syntax, and $\mathcal{L}$ is a partial mapping called the *language learning function* [9, 10]. $\mathcal{L}$ maps semantic constructs in $\mathcal{A}^{sem}$ to their expressions as syntactic constructs in $\mathcal{A}^{syn}$ such that there exists a complementary homomorphism $\mathcal{E}\colon \mathcal{A}^{syn} \to \mathcal{A}^{sem}$ where if $\mathcal{L}(\alpha)$ is defined, then $\mathcal{E}(\mathcal{L}(\alpha)) = \alpha$, $\alpha \in \mathcal{A}^{sem}$. $\mathcal{E}$ is called the *language evaluation function* and maps expressions in $\mathcal{A}^{syn}$ to their semantic constructs in $\mathcal{A}^{sem}$.

CTL as a *Σ*–language CTL can be specified as the Σ–language $L_{ctl} = \langle \mathcal{A}^{sem}_{ctl}, \mathcal{A}^{syn}_{ctl}, \mathcal{L}_{ctl}\rangle$ [12] using the operator scheme $\Sigma_{ctl} = \langle S_{ctl}, Op_{ctl}, \sigma_{ctl}\rangle$ where $S_{ctl} = \{F\}$, the set of sorts containing only one sort for "formula", $Op_{ctl} = \{true, false, not, and, or, ax, ex, au, eu\}$, and σ_{ctl} is defined below:

$$\begin{array}{rlrlrl}
\sigma_{ctl}(true) = & \emptyset \to F & \sigma_{ctl}(not) = & F \to F & \sigma_{ctl}(ax) = & F \to F \\
\sigma_{ctl}(false) = & \emptyset \to F & \sigma_{ctl}(and) = & F \times F \to F & \sigma_{ctl}(ex) = & F \to F \\
 & & \sigma_{ctl}(or) = & F \times F \to F & \sigma_{ctl}(au) = & F \times F \to F \\
 & & & & \sigma_{ctl}(eu) = & F \times F \to F
\end{array}$$

As CTL formulas are written using atomic propositions from a specific model M, the syntax algebra $\mathcal{A}_{ctl}^{syn}$ is parameterized by the set of atomic propositions AP from M and is denoted as $\mathcal{A}_{ctl}^{syn}(AP)$. For example, the formula $\neg(C_1 \wedge C_2)$ shown above has variables C_1 and C_2 from AP of the above model and the $\wedge$ and $\neg$ operations construct the CTL formula (in the syntax word algebra) from these variables. The syntax (word) algebra $\mathcal{A}_{ctl}^{syn}(AP)$ has as its carrier set all possible CTL formulas written using the atomic propositions in AP. The operations of this algebra construct formulas (words, if you like) from variables and operator names. The set of variables AP generates the algebra $\mathcal{A}_{ctl}^{syn}(AP)$.

Just as the syntactic algebra $\mathcal{A}_{ctl}^{syn}(AP)$ is parameterized by the atomic propositions AP of the model M, the semantic algebra $\mathcal{A}_{ctl}^{sem}$ is also parameterized by M in that the carrier set of the semantic algebra $\mathcal{A}_{ctl}^{sem}$ is the power set of the set of states of the model M. The operations in this algebra, while *similar* (that is, having the same signature) to those in $\mathcal{A}_{ctl}^{syn}$, operate on sets, not formulas, since the meaning of a CTL formula is in fact its satisfiability set. Although the operations in the word algebra $\mathcal{A}_{ctl}^{syn}(AP)$ are easily defined as simply concatenating operation names and operands together, the operations in the semantic algebra $\mathcal{A}_{ctl}^{sem}$ are not so simply defined. We will thus name these operations in $\mathcal{A}_{ctl}^{sem}$ and define them individually. The operation names $\{true, false, not, and, or, ax, ex, au, eu\}$ in Op_{ctl} are instantiated in $\mathcal{A}_{ctl}^{sem}$ by the respective operations $\{\mathcal{S}, \emptyset, \mathcal{C}, \cap, \cup, Next_{all}, Next_{some}, lfp_{all}, lfp_{some}\}$ where

- 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 [5]. 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 $Z = \beta \cup (\alpha \cap \{s \in S | (successors(s) \cap \alpha \cap Z) \neq \emptyset\})$ [3].

