A Foundation for Flow-Based Program Matching Using Temporal Logic and Model Checking

Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia Lawall, Gilles Muller

A Foundation for Flow-Based Program Matching

Using Temporal Logic and Model Checking

Julien Brunel
DIKU, University of Copenhagen, Denmark
brunel@diku.dk

Damien Doligez
INRIA, Gallium Project, France
damien.doligez@inria.fr

René Rydhof Hansen
Department of Computer Science
Aalborg University, Denmark
rrh@cs.aau.dk

Julia L. Lawall
DIKU, University of Copenhagen, Denmark
julia@diku.dk

Gilles Muller
École des Mines de Nantes, France
Gilles.Muller@emn.fr

Research report 08/2/INFO, Ecole des Mines de Nantes
July 16, 2008

Reasoning about program control-flow paths is an important functionality of a number of recent program matching languages and associated searching and transformation tools. Temporal logic provides a well-defined means of expressing properties of control-flow paths in programs, and indeed an extension of the temporal logic CTL has been applied to the problem of specifying and verifying the transformations commonly performed by optimizing compilers. Nevertheless, in developing the Coccinelle program transformation tool for performing Linux collateral evolutions in systems code, we have found that existing variants of CTL do not adequately support rules that transform subterms other than the ones matching an entire formula. Being able to transform any of the subterms of a matched term seems essential in the domain targeted by Coccinelle.

In this technical report, we propose an extension to CTL named CTL-VW (CTL with variables and witnesses) that is a suitable basis for the semantics and implementation of the Coccinelle's program matching language. Our extension to CTL includes existential quantification over program fragments, which allows metavariables in the program matching language to range over different values within different control-flow paths, and a notion of witnesses that record such existential bindings for use in the subsequent program transformation process. We formalize CTL-VW and describe its use in the context of Coccinelle. We then assess the performance of the approach in practice, using a transformation rule that fixes several reference count bugs in Linux code.

1. Introduction

Program matching is the process of searching within the source code of a program for code fragments matching a given pattern, described using some language. Recently, program matching languages that combine descriptions of code fragments with information about the control-flow paths between them have been found useful in specifying rules for program manipulation tasks such as compiler optimizations [16, 17], bug finding [10], refactorings [23], and evolution [20].

In recent work, we have developed the transformation system Coccinelle [20], which provides the language SmPL (Semantic Patch Language) for specifying desired matches and transformations. Coccinelle is targeted towards performing *collateral evolutions* in systems code. Such evolutions comprise the changes that are needed in client code in response to evolutions in library APIs, and may include modifications such as renaming a function, adding a function argument whose value is somehow context-dependent, and reorganizing a data structure. Collateral evolutions may involve fragments of code that are scattered throughout a function or a file, such as a function call and its corresponding error-handling code, and thus it is often necessary to take into account the control-flow relationships between the relevant code fragments. Beyond collateral evolutions, we have also found Coccinelle useful for finding and fixing bugs in the use of API functions [18], for which control-flow relationships must also be taken into account.

In developing Coccinelle, it was necessary to choose a foundation on which to base the matching process. In the context of specifying compiler optimizations, Lacey and De Moor have observed that the temporal logic CTL can provide a convenient foundation for the semantics of program matching languages that take control-flow paths into account, because it is designed for reasoning about paths [16]. In this setting, a pattern is compiled into a formula of the logic and a program is translated into its control-flow graph. Model checking [7, 12] is then used to match the formula against the control-flow graph, thus identifying where the pattern matches in the program. To support the specification of compiler optimizations, Lacey and De Moor extended CTL first with predicates over free variables, producing the logic CTL-FV [17], and then with predicates over variables that may be existentially quantified, producing a logic that we refer to as CTL-V [15]. The former allows collecting information about the program during the matching process, via variable bindings, and imposing constraints on the allowed subterms when a given variable is matched more than once. The latter enables localizing collected information and imposed constraints to within individual subsets of the possible control-flow paths, such as the different branches of a conditional statement.

The needs of Coccinelle, however, have made apparent some inadequacies of CTL-V as the back end of a program matching language. While the use of existentially quantified variables allows variables to have different values within different control flow paths, the semantics of CTL-V does not provide a means of retrieving the values of such variables for use during subsequent transformation. Furthermore, CTL-V model checking, like standard CTL model checking, only provides information about the state at which the entire formula is satisfied, and not about the results for subformulas that contribute to a successful match. Information about such intermediate results, however, is important when a single formula can describe transformations of subterms, as is needed for Coccinelle.

In this paper, we present a variant of CTL named CTL-VW (CTL with variables and witnesses), that addresses the above needs for program matching. The syntax of CTL-VW is the same as that of CTL-V, extending that of CTL with *predicates defined over metavariables* that can be existentially quantified over the set of program fragments. The semantics of CTL-VW maintains a collection of *witnesses* that record the states and bindings that satisfy existentially quantified subformulas. We exploit these witnesses both to record variable bindings and to identify states at which transformation should take place. In practice, we have used CTL-VW as the basis of the implementation of Coccinelle. We have used Coccinelle to implement over 60 collateral evolutions, affecting in total over 5800 files in various recent versions of Linux [20]. We have also used Coccinelle to find over 45 bugs in Linux code.

The contributions of this paper are as follows:

- We present the semantics of CTL-VW. We show that this semantics is a conservative extension of that of CTL-V.
- We present in two steps a model checking algorithm for CTL-VW. In the first step, we extend the CTL model checking algorithm with environments, producing a new model checking algorithm for both CTL-FV and CTL-V. In the second step, we further extend this model checking algorithm with witnesses, for CTL-VW.
- We show that the model checking algorithm for CTL-VW is sound and complete with respect to the semantics of CTL-VW.
- We show how to translate the core of Coccinelle's program matching language SmPL into CTL-VW. This core is sufficient to express a semantic patch for finding and fixing some bugs in the use of reference counts in Linux code. The resulting corrections have been validated by Linux experts and accepted into the Linux kernel.[1] Our examples show that the CTL-VW based implementation of Coccinelle is efficient enough to be usable on a 1.4GHz laptop.