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

A model as an $\boldsymbol{\Sigma}$–language As the target language of our algebraic model checker, we develop a Σ–language based on sets which is parameterized by a specific model. For a model M, $L_M = \langle \mathcal{A}_M^{sem}, \mathcal{A}_M^{syn}, \mathcal{L}_M \rangle$ using operator scheme $\Sigma_M = \langle S_M, Op_M, \sigma_M \rangle$ where $S_{ctl} = \{Set, Node, Boole\}$, $Op_{ctl} = \{\emptyset, \cup, \cap, \backslash,$ $succ, \subseteq, =, \in,$ "{ }" $\}$, and σ_M is defined below:

$$\begin{array}{rrrr}
\sigma_M(\emptyset) = & \emptyset \to Set & \sigma_M(\subseteq) = & Set \times Set \to Boole \\
\sigma_M(S) = & \emptyset \to Set & \sigma_M(=) = & Set \times Set \to Boole \\
\sigma_M(\cup) = & Set \times Set \to Set & \sigma_M(\in) = & Set \times Node \to Boole
\end{array}$$

$$\sigma_M(\cap) = Set \times Set \to Set \qquad \sigma_M(succ) = Node \to Set$$
$$\sigma_M(\setminus) = Set \times Set \to Set \qquad \sigma_M(\{\ \}) = Node \to Set$$

The operators here are mostly self–descriptive. $\emptyset$ and S generate respectively the empty set and the full set of states S. The binary operators $\cup$, $\cap$, and $\setminus$ are respectively set union, intersection and difference. We also have the subset ($\subseteq$), set equality ($=$), and membership operations ($\in$) and the successor function *succ* and singleton set creation function denoted by $\{\}$. These operators build set expressions in the syntax algebra $\mathcal{A}_M^{syn}$ and sets in the semantic algebra $\mathcal{A}_M^{sem}$.

3.2 Algebraic compilers

An *algebraic compiler* [9,10] $\mathcal{C}\colon L_S \to L_T$ which maps the language $L_S = \langle \mathcal{A}_S^{syn}, \mathcal{A}_S^{sem}, \mathcal{L}_S\rangle$ into the language $L_T = \langle \mathcal{A}_T^{syn}, \mathcal{A}_T^{sem}, \mathcal{L}_T\rangle$ is a pair of (generalized) homomorphisms $(H_{syn}\colon \mathcal{A}_S^{syn} \to \mathcal{A}_T^{syn}, H_{sem}\colon \mathcal{A}_S^{sem} \to \mathcal{A}_T^{sem})$ defined such that the diagram in Figure 2 commutes. In general, the operator schemes of the

$$\begin{tikzcd}
\mathcal{A}_S^{sem} \arrow[r, "\mathcal{L}_S"] \arrow[d, "H_{sem}"] & \mathcal{A}_S^{syn} \arrow[r, "\mathcal{E}_S"] \arrow[d, "H_{syn}"] & \mathcal{A}_S^{sem} \arrow[d, "H_{sem}"] \\
\mathcal{A}_T^{sem} \arrow[r, "\mathcal{L}_T"] & \mathcal{A}_T^{syn} \arrow[r, "\mathcal{E}_T"] & \mathcal{A}_T^{sem}
\end{tikzcd}$$

Fig. 2. An algebraic compiler.

algebras in these two languages may not be similar, as is the case with the operator schemes Σ_{ctl} and Σ_M for the languages L_{ctl} and L_M we intend to use in our model checker. Thus, a homomorphism, which associates a single target algebra operation with each source algebra operation is not possible. Instead, for each source algebra operation, we must build an appropriate operation from several target algebra operations. Such operations are called *derived operations*. Derived operations are written using words from the target word algebra using a set of *meta variables*. We will use subscripted versions of the sort names from the source language operator scheme as meta variables. The word "$\mathcal{S} \setminus F_1$", is a word in the algebra $\mathcal{A}_M^{syn}(\{F_1\})$ which represents the unary derived operation for taking the complement of a set with respect to the full set of states S. The meta variable F_1 is the "formal parameter" of the derived operation. We will associate this derived operation with the CTL operation *not* since given the satisfiability set of a formula f, it will generate the satisfiability set of the formula *not* f.

To define a generalized homomorphism [6] H from algebra $\mathcal{A}_{\Sigma_S}$ with operator scheme $\Sigma_S = \langle S_S, Op_S, \sigma_S\rangle$ to algebra $\mathcal{A}_{\Sigma_T}$ with the possibly dissimilar operator scheme $\Sigma_T = \langle S_T, Op_T, \sigma_T\rangle$ we must define two mappings:

1. a *sort map*, $sm\colon S_S \rightarrow S_T$ which maps source algebra sorts to target algebra sorts. In a generalized homomorphism, an object of sort s of Σ_S will be mapped to an object of sort $sm(s)$ of Σ_T.
2. a *operator map*, $om\colon Op_S \rightarrow W_{\Sigma_T}(S'_S)$, which maps operators in the source algebra to derived operations written as words in the target syntax algebra with meta variables S'_S – the source sorts with subscripts.

The derived operations, which take operands from the target algebra, have the same signatures as their counterparts in the source algebra, and thus we implicitly create an intermediate, hybrid algebra $\mathcal{A}^{ST}_{\Sigma_S}$ which has the same operator scheme Σ_S as the source algebra, but whose carrier sets are populated by values from the target algebra and whose operations are the derived operations defined by the operator map *om*. The generalized homomorphism $H\colon \mathcal{A}_{\Sigma_S} \rightarrow \mathcal{A}_{\Sigma_T}$ is thus the composition of an *embedding* homomorphism from $\mathcal{A}_{\Sigma_S}$ to the intermediate algebra $\mathcal{A}^{ST}_{\Sigma_S}$, $(em\colon \mathcal{A}_{\Sigma_S} \rightarrow \mathcal{A}^{ST}_{\Sigma_S})$ with an identity *injection mapping* from the intermediate algebra to $\mathcal{A}_{\Sigma_T}$, $(im\colon \mathcal{A}^{ST}_{\Sigma_S} \rightarrow \mathcal{A}_{\Sigma_T})$ [9, 17]. The mapping *im* is an identity mapping that maps elements in sort $s, s \in S_S$ in $\mathcal{A}^{ST}_{\Sigma_S}$ to the same value in sort $sm(s) \in S_T$ in $\mathcal{A}_{\Sigma_T}$. Thus $H = im \circ em$. Since both the syntax and semantic generalized homomorphisms of Figure 2 are implemented in this manner, the intermediate algebras form an intermediate Σ–language and thus, the diagram of Figure 2 becomes the commutative diagram in Figure 3.

$$\begin{tikzcd}
\mathcal{A}_S^{sem} \arrow[r, "\mathcal{L}_S"] \arrow[d, "em_{sem}"] & \mathcal{A}_S^{syn} \arrow[r, "\mathcal{E}_S"] \arrow[d, "em_{syn}"] & \mathcal{A}_S^{sem} \arrow[d, "em_{sem}"] \\
\mathcal{A}_{ST}^{sem} \arrow[r, "\mathcal{L}_{ST}"] \arrow[d, "im_{sem}"] & \mathcal{A}_{ST}^{syn} \arrow[r, "\mathcal{E}_{ST}"] \arrow[d, "im_{syn}"] & \mathcal{A}_{ST}^{sem} \arrow[d, "im_{sem}"] \\
\mathcal{A}_T^{sem} \arrow[r, "\mathcal{L}_T"] & \mathcal{A}_T^{syn} \arrow[r, "\mathcal{E}_T"] & \mathcal{A}_T^{sem}
\end{tikzcd}$$

Fig. 3. An algebraic compiler with the intermediate language displayed.

Given a mapping g which maps generators of the source algebra into the target algebra, $g = \{g_s\colon s \rightarrow sm(s)\}_{s \in S_s}$, g can be uniquely extended to a homomorphism $H\colon \mathcal{A}_{\Sigma_S} \rightarrow \mathcal{A}_{\Sigma_T}$ [6, 9]. The algorithm for implementing a generalized homomorphism from a Σ_S algebra generated by $G = \{g_s\}_{s \in S_S}$ is

```
$H(a) =$ **if** $a \in g_s$ for some $s \in S_S$ **then** $g_s(a)$
             **else if** $a = f(a_1, a_2, \ldots, a_n)$ for some $f \in Op_S$
                 **then** $om(f)(h(a_1), h(a_2), \ldots, h(a_n))$.
```

This is all made clear by examining it in the context of our model checker as an algebraic compiler. For starters, the sort map *sm* simply maps the sort F in

Σ_{ctl} to the sort Set in Σ_M. The generators G are the set of atomic propositions, $G_F = AP$, and g_F is the function P which maps atomic propositions to their satisfiability sets. What is left then, is to define the operator map om which maps CTL operators in Op_{ctl} to derived operations over satisfiability sets. We saw above how the word "$\mathcal{S} \setminus F_1$" could be used to define the derived operation for the CTL operation $not\colon F_1 \rightarrow F_0$. The use of the indexed sort name F (F_1) as the meta variable is to show the correspondence between the parameters of the source and derived operations. The subscripts are used to distinguish between multiple parameters of the same sort, different sorts will have different names.