The rest of the paper is organized as follows. Section 2 briefly presents Coccinelle in terms of a real example that highlights the above-cited requirements. Section 3 describes

[1] `http://www.emn.fr/x-info/coccinelle/#impact`

CTL and some variants that are precursors to our work. Section 4 describes our first contribution: a bottom-up model checking algorithm for CTL-V. Section 5 introduces our second contribution: CTL-VW, including its syntax, semantics, and a model checking algorithm. Section 6 describes how CTL-VW is used in the context of Coccinelle. Finally, Section 7 presents related work and Section 8 concludes.

2. Overview of Coccinelle

Our motivation for this work was to be able to use CTL as a foundation for the semantics and implementation of SmPL, the program matching and transformation language of Coccinelle [20]. The syntax of SmPL is derived from that of a Linux patch file [19], which is a notation familiar to Linux programmers. Unlike a standard patch, however, which is text-based, a semantic patch takes into account the semantics of the matched code, in particular its intraprocedural control-flow. From the point of view of CTL, the relevant features of a semantic patch are that it may describe a complex region of code, specified in terms of program fragments that should be connected by control-flow paths, and that it may be needed to specify transformations at any point within the described region. Furthermore, the region may involve multiple control-flow paths, *e.g.*, due to conditionals, and different control-flow path may involve different computations, each of which may be relevant to the transformation process.

In this section, we present the SmPL language via a short example that illustrates the above issues. We defer a more formal presentation of the code language of SmPL to Section 6, where we show how to define its semantics by translation into the CTL-VW logic developed in the next three sections.

2.1 A simple SmPL sample

Managing reference counts is a common source of errors in C code. In particular, we have observed that Linux error handling code sometimes does not appropriately decrement reference counts for objects acquired in the current function. Figure 1 shows a semantic patch type_ref for correcting such problems involving the function `of_find_node_by_type`. This function increases a reference count that should subsequently be decremented by calling the function `of_node_put`. The semantic patch in Figure 1 inserts a call to `of_node_put` before a return in error handling code (indicated by a return of a negative value), when there was no such call previously, and when it is not possible that the value returned by `of_find_node_by_type` has been saved in a more permanent manner.

The type_ref semantic patch consists of a single rule, which first declares a collection of metavariables and then defines a transformation specification. The metavariables are designated according to the kind of terms they can match, such as a statement, an identifier, or an expression (line 2). An expression metavariable can be further constrained by its type (line 3). The transformation specification essentially has

```
@type_ref@                                                                    1
statement S; identifier f1,f2; expression E1,E2; constant C;                 2
struct device_node *n; struct device_node *n1; struct device_node *n2;       3
@@                                                                            4
                                                                              5
n = of_find_node_by_type(...)                                                 6
...                                                                           7
if (!n) S                                                                     8
... when != of_node_put(n)                                                    9
    when != n1 = f1(n,...)                                                    10
    when != E1 = n                                                            11
(                                                                             12
+ of_node_put(n);                                                            13
  return -C;                                                                  14
| of_node_put(n);                                                           15
| n2 = f2(n,...)                                                            16
| E2 = n                                                                    17
| return ...;                                                               18
)                                                                             19
```

Figure 1. The semantic patch type_ref written in simplified SmPL

the form of C code, except that lines to remove are annotated with - in the first column, and lines to add are annotated with + (line 13). A transformation specification can also use *dots*, "`...`" (line 7), describing an arbitrary sequence of instructions within a control-flow path. Dots may be modified with a `when` clause (lines 9-11), indicating a pattern that should not occur anywhere within the matched sequence. Finally, lines 12-19 specify a disjunction of patterns, of the form (pat_1 | ... | pat_n).

Full SmPL, as implemented in Coccinelle, provides many features not illustrated by this example, such as multiple rules, subpatterns that match 0 or more times, dots within arbitrary subterms, and the ability to use both universal and existential path quantification [20]. All of these features can be encoded similarly to the basic strategies presented in Section 6.

2.2 Assessment

Our goal is to encode the semantic patch of Figure 1 as a single formula of a CTL-like logic. For this, the main issues are 1) to manage the metavariables *n*x, *C*, etc., 2) to allow metavariables to match different terms along different control-flow paths, as *e.g.*, the return value *C* may be different at each return site when there are multiple returns under conditionals, 3) to record the various bindings of the metavariables within the different control-flow paths, as *e.g.*, the binding of *C* may be needed to transform the enclosing `return`, and 4) to record the sites where transformation is needed. For the last point, in our case, the complete formula representing the semantic patch would match at a call to `of_find_node_by_type`, but transformation is required at the various matches of the subformula representing `return -C;`. A semantic patch may indeed specify a transformation anywhere within the matched term. Among these requirements, CTL-FV provides only a representation of metavariables, and CTL-V additionally allows metavariables to match different terms along different control-flow paths. We thus develop the logic

CTL-VW that permits an encoding of a semantic patches that addresses the remaining requirements, *i.e.*, 3) and 4).

3. Background

In this section, we present the theoretical background for the work in the present paper. First, we briefly review the syntax, semantics, and algorithmic implementation of the well-known Computational Tree Logic (CTL) [7]. Next, we present a variant of CTL called CTL-FV (CTL with free variables) used by Lacey *et al.* to show the correctness of some classical compiler optimizations [17]. Finally, we present CTL-V (CTL with variables), a CTL variant with existentially quantified variables, and formally define its semantics.

3.1 Computational Tree Logic

Computational Tree Logic is a temporal logic based on the notion of branching time [7]. It has been implemented in a variety of model checkers and used to model check properties ranging from hardware verification [11, 14] to program analysis [22].

In what follows we briefly review the syntax and semantics of CTL without going into detail. For a textbook treatment, see [12]. CTL is a logic for reasoning about states and the paths between them. Its syntax is as follows:

$$\begin{array}{rcl} \phi & ::= & p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \\ & & \mathsf{AX}\phi \mid \mathsf{EX}\phi \mid \mathsf{A}[\phi\,\mathsf{U}\,\phi] \mid \mathsf{E}[\phi\,\mathsf{U}\,\phi] \end{array}$$

Intuitively, the formulas p (propositions), $\phi \wedge \phi$, $\phi \vee \phi$, and $\neg\phi$ are the same as in propositional logic, and permit reasoning about the properties of a given state. The remaining formulas permit reasoning about paths. $\mathsf{AX}\phi$ holds if all the successors of the current state satisfy ϕ, while $\mathsf{EX}\phi$ holds if at least one successor of the current state satisfies ϕ. $\mathsf{A}[\phi_1\,\mathsf{U}\,\phi_2]$ expresses that along all paths there is some state where ϕ_2 holds, and requires that ϕ_1 hold at all preceding states. $\mathsf{E}[\phi_1\,\mathsf{U}\,\phi_2]$ is similar, except that there need be only one path with these properties. For example, in Figure 2a, the formula $\mathsf{A}[(f(1) \vee g(2))\,\mathsf{U}\,h(1,2)]$ holds at state 1, because all paths eventually reach a state, *i.e.*, state 3 or 5, where the proposition $h(1,2)$ holds, and at all previous states, *i.e.*, states 1, 2, and 4, either $f(1)$ or $g(2)$ holds. On the other hand, in Figure 2c, $\mathsf{AX}(g(2))$ does not hold at state 1, because $g(2)$ does not hold at one of the neighbors, state 4. The semantics of CTL is defined over a *model*:

DEFINITION 1. *A* model *is a triple,* $(\mathsf{States}, \rightarrow, \mathsf{Label})$*, where* States *is a finite set of states;* $\rightarrow \subseteq \mathsf{States} \times \mathsf{States}$ *is the successor relation such that* $\forall s.\exists r.s \rightarrow r$*,* i.e.*, every state has at least one successor; and* $\mathsf{Label} : \mathsf{States} \rightarrow \mathcal{P}(\mathsf{Atom})$ *is a labelling function that assigns a set of atomic propositions to each state.*

The set of infinite *paths* starting in state s is denoted $\mathsf{Path}(s)$. For a path $\pi = s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_j \rightarrow \cdots$ the jth element is denoted by $\pi[j]$, *i.e.*, $\pi[j] = s_j$. Note that the set of paths may be infinite. Finally, the next function computes the set of successors of a given state: $\mathsf{next}(s) = \{s' \mid s \rightarrow s'\}$.

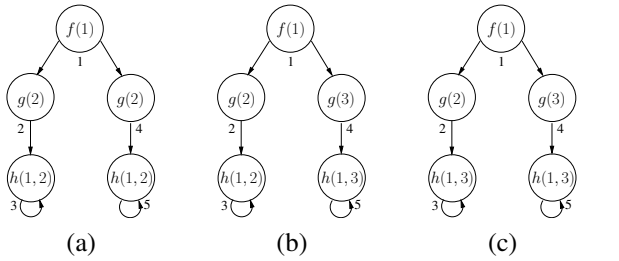

Figure 2. Some simple CTL models illustrating differences between control-flow paths

We now formally define the semantics for CTL as a set of judgements of the form $\mathcal{M}, s \models \phi$, where $\mathcal{M}$ is a model, $s \in \mathsf{States}$ and ϕ is a CTL formula. For convenience, we elide the model in the judgements:

$$\begin{array}{lcl} s \models p & \Leftrightarrow & p \in \mathsf{Label}(s) \\ s \models \phi_1 \wedge \phi_2 & \Leftrightarrow & s \models \phi_1 \wedge s \models \phi_2 \\ s \models \phi_1 \vee \phi_2 & \Leftrightarrow & s \models \phi_1 \vee s \models \phi_2 \\ s \models \neg\phi & \Leftrightarrow & s \not\models \phi \\ s \models \mathsf{AX}\phi & \Leftrightarrow & \forall s' \in \mathsf{next}(s).s' \models \phi \\ s \models \mathsf{EX}\phi & \Leftrightarrow & \exists s' \in \mathsf{next}(s).s' \models \phi \\ s \models \mathsf{A}[\phi_1\,\mathsf{U}\,\phi_2] & \Leftrightarrow & \forall \pi \in \mathsf{Path}(s).\exists i \geq 0.\pi[i] \models \phi_2 \wedge \\ & & \quad \forall 0 \leq j < i.\pi[j] \models \phi_1 \\ s \models \mathsf{E}[\phi_1\,\mathsf{U}\,\phi_2] & \Leftrightarrow & \exists \pi \in \mathsf{Path}(s).\exists i \geq 0.\pi[i] \models \phi_2 \wedge \\ & & \quad \forall 0 \leq j < i.\pi[j] \models \phi_1 \end{array}$$

Figure 3 presents a model checking algorithm, SAT, for CTL. Given a formula and a model, SAT returns the set of states at which the formula is true in the model. The algorithm is based on that presented in [12], but we have reorganized it so that it can be easily extended to the logics CTL-V and CTL-VW that we consider subsequently. In particular, the algorithm accepts predicates parameterized by a list of variables $\overline{x}$, which in the case of CTL will always be an empty list, and it includes a line for formulas $\exists x.\phi$, which are not part of CTL. In the latter case, the instantiation of the model checking algorithm for CTL simply fails.