Consider now, the CTL operation ax. We cannot write a correct derived operation using only the operators from the target language. We need additional constructs with which to compose a derived operation. It is at this point that we can begin to speak of *meta languages* used in the specification of algebraic compilers instead of just *meta variables*. By introducing some functional language constructs into the language in which we write derived operations, we may like to write the derived operation for ax as

$$om(ax\colon F_1 \rightarrow F_0) = \text{filter } (\lambda \text{ n . } succ(\text{n}) \subseteq F_1 \text{) } S$$

where "filter" is a generic operation which applies a boolean function (given by the λ–expression) to each element of a container type, returning a similar container type which contains only those elements from the original which, when provided to the boolean function, generate a value of *true*. Where F_1 is the satisfiability set of a CTL formula f, the derived operation denoted by this term will compute the satisfiability set of the CTL formula $ax\ f$, by extracting from S, those states that satisfy the condition that all of their successors satisfy f.

Instead of extending the target algebra with these operations, we show in the following section how a meta language containing these constructs can be used in conjunction with the target language to write the appropriate derived operations. The advantage of keeping the meta language separate from the target language is that we can populate an algebraic language processing environment with several reusable meta languages which a language designer may use to build translators.

3.3 Evaluation of derived operations

Derived operations are specified by words from the target language syntax algebra $\mathcal{A}_T^{syn}(S'_S)$ over a subscripted set of meta variables from the source signature set of sorts S_S. In Figure 3, the same words from $\mathcal{A}_T^{syn}(S'_S)$ are used to define the operations of the syntax algebra $\mathcal{A}_{ST}^{syn}$ and the semantics algebra $\mathcal{A}_{ST}^{sem}$. Thus, we could build a generalized homomorphism $h'\colon \mathcal{A}_S^{syn} \rightarrow \mathcal{A}_{ST}^{sem}$ which maps words in $\mathcal{A}_S^{syn}$ directly to values in $\mathcal{A}_{ST}^{sem}$. Thus, h' is the composition of the embedding morphism $em_{syn}\colon \mathcal{A}_S^{syn} \rightarrow \mathcal{A}_{ST}^{sem}$ and the L_{ST} evaluation function $\mathcal{E}_{ST}$, i.e. $h' = em_{syn} \circ \mathcal{E}_{ST}$. In the case of our model checker, such a homomorphism would map CTL formulas directly to their satisfiability sets in the intermediate semantic algebra. For efficiency reasons this may be desirable and is often the way we will actually implement model checkers as algebraic compilers.

4 Meta languages in algebraic compilers

A meta language L_{ML} used in an algebraic compiler is essentially a parameterized Σ–language. Like all Σ–languages, it has an operator scheme $\Sigma_{ML} = \langle S_{ML}, Op_{ML}, \sigma_{ML} \rangle$ where S_{ML} and Op_{ML} are a set of sorts and operator names as seen above. The signatures of these operator names, however, may include parameters as well as sorts from S_{ML}. That is, $\sigma_{ML}\colon Op_{ML} \to PS^*_{ML} \times PS_{ML}$, where PS_{ML} where $PS_{ML} = S_{ML} \cup Param$, $Param$ is a set of parameter names. The meta language has additional constructs that we will use to write the derived operations of the algebraic compiler. In the functional instance of the model checker, these meta language operations will include the "filter" and λ–expression operators we saw above, in the "imperative" instance, the meta language constructs will include *if* and *while* statements, assignment statements, and a *for each* loop operation. These meta operations, in combination with the target language operations of set intersection, union, membership, etc., are used to write the derived operations specifying the model checker.

To write derived operations using meta (L_{ML}) and target (L_T) language operations, an *instantiation* of the meta language is created (by the language processing environment) from these two languages. This language is denoted $L_{ML^T} = \langle \mathcal{A}^{sem}_{ML^T}, \mathcal{A}^{syn}_{ML^T}, \mathcal{L}_{ML^T} \rangle$ with operator scheme Σ_{ML^T}. To instantiate a meta language the following tasks must be performed:

1. Instantiate the operator scheme Σ_{ML^T}. $\Sigma_{ML^T} = \langle S_{ML^T}, Op_{ML^T}, \sigma_{ML^T} \rangle$ where the set of sorts S_{ML^T} is the union of the meta and target sorts $S_{ML} \cup S_T$, the operator names Op_{ML^T} are the union of meta and target operator names $Op_{ML} \cup Op_T$, and signatures in σ_{ML^T} are created by replacing parameters in σ_{ML} signatures with sort names in S_T and adding the target languages signatures in σ_T. In our model checker, the target language sorts $Node$ and Set replace the parameters in the meta language signatures.
2. Instantiate the syntax algebra $\mathcal{A}^{syn}_{ML^T}$. We must instantiate the operations for the syntax algebra, but these are can be automatically constructed using a prefix format for these "word constructing" operations.
3. Instantiate the semantic algebra $\mathcal{A}^{sem}_{ML^T}$. We must also instantiate the operations of this algebra. Either they are explicitly constructed for the new types, a kind of ad hoc polymorphism, or, preferably, the existing meta language operations are *generic* (polymorphic or polytypic) [2] and can thus automatically work on the data-types from the target algebra.