In this algorithm, the SAT function simply serves to recursively traverse the formula, except in the case of a proposition, where it returns each state where the proposition is satisfied (recall that $\overline{x}$ is empty for CTL). The functions applied to the intermediate SAT results then perform the main calculation of the algorithm. The function conj combines compatible information from two sets of results, and thus implements conjunction. In the case of CTL, this amounts to intersection. The function disj collects all information from two sets of results, and thus implements disjunction. The functions Conj and Disj fold conj and disj over a set of results, respectively, and are used to define some of the other functions. The function neg computes the complement of a set of results, and in the case of CTL amounts to returning the

SAT function:

$$
\begin{array}{lcl}
\mathrm{SAT}(p(\overline{x})) & = & \{\mathsf{inj}(s,\theta) \mid p(\theta(\overline{x})) \in \mathsf{Label}(s) \wedge \\
 & & \qquad \mathsf{dom}(\theta) = \mathsf{fv}(p(\overline{x}))\} \\
\mathrm{SAT}(\phi_1 \wedge \phi_2) & = & \mathsf{conj}(\mathrm{SAT}(\phi_1), \mathrm{SAT}(\phi_2)) \\
\mathrm{SAT}(\phi_1 \vee \phi_2) & = & \mathsf{disj}(\mathrm{SAT}(\phi_1), \mathrm{SAT}(\phi_2)) \\
\mathrm{SAT}(\neg\phi) & = & \mathsf{neg}(\mathrm{SAT}(\phi)) \\
\mathrm{SAT}(\exists x.\phi) & = & \mathsf{exists}(x, \mathrm{SAT}(\phi)) \\
\mathrm{SAT}(\mathsf{AX}\,\phi) & = & \mathsf{pre}_\forall(\mathrm{SAT}(\phi)) \\
\mathrm{SAT}(\mathsf{EX}\,\phi) & = & \mathsf{pre}_\exists(\mathrm{SAT}(\phi)) \\
\mathrm{SAT}(\mathsf{A}[\phi_1\,\mathsf{U}\,\phi_2]) & = & \mathrm{SAT}_{\mathtt{AU}}(\mathrm{SAT}(\phi_1), \mathrm{SAT}(\phi_2)) \\
\mathrm{SAT}(\mathsf{E}[\phi_1\,\mathsf{U}\,\phi_2]) & = & \mathrm{SAT}_{\mathtt{EU}}(\mathrm{SAT}(\phi_1), \mathrm{SAT}(\phi_2))
\end{array}
$$

Operators on SAT results:

$$
\begin{array}{lcl}
\mathsf{conj}(T_1, T_2) & = & \{t_1 \bigcap t_2 \mid t_1 \in T_1 \wedge t_2 \in T_2 \wedge t_1 \bigcap t_2 \text{ is defined}\} \\
\mathsf{Conj} & = & \mathsf{fold\ conj}\ \{\mathsf{inj}(s,\emptyset) \mid s \in \mathsf{States}\} \\
\mathsf{disj}(T_1, T_2) & = & T_1 \cup T_2 \\
\mathsf{Disj} & = & \mathsf{fold\ disj}\ \emptyset \\
\mathsf{neg}(T) & = & \mathsf{Conj}\ \{\mathsf{negone}(t) \mid t \in T\} \\
\mathsf{exists}(x, T) & = & \{\mathsf{existsone}(t) \mid t \in T\} \\
\mathsf{pre}_\forall(T) & = & \bigcup_{s \in \mathsf{States}} (\mathsf{Conj}\ \{\mathsf{shift}(s', T, s) \mid s' \in \mathsf{next}(s)\}) \\
\mathsf{pre}_\exists(T) & = & \bigcup_{s \in \mathsf{States}} (\mathsf{Disj}\ \{\mathsf{shift}(s', T, s) \mid s' \in \mathsf{next}(s)\}) \\
\mathrm{SAT}_{\mathtt{AU}}(T_1, T_2) & = & \textbf{local var}\ W = T_1, Y = T_2, X; \\
 & & \textbf{repeat}\ X = Y; Y = \mathsf{disj}(Y, \mathsf{conj}(W, \mathsf{pre}_\forall(Y))); \\
 & & \textbf{until}\ \mathsf{same}(X, Y); \\
 & & \textbf{return}\ Y; \\
\mathrm{SAT}_{\mathtt{EU}}(T_1, T_2) & = & \textbf{local var}\ W = T_1, Y = T_2, X; \\
 & & \textbf{repeat}\ X = Y; Y = \mathsf{disj}(Y, \mathsf{conj}(W, \mathsf{pre}_\exists(Y))); \\
 & & \textbf{until}\ \mathsf{same}(X, Y); \\
 & & \textbf{return}\ Y;
\end{array}
$$

Element level operators, for CTL:

$$
\begin{array}{ll}
\mathsf{inj}(s, \theta) & = s \\
s_1 \bigcap s_2 & = s_1, \text{ if } s_1 = s_2 \\
\mathsf{negone}(s) & = \{s' \mid s' \in \mathsf{States} - \{s\}\} \\
\mathsf{shift}(s_1, T, s_2) & = \{s_2\}, \text{ if } s_1 \in T, \emptyset \text{ otherwise} \\
\mathsf{same}(T_1, T_2) & = T_1 = T_2 \\
\mathsf{existsone} & \text{is not defined}
\end{array}
$$

Figure 3. A generic model checking algorithm and its instantiation for CTL

difference between the complete set of states and the given set of states. The functions $\mathsf{pre}_\forall$ and $\mathsf{pre}_\exists$ have the effect of checking whether all neighbors of a state are in a given set of results or whether there exists a neighbor of a state that is in a given set of results, respectively. These functions are used in the implementation of AX and EX, respectively, and in the definition of the functions $\mathrm{SAT}_{\mathtt{AU}}$ and $\mathrm{SAT}_{\mathtt{EU}}$ that implement AU and EU.