Derived operations for the generalized homomorphism are now written in $\mathcal{A}^{syn}_{ML^T}(S'_S)$, the instantiated meta language word algebra with meta variables S'_S, instead of the syntax algebra $\mathcal{A}^{syn}_{ST}(S'_S)$ of the intermediate hybrid language L_{ST} as done before. Thus, the operator map om used in defining the generalized homomorphism has the signature $om : Op_S \to \mathcal{A}^{syn}_{ML^T}(S'_S)$. The sort map sm is the same as before, so that target images of source language constructs are still objects of sorts in the target language, not the meta language.

When building such an algebraic compiler the hybrid intermediate language L_{ST} from Figure 3 is replaced by the hybrid intermediate language $L_{SML^T} =$

$\langle \mathcal{A}^{sem}_{SML^T}, \mathcal{A}^{syn}_{SML^T}, \mathcal{L}_{SML^T} \rangle$ as shown in Figure 4. Like L_{ST}, this language has the same operator scheme Σ_S as the source language, but has operations built using the operations from L_{ML^T}. The embedding morphisms em_{syn} and em_{sem} in Figure 4 are computed in the same manner as those in Figure 3. We also add an extra pair of identity injection mappings between L_{SML^T} and L_{ML^T}.

$$\begin{tikzcd}
\mathcal{A}^{sem}_{S} \arrow[r, "\mathcal{L}_S"] \arrow[d, "em_{sem}"] & \mathcal{A}^{syn}_{S} \arrow[r, "\mathcal{E}_S"] \arrow[d, "em_{syn}"] & \mathcal{A}^{sem}_{S} \arrow[d, "em_{sem}"] \\
\mathcal{A}^{sem}_{SML^T} \arrow[r, "\mathcal{L}_{SML^T}"] \arrow[d, "in2_{sem}"] & \mathcal{A}^{syn}_{SML^T} \arrow[r, "\mathcal{E}_{SML^T}"] \arrow[d, "in2_{syn}"] & \mathcal{A}^{sem}_{SML^T} \arrow[d, "in2_{sem}"] \\
\mathcal{A}^{sem}_{ML^T} \arrow[r, "\mathcal{L}_{ML^T}"] \arrow[d, "in1_{sem}"] & \mathcal{A}^{syn}_{ML^T} \arrow[r, "\mathcal{E}_{ML^T}"] \arrow[d, "in1_{syn}"] & \mathcal{A}^{sem}_{ML^T} \arrow[d, "in1_{sem}"] \\
\mathcal{A}^{sem}_{T} \arrow[r, "\mathcal{L}_T"] & \mathcal{A}^{syn}_{T} \arrow[r, "\mathcal{E}_T"] & \mathcal{A}^{sem}_{T}
\end{tikzcd}$$

Fig. 4. An algebraic compiler with a meta language level.

Just as the intermediate hybrid language L_{ST} in Figure 3 is automatically created, so it $L_{SML^T} = \langle \mathcal{A}^{sem}_{SML^T}, \mathcal{A}^{syn}_{SML^T}, \mathcal{L}_{SML^T} \rangle$. However, we do need to explicitly create the meta language L_{ML^T}. But, this makes sense, whereas before we specified the source and target language of the algebraic compiler and wrote derived operations in the target syntax algebra with meta variables, we must now specify the meta language we wish to use as well. The derived operations are then written in the instantiated meta language syntax algebra. An appropriate set of algebraic language processing tools can automatically instantiate the meta language, provided the existing meta language operations are generic, but we must at least specify which meta language is to be composed with the selected target language in order to write derived operations and generate the algebraic compiler from these specifications.

4.1 A functional meta language

As alluded to above, we can use a functional meta language in specifying our algebraic model checker $MC\colon L_{ctl} \to L_M$. This allows us write derived operations for the temporal logic operators ax, ex, au, and eu using functional language constructs and thus provide concise specifications for our model checker. Although a full functional meta language would have many higher order functions, like map and $fold$, we only describe here the operations which are used in our algebraic specification. We do however use λ expressions and higher order functions $filter$, $limit$ and $iterate$ which are defined below.