The remaining functions manipulate the elements of the sets of results, which are states in the case of CTL. The function inj takes as arguments a state and an environment (always empty for CTL) and injects it into the type of results for the given logic, which in the case of CTL implies just dropping the environment. The (partial) function $\bigcap$ determines whether two results are compatible, which simply means that they are equal in the case of CTL. The function negone computes the complement of a single result. The function shift returns all results in a set T that are associated with a state s_1, but replaces the state information in each case by another state s_2; this function makes it convenient to implement $\mathsf{pre}_\forall$ and $\mathsf{pre}_\exists$. Finally, the function same makes it possible to vary the termination condition for the fixpoint iterations in $\mathrm{SAT}_{\mathtt{AU}}$ and $\mathrm{SAT}_{\mathtt{EU}}$.

It is straightforward to show that this algorithm is sound and complete with respect to the semantics [12]:

THEOREM 1 (Soundness and completeness).
$s \models \phi \Leftrightarrow s \in \mathrm{SAT}(\phi)$

3.2 Computational Tree Logic with free variables

While CTL has proven to be very useful in the context of program analysis [22], Lacey *et al.* showed that the extension CTL-FV, supporting not propositions but predicates over free variables, could be used in the context of program *transformation* [17]. In particular CTL-FV has been used to formalize and prove correct a number of classical compiler optimizations [17, 21]. The introduction of variables implies that logic satisfies our first requirement for Coccinelle (Section 2.2).

The syntax of CTL-FV is essentially the same as that of CTL with the addition of predicates over free variables. We refer to these variables as *metavariables*:

$$
\begin{array}{lcl}
\phi & ::= & p(\overline{x}) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \\
 & & \mathsf{AX}\phi \mid \mathsf{EX}\phi \mid \mathsf{A}[\phi\,\mathsf{U}\,\phi] \mid \mathsf{E}[\phi\,\mathsf{U}\,\phi]
\end{array}
$$

$\mathsf{fv}(\phi)$ is the set of free metavariables of a formula ϕ.

The semantics of CTL-FV is almost the same as that of CTL; indeed, they are based on the same models. The new features are that the semantics of predicates now takes free variables into account and all judgements now carry an *environment* θ, mapping metavariables to an arbitrary set of values, Val:

$$\mathsf{Env} = \mathsf{MetaVar} \to \mathsf{Val}$$

When using CTL-FV to describe program transformations, Val is the set of subterms of the program to be transformed. The semantics is defined as follows. Again we elide the model:

$$
\begin{array}{lcl}
s \models_\theta p(\overline{x}) & \Leftrightarrow & p(\theta(\overline{x})) \in \mathsf{Label}(s) \\
s \models_\theta \phi_1 \wedge \phi_2 & \Leftrightarrow & s \models_\theta \phi_1 \wedge s \models_\theta \phi_2 \\
s \models_\theta \phi_1 \vee \phi_2 & \Leftrightarrow & s \models_\theta \phi_1 \vee s \models_\theta \phi_2 \\
s \models_\theta \neg\phi & \Leftrightarrow & s \not\models_\theta \phi \\
s \models_\theta \mathsf{AX}\phi & \Leftrightarrow & \forall s' \in \mathsf{next}(s).s' \models \phi \\
s \models_\theta \mathsf{EX}\phi & \Leftrightarrow & \exists s' \in \mathsf{next}(s).s' \models \phi \\
s \models_\theta \mathsf{A}[\phi_1\,\mathsf{U}\,\phi_2] & \Leftrightarrow & \forall \pi \in \mathsf{Path}(s).\exists i \geq 0.\pi[i] \models_\theta \phi_2 \\
 & & \qquad \forall 0 \leq j < i.\pi[j] \models_\theta \phi_1 \\
s \models_\theta \mathsf{E}[\phi_1\,\mathsf{U}\,\phi_2] & \Leftrightarrow & \exists \pi \in \mathsf{Path}(s).\exists i \geq 0.\pi[i] \models_\theta \phi_2 \\
 & & \qquad \forall 0 \leq j < i.\pi[j] \models_\theta \phi_1
\end{array}
$$

Model checking for CTL-FV has been done using a standard model checker by instantiating the formula with respect to all possible bindings of the metavariables [21].

$f(x) \wedge \mathsf{AX}(g(y) \wedge \mathsf{AX}(h(x,y)))$
$f(x) \wedge \mathsf{AX}(\exists y.(g(y) \wedge \mathsf{AX}(h(x,y))))$

Figure 4. CTL-FV and CTL-V formulas

3.3 Computational Tree Logic with quantified variables

Lacey further extends CTL-FV with the ability to existentially quantify over metavariables [15], producing a logic that we refer to as CTL-V. The syntax of CTL-V is as follows:

$$\begin{array}{rcl} \phi & ::= & p(\overline{x}) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \exists x.\phi \\ & & \mid \mathsf{AX}\phi \mid \mathsf{EX}\phi \mid \mathsf{A}[\phi\,\mathsf{U}\,\phi] \mid \mathsf{E}[\phi\,\mathsf{U}\,\phi] \end{array}$$

The semantics is the same as that of CTL-V, augmented with:

$$s \models_\theta \exists x.\phi \quad \Leftrightarrow \quad \exists v \in \mathsf{Val}.s \models_{\theta[x \mapsto v]} \phi$$

Introducing the ability to quantify over metavariables allows metavariables to have different values within different control-flow paths (our second requirement for Coccinelle) and thus adds flexibility as compared to CTL-FV. For example, consider the formulas in Figure 4, of which the first is in both CTL-FV and CTL-V, and the second is only in CTL-V. Both formulas are satisfied at state 1 in the model in Figure 2(a), as it is possible to uniformly assign x to 1 and y to 2. Only the second formula is satisfied at state 1 in the model in Figure 2(b), as in the left branch (states 2 and 3), y must be 2, while in the right branch (states 4 and 5), y must be 3. Finally, neither formula is satisfied at state 1 in the model in Figure 2(c), as it is not possible to choose a consistent value for y in the left branch (states 2 and 3).