Our functional meta language $L_{FM} = \langle\ \mathcal{A}^{sem}_{FM},\ \mathcal{A}^{syn}_{FM},\ \mathcal{L}_{FM}\rangle$ has operator scheme $\Sigma_{FM} = \langle S_{FM} = \{Boole, Var, Func(\rightarrow), List([\ ])\},\ Op_{FM} = \{not,\ and,\ filter,\ \lambda,\ limit,\ iterate\}, \sigma_{FM}\rangle$, where σ_{FM} is defined below:

$$\begin{aligned}
\sigma_{FM}(not) &= Boole && \rightarrow Boole\\
\sigma_{FM}(and) &= Boole \times Boole && \rightarrow Boole\\
\sigma_{FM}(filter) &= (a \rightarrow Boole) \times b && \rightarrow b\\
\sigma_{FM}(\lambda) &= Var_b \times a && \rightarrow (b \rightarrow a)\\
\sigma_{FM}(limit) &= [a] && \rightarrow a\\
\sigma_{FM}(iterate) &= (a \rightarrow a) \times a && \rightarrow [a]
\end{aligned}$$

The *Boole* sort is for boolean variables, *Var* for variables used in λ–expressions, *Func* for functions between two types, denoted $a \rightarrow b$ for respective source and target types a and b, and *List*, denoted $[a]$ for lists of elements of type a. *filter* is a generic operation which applies a boolean function to each element (parameter a) of a container type (parameter b), and constructs the container type with only those original elements which evaluate to *true* under the boolean function. λ is the operation for creating functions from λ–expressions. The parameter a in this signature represents an expression of type a with a free variable of type b which when combined with a variable of type b, (Var_b) generates a function of type $b \rightarrow a$. *limit* is a function which lazily evaluates a list of elements, returning the first element in the list which is followed by a element of the same value (limit $[1, 2, 3, 3, ...]$ evaluates to 3). *iterate* is also lazy and repeatedly applies a unary function first using a given initial value and then to the value returned from the previous application That is, $iterate\ f\ x = x\ cons\ (iterate\ f\ (f\ x))$ (for example $iterate\ inc\ 3 = [3, 4, 5, 6, ...]$).

We can instantiate this meta language with the model language L_M by writing new operator signatures by replacing the parameters a and b in σ_{FM} with sort names *Set* and *Node* from the operator scheme Σ_M of L_M. Since the *filter* operation is generic [2], we do not need to explicitly implement versions of this function for sets.

4.2 An imperative meta language

We can similarly design an imperative meta language $L_{IM} = \langle\ \mathcal{A}^{sem}_{IM},\ \mathcal{A}^{syn}_{IM},\ \mathcal{L}_{IM}\rangle$ that has operator scheme $\Sigma_{IM} = \langle S_{IM}, Op_{IM}, \sigma_{IM}\rangle$. The sort set contains sorts $S_{IM} = \{Expr, Stmt, StmtList, Var, Boole\}$ for expressions, statements, statement lists, etc., as are familiar in imperative languages. A set of operators Op_{IM} would thus include the set $\{if, while, assign, block, for\ each, not, and, ...\}$. These operator's signatures and others as defined by σ_{IM} are shown below:

$$\begin{aligned}
\sigma_{IM}(not) &= Boole && \rightarrow Boole\\
\sigma_{IM}(and) &= Boole \times Boole && \rightarrow Boole\\
\sigma_{IM}(foreach) &= Var \times Expr \times Stmt && \rightarrow Stmt\\
\sigma_{IM}(if) &= Boole \times Stmt && \rightarrow Stmt\\
\sigma_{IM}(while) &= Boole \times Stmt && \rightarrow Stmt\\
\sigma_{IM}(assign) &= Var \times Expr && \rightarrow Stmt\\
\sigma_{IM}(block) &= StmtList && \rightarrow Stmt
\end{aligned}$$

$$\begin{aligned}
\sigma_{IM}(list_1) &= Stmt &&\to StmtList\\
\sigma_{IM}(list_2) &= StmtList \times Stmt &&\to StmtList\\
\sigma_{IM}(expr_1) &= Expr &&\to a\\
\sigma_{IM}(expr_2) &= Expr &&\to Boole\\
\sigma_{IM}(valof) &= Stmt &&\to Expr
\end{aligned}$$

The familiar imperative language operations are present here. Of interest is the generic $for\ each$ operation which will iterate through all elements of a container type, and perform some statement for each element and the $valof$ operation which embeds statements in expressions using the value of the last assignment.