Lacey has presented a model checking algorithm for CTL-V [15]. This algorithm follows a top-down strategy, in which it tries to satisfy the formula at each state of the model, rather than following a bottom-up strategy, which is the source of the efficiency of the standard CTL model checking algorithm presented in Figure 3. No precise performance measurements are provided.

4. A Model Checking Algorithm for CTL-V

In this section, we present a bottom-up model checking algorithm for CTL-V that is based on the CTL model checking algorithm presented in Figure 3. This algorithm is not sufficient as a foundation for Coccinelle, as it does not record information about the bindings of existentially quantified variables. Nevertheless, it permits to introduce a representation of environments that is common to model checking for both CTL-V and CTL-VW, and the algorithm may be of independent interest in context of the work of Lacey and De Moor [16]. While the CTL model checking algorithm identifies the set of states where a formula is satisfied, the CTL-V model checking algorithm must identify the set of pairs of a state and an environment that satisfy the formula. The algorithm is applicable to both CTL-V and CTL-FV, and even to CTL, although the environment it collects is unnecessary in the latter case.

We first present a representation of environments that allows the CTL-V model checking algorithm to efficiently represent information about bindings, then present the CTL-V model checking algorithm, and finally consider its soundness and completeness with respect to the semantics of Section 3.3.

4.1 Environments for CTL-V model checking

A recurring problem in model checking is how to represent the result of negation; the size of the complement of a set of states depends on the size of the model, which can be very large. For CTL-V, where the result of model checking is a set of state-environment pairs, taking the complement of the result includes taking the complement of the environment, where the size of the result depends on the size of Val. In the case of program matching, Val amounts to the set of subterms of the program. In our experience, the set of bindings that derive from matching a predicate against the source code is quite small, and thus taking the complement in this manner would incur a substantial performance overhead. To address this issue, we use *constructive negation* [6].[2]

Rather than negating a binding by creating a disjunction of all of the possible bindings for the variable other than the current one, we add the ability to represent negative bindings directly. Thus, we define an extended form of environment, $\mathsf{Env}^\pm$, in which a variable is mapped to either a positive binding (a particular value in Val) or a set of negative bindings (in $\mathcal{P}(\mathsf{Val})$):

$$\mathsf{Env}^\pm = (\mathsf{MetaVar} \to \mathsf{Val} + \mathcal{P}(\mathsf{Val}))_\perp$$

The added bottom element is used to represent an environment that contains conflicting information. The domain of $\perp$ is undefined. The following notation facilitates reasoning about the positive and negative bindings of an environment:

DEFINITION 2. *Let $\theta \in \mathsf{Env}^\pm$. We define*

1. $\mathsf{dom}^+(\theta) = \{x \in \mathsf{dom}(\theta) \mid \theta(x) \in \mathsf{Val}\}$
2. $\mathsf{dom}^-(\theta) = \{x \in \mathsf{dom}(\theta) \mid \theta(x) \in \mathcal{P}(\mathsf{Val})\}$
3. $\theta^+(x) = \theta(x) \quad \text{iff} \quad x \in \mathsf{dom}^+(\theta)$
4. $\theta^-(x) = \theta(x) \quad \text{iff} \quad x \in \mathsf{dom}^-(\theta)$

$\mathsf{Env}^+ = \{\theta \in \mathsf{Env}^\pm \mid \mathsf{dom}(\theta) \subseteq \mathsf{dom}^+(\theta)\}$ is the set of the environments that have only positive bindings. Env^+ is the same as Env, and thus is the form of environment accepted by the semantics. $\mathsf{Env}^+_\phi = \{\theta \in \mathsf{Env}^\pm \mid \mathsf{dom}(\theta) = \mathsf{dom}^+(\theta) \cap \mathsf{fv}(\phi)\}$ is the subset of Env^+ restricted to the free variables of the formula ϕ.

NOTATION 1. *For convenience, we define the following explicit notation for environments. Let* $\{x_1, \ldots, x_m, y_1, \ldots, y_m\} \subseteq$

[2] Another approach would be to use BDDs, for which negation can be performed in constant time. We have not taken this option, however, because it is not clear how it scales to include witnesses, which we add in CTL-VW.

$\mathsf{MetaVar}$ *be a set of pairwise different meta-variables. Then* $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, y_1 \not\mapsto V_1, \ldots, y_n \not\mapsto V_n]$ *represents the environment* θ *such that* $\theta(x_i) = v_i \in \mathsf{Val}$ *for every* $i \in 1..n$ *and* $\theta(y_j) = V_j \in \mathcal{P}(\mathsf{Val})$ *for every* $j \in 1..m$.

DEFINITION 3 (Conflict). *Two environments* $\theta_1, \theta_2 \in \mathsf{Env}^{\pm}$, *with* $\theta_1 \neq \bot$, $\theta_2 \neq \bot$, *conflict iff* $\exists x \in \mathrm{dom}(\theta_1) \cap \mathrm{dom}(\theta_2)$

$$\theta_1^+(x) \neq \theta_2^+(x) \quad \vee \quad \theta_1^+(x) \in \theta_2^-(x) \quad \vee \quad \theta_2^+(x) \in \theta_1^-(x)$$

Otherwise, θ_1 *and* θ_2 *are said to be compatible.*

Next, we introduce an ordering on environments. This ordering is needed to relate the environments that are returned by the algorithm, which can contain negative bindings, to the ones that satisfy the formula according to the semantics. Intuitively an environment with more specific information is less than one with less specific information, *e.g.*, $[x = 42, y = 12] \sqsubseteq [x = 42] \sqsubseteq [x \neq 87]$.

DEFINITION 4 (Environment ordering). *Let* $\theta_1, \theta_2 \in \mathsf{Env}^{\pm}$ *and define* $\theta_1 \sqsubseteq \theta_2$ iff $\theta_1 = \bot \vee \big(\mathrm{dom}(\theta_1) \supseteq \mathrm{dom}(\theta_2) \wedge \forall x \in \mathrm{dom}(\theta_1).\theta_1^+(x) = \theta_2^+(x) \vee \theta_1^+(x) \notin \theta_2^-(x) \vee \theta_1^-(x) \supseteq \theta_2^-(x)\big)$

According to this ordering, the environment with the empty set as its domain is the greatest element $\top$ and $\bot$ is the least element.

To compute the environment that satisfies a conjunction in a given state, the algorithm joins the environments that satisfy each of the conjunct formulas in the same state. The join of two environments is defined as follows:

DEFINITION 5 (Environment join). *Let* $\theta_1, \theta_2 \in \mathsf{Env}^{\pm}$. *We define* $\theta_1 \sqcap \theta_2$ *to be* $\bot$ *if* $\theta_1 = \bot$ *or* $\theta_2 = \bot$, *i.e.,* $\theta_1 \sqcap \bot = \bot \sqcap \theta_2 = \bot$. *For* $\theta_1 \neq \bot$, $\theta_2 \neq \bot$, *if* θ_1 *and* θ_2 *are compatible,* $\theta_1 \sqcap \theta_2$ *is defined as follows:*

$$(\theta_1 \sqcap \theta_2)(x) = \begin{cases} v & \text{if } \theta_1^+(x) = v \vee \theta_2^+(x) = v \\ V_1 & \text{if } \theta_1^-(x) = V_1 \wedge x \notin \mathrm{dom}(\theta_2) \\ V_2 & \text{if } \theta_2^-(x) = V_2 \wedge x \notin \mathrm{dom}(\theta_1) \\ V_1 \cup V_2 & \text{if } \theta_1^-(x) = V_1 \wedge \theta_2^-(x) = V_2 \end{cases}$$

If θ_1 *and* θ_2 *conflict, then* $\theta_1 \sqcap \theta_2 = \bot$.

It is straightforward to show that the join operator computes the greatest lower bound of two environments according to the environment ordering.

Finally, we conclude with the definition of the negation of an environment. This operation can create negative bindings.

DEFINITION 6 (Negation of an environment). *For* $\theta \in \mathsf{Env}^{\pm}$, *we define*

$$\neg\theta = \begin{cases} \{\top\} & \text{if } \theta = \bot \\ \{[x \not\mapsto \{v\}] \mid v = \theta^+(x)\} \cup \{[x \mapsto v] \mid v \in \theta^-(x)\} & \\ \quad \text{otherwise} & \end{cases}$$

Notice that $\neg\theta \in \mathcal{P}(\mathsf{Env}^{\pm})$.

4.2 A bottom-up CTL-V model checking algorithm

The algorithm SAT for CTL-V takes as arguments a model and a formula and returns result in the form of a set of pairs $(s, \theta) \in \mathsf{State} \times \mathsf{Env}^{\pm}$. As compared to the definition of SAT in Figure 3, it is only necessary to redefine the element-level functions inj, $\bigsqcap$, negone, $\mathsf{existsone}$, shift, and same to take environments into account. The new definitions are as follows:

$$\begin{array}{lcl} \mathsf{inj}(s, \theta) & = & (s, \theta) \\ (s_1, \theta 1) \bigsqcap (s_2, \theta 1) & = & (s_1, \theta_1 \sqcap \theta_2), \text{ if } s_1 = s_2 \wedge \theta_1 \sqcap \theta_2 \neq \bot \\ \mathsf{negone}(s, \theta) & = & \{(s', \emptyset) \mid s' \in \mathsf{States} - \{s\}\} \cup \\ & & \{(s, \theta') \mid \theta' \in \neg\theta\} \\ \mathsf{existsone}(x, (s, \theta)) & = & (s, \theta - [x \mapsto v]), \text{ if } \theta^+(x) = v \\ & & (s, \theta - [x \not\mapsto V]), \text{ if } \theta^-(x) = V \\ & & (s, \theta), \text{ otherwise} \\ \mathsf{shift}(s_1, T, s_2) & = & \{(s_2, \theta) \mid (s_1, \theta) \in T\} \\ \mathsf{same}(T_1, T_2) & = & T_1 = T_2 \end{array}$$

The function inj, used to inject the result of matching a predicate into the codomain of SAT, now keeps both the state and the environment argument, as the environment argument is now non-trivial. The function $\bigsqcap$, used in computing a conjunction of results, extends the CTL definition by taking the join of the associated environments. The function negone, used in computing the negation of a result, similarly extends the definition for CTL by including not only each element of the complement of the given state, paired with the empty environment, but also pairs for the current state combined with each element of the negation of the current environment. These pairs represent the least specific ones that are incompatible with the given state and environment pair. The function shift selects the pairs in T that have the same state as its first argument and replaces the state in each pair with the state in its third argument, as is needed to implement the operators AX and EX. Finally, the function same simply checks whether the results it is given represent equivalent sets, as in the CTL definition.