5 Model checker specification

In this section we can show the specifications for the algebraic model checker using the functional and imperative meta languages. We will write the translation specifications for each CTL operation $op \in Op_{ctl}$, by writing the signature of the operation, $\sigma_{ctl}(op)$, followed by its derived operation in the target, $om(op)$, but we will drop the *om* for convenience. The operation's signatures are written with the output sort of each operation to the left and the operation name split between the input sorts in a BNF notation. (In fact, some algebraic tools like TICS [11] use this specification to generate a parser for the source language.) The meta variables used in the derived operations are indexed source language sorts found in the source operation signature. In the derived operations, a meta variable for an input sort represents the target image of the corresponding source language component. These specifications are processed by an algebraic language processing environment to automatically generate the model checker [12, 13].

5.1 Functional meta language specification

The functional version of the algebraic model checker maps CTL formulas in $\mathcal{A}^{syn}_{ctl}(AP)$ to their satisfiability sets. For the non–temporal operators in L_{ctl} we have straightforward derived operations shown below:

$F_0 ::= true$	$F_0 ::= false$	$F_0 ::= not\ F_1$	$F_0 ::= F_1\ and\ F_2$
S	$\emptyset$	$S \setminus F_1$	$F_1 \cap F_2$

The operation *true* has the derived operation S (shown directly below it) indicating that the satisfiability set of *true* is the full set of states S in the model M; $false$ has derived operation $\emptyset$ indicating that the satisfiability set of $false$ is the empty set. The derived operation associated with *not* shows that the satisfiability of *not* f is the set difference of S and the satisfiability set of f, denoted by the sort name F_1. Similarly, *and* is defined by the intersection of the satisfiability sets of the two sub formulas, respectively denoted F_1 and F_2.

In the derived operation for *ax*, seen below, we see the use of some meta language constructs. Here, we define the satisfiability set of *ax* f by filtering the set of states by a function which selects only those nodes such that all of their successors are in the satisfiability set of f.

$F_0 ::= ax\ F_1$ filter (λ n . $succ$(n) $\subseteq F_1$) S

The derived operation for *au* is similar, but uses the *limit* and *iterate* operations to implement a type of least fixed point operator of the function specified by the λ–expression.

$F_0 ::= a\ [\ F_1\ u\ F_2\ ]$
limit (iterate (λ z . z $\cup$ (filter (λ n . ($succs$(n) $\subseteq$ z)) F_1)) F_2)

The atomic propositions, specified as variables AP in $\mathcal{A}^{syn}_{ctl}(AP)$, are mapped to their satisfiability set by P, the model labeling function.

$F_0 ::= p$ $P(p)$

5.2 Imperative meta language specification

Since the non–temporal CTL operators do not use any meta language constructs in their derived operations, they are the same here as in the functional specification. Thus, we show only the temporal operators *ax* and *au*. We use an additional meta variables $\$temp_i$ in these derived operations which are replaced by a new temporary variables for each use of a derived operation.

```
$F_0 ::= ax\ F_1$
 valof {
   $\$temp := \emptyset$
   for each n in $S$
     if ( $succ$(n) $\subseteq$ ($F_1$) then
       $\$temp := \$temp \cup \{$ n $\}$
   $F_0 := \$temp$ }
```

```
$F_0 ::= a\ [\ F_1\ u\ F_2\ ]$
 valof {
   $\$temp_1 := \emptyset$; $\$temp_2 := F_2$ ;
   while ( not $\$temp_1 == \$temp_2$ ) {
     $\$\mathrm{temp}_1 := \$temp_2$ ;
     for each n in $F_1$ do
       if ($succ$(n) $\subseteq \$temp_1$ ) then
         $\$temp_2 := \$temp_2 \cup \{n\}$ }
   $F_0 := \$\mathrm{temp}_1$ }
```

These derived operations are the imperative versions of the functional derived operations given above in Section 5.1. Here, the *while* and *for each* operators are used to implement a least fixed point operation to compute satisfiability sets.

6 Comments and future work

The meta languages described here are just the required subsets of general purpose meta languages which would populate an algebraic language processing environment. Meta languages should be reusable components in such an environment so that algebraic compiler designers can choose from a collection of existing meta languages in which to write their translator specifications. A well–stocked environment would have functional and imperative style meta languages giving the language designer some choice based on personal preference of language style.

More importantly, however, we would also expect this environment to contain *domain specific meta languages* [18] with specialized constructs to address issues found in specific domains commonly encountered in language processing as well as other domains, such as temporal logic model checking, which also have solutions as algebraic compilers. Traditional language processing tasks with specific domains include type checking, optimization and parallelization, and code generation. In a type checker, for example, the target algebras would have operators for the base types and type constructors and carrier sets containing types or type expressions. A domain specific meta language for type checking which has specific constructs for managing symbol tables and environments would be helpful to the implementor and reusable in different compilers. In the case of the model checker, a domain specific meta language would include a least fixed point operator, since this domain would make good use of such a construct.

We opened this paper with a mention of attribute grammars and comment here on meta languages within attribute grammars since they take a slightly different form than in algebraic compilers. Algebraic compilers rely on an explicit definition of the target language and use target language operations for writing derived operations. These operations thus provide a starting point for adding meta language features. Attribute grammars, to their detriment, make no explicit mention of the target language and thus do not have a set of target language operations to provide as a starting point for writing semantic functions for defining attribute values. Instead, they provide a single general purpose language for writing semantic functions. This language doesn't suffer the expressiveness problems we saw above, but it does lock the user into a single "meta language" for defining attribute values. We have thus argued [18] that a choice of domain specific meta languages in attribute grammars is also desirable for many of the same reasons as they are beneficial in algebraic compilers.

We are pursuing this work in an effort to find appropriate meta languages for defining language constructs for the Intentional Programming (IP) [15] system under development at Microsoft. IP is an extensible programming environment which allows programmers to define their own language constructs, called *intentions*, and add them to their programming environment. We are interested in exploring different meta languages, in the broad sense of the term, for defining such intentions. Since the same domains of type checking, optimization, code generation, etc., are encountered in IP, domain specific meta languages will be useful in this system as well. They are especially important here since appropriate domain specific meta languages raise the level of abstraction in which the intention designer works and will thus make designing intentions a more reasonable process that experienced programmers could perform to create their own language extensions. To experiment with different meta languages, we are currently developing a set of lightweight prototype tools using algebraic compilers and attribute grammars which use domain specific meta languages.

Our choice of model checking as an example isn't as esoteric as it may appear. Model checking has been used to perform data flow analysis on program control and data flow graphs [16] and to find optimization and parallelization opportunities in program dependency graphs [14]. In both cases, temporal logic acts as a specification language for certain patterns in a graph representation of the program which are found by a model checker. Thus, temporal logic does have applications as a domain specific meta language in algebraic compilers, attribute grammars and IP.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
2. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In *LNCS*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.
3. 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.
4. P.M. Cohn. *Universal Algebra*. Reidel, London, 1981.
5. M. Gordon. *Programming Language Theory and its Implementation*. Prentice Hall, 1988.
6. P.J. Higgins. Algebras with a scheme of operators. *Mathematische Nachrichten*, 27:115–132, 1963/64.
7. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in 5(2):95–96, 1971.
8. S. Kripke. Semantical analysis of modal logic i: Normal modal propositional calculi. *Zeitschrift f. Math. Logik und Grundlagen d. Math.*, 9, 1963.
9. T. Rus. Algebraic construction of compilers. *Theoretical Computer Science*, 90:271–308, 1991.
10. T. Rus. Algebraic processing of programming languages. *Theoretical Computer Science*, 199(1):105–143, 1998.
11. T. Rus, T. Halverson, E. Van Wyk, and R. Kooima. An algebraic language processing environment. In Michael Johnson, editor, *LNCS 1349*, pages 581–585, Sydney, Australia, 1997.
12. T. Rus and E. Van Wyk. Algebraic implementation of model checking algorithms. In *Third AMAST Workshop on Real-Time Systems, Proceedings*, pages 267–279, March 6 1996. Available from URL:
`http://www.comlab.ox.ac.uk/oucl/work/eric.van.wyk/`
13. 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.
14. T. Rus and E. Van Wyk. Using model checking in a parallelizing compiler. *Parallel Processing Letters*, 8(4):459–471, 1998.
15. C. Simonyi. Intentional programming: Innovation in the legacy age. Presented at IFIP Working group 2.1., 1996. Available from URL:
`http://www.research.microsoft.com/research/ip/`
16. B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21:115–139, 1993.
17. E. Van Wyk. *Semantic Processing by Macro Processors*. PhD thesis, The University of Iowa, Iowa City, Iowa, 52240 USA, July 1998.
18. E. Van Wyk. Domain specific meta languages. In *ACM Symposium on Applied Computing*, March 19–21 2000.