4.3 Examples

As examples of the CTL-V model checking process, we consider again the models of Figure 2 and the formulas of Figure 4. In checking the formula $f(x) \wedge \mathsf{AX}(g(y) \wedge \mathsf{AX}(h(x, y)))$ with respect the model in Figure 2b, the result for the subformula $g(y) \wedge \mathsf{AX}(h(x, y))$ is

$$\{(2, [x \mapsto 1, y \mapsto 2]), (4, [x \mapsto 1, y \mapsto 3])\}$$

Model checking of the enclosing AX, however fails, because although both of the neighbors of state 1 are in the above result, they are associated with conflicting environments. On the other hand, in checking the formula $f(x) \wedge \mathsf{AX}(\exists y.(g(y) \wedge \mathsf{AX}(h(x, y))))$, with respect to the same model, the same result is obtained for $g(y) \wedge \mathsf{AX}(h(x, y))$, but for $\exists y.(g(y) \wedge$

$\mathsf{AX}(h(x,y)))$, the result is

$$\{(2,[x \mapsto 1]),(4,[x \mapsto 1])\}$$

in which the conflicting bindings of y have been dropped. This time, the result of the enclosing AX is $\{(1,[x \mapsto 1])\}$, which in turn is the result of checking the entire formula.

4.4 Soundness and completeness

The model checking algorithm for CTL-V can return environments that contain negative bindings, while the semantics for CTL-V only accepts environments with positive bindings. Thus, the relationship between them is not as direct as in the case of CTL (Theorem 1). Instead, the soundness and completeness theorem for the CTL-V model checking algorithm relates an environment produced by the algorithm, which may contain negative bindings, to the set of more specific environments that contain only positive bindings and are accepted by the semantics:

THEOREM 2 (Soundness and completeness). *Let ϕ be a formula, and $T \in \mathcal{P}(\mathsf{State} \times \mathsf{Env}^{\pm})$. Let us define $\eta_\phi(T) = \{(s,\theta) \in \mathsf{State} \times \mathsf{Env}^{+}_{\phi} \mid \exists (s',\theta') \in T.s = s' \wedge \theta \sqsubseteq \theta'\}$. Then,*

$$\forall s \forall \theta \quad (\theta \in \mathsf{Env}^{+}_{\phi} \wedge s \models_\theta \phi) \Leftrightarrow (s,\theta) \in \eta_\phi(\mathrm{SAT}(\phi))$$

The proof of this theorem has been validated with Coq [3]. [3]

5. CTL-VW

CTL-V allows variables to have different values in different control-flow paths, but discards their bindings, making it impossible to refer to such variables in a subsequent transformation phase. The goal of CTL-VW is to collect the bindings of such variables, as *witnesses*, in a way that does not affect the rest of the matching process. This satisfies the third of our requirements for Coccinelle (Section 2.2). We first define witnesses, then give the semantics and model checking algorithm of CTL-VW, and finally sketch the proof of the soundness and completeness of the model checking algorithm.

5.1 Witnesses

A witness is essentially a record of a state, a binding, and the set of witnesses for other bindings that contributed to establishing the binding in the given state. A witness thus has a tree structure, which corresponds to the structure of the nested existential quantifiers in the formula.

DEFINITION 7 (Witnesses). *The set* Wit *of witnesses is defined as:*

$$\mathsf{Wit} = \mathsf{States} \times \mathsf{MetaVar} \times (\mathsf{Val} + \mathcal{P}(\mathsf{Val})) \times \mathsf{WitForest}$$

A witness forest $\Omega \in \mathsf{WitForest}$ is a multiset of witnesses, i.e., a pair (W,f), where $W \subseteq \mathsf{Wit}$ is a set of witnesses, and $f : W \to \mathbb{N}^*$ is a function that associates each witness with its multiplicity (a nonnegative integer). We define $\mathsf{WitForest}^+$ as the set of the witness forests in which all bindings are positive.

The operator $\uplus$ produces the join of two multisets. For ease of reading, we sometimes use the set-like notation $\{\ldots\}$ to enumerate the elements of a multiset. Then, $\{a,a,b\}$ represents the multiset whose underlying set is $\{a,b\}$, and the multiplicities of a and b are 2 and 1, respectively.

We use multisets rather than sets for technical reasons. However, in order to specify a termination criterion in the model checking algorithm, we need to reason about the underlying sets of witness forests. We define the binary operator $\simeq$ as follows:

$$\Omega_1 \simeq \Omega_2 \quad \text{iff} \quad \mathsf{wit\_red}(\Omega_1) = \mathsf{wit\_red}(\Omega_2)$$

$\mathsf{wit\_red}(\Omega)$ translates Ω into the corresponding underlying set.

As for environments, we define an ordering on witnesses, and on witness forests. These orderings are useful to relate the results of the algorithm, which can contain negative bindings, to witness forests that satisfy the formula according to the semantics.

DEFINITION 8 (Witness ordering). *Let $w = \langle s,x,a,\Omega\rangle$ and $w' = \langle s',x',a',\Omega'\rangle$ be two witnesses. We define $w \sqsubseteq w'$ iff $s = s' \wedge x = x' \wedge [x \mapsto a] \sqsubseteq [x' \mapsto a'] \wedge \Omega \sqsubseteq \Omega'$.*

The ordering for witness forests is defined as follows: $(W,f) \sqsubseteq (W',f')$ iff there is a bijection h such that

1. $h : \{(w,i) \mid w \in W \wedge i \in 1..f(w)\} \to \{(w',j) \mid w' \in W' \wedge j \in 1..f'(w')\}$
2. $\forall w \in W. \forall i \in 1..f(w). w \sqsubseteq w'$, *where there is some j such that* $(w',j) = h(w,i)$

5.2 A semantics for CTL-VW

The new feature of the semantics of CTL-VW is to define what it means to be a witness forest for a formula ϕ, with respect to some state and environment. If $\phi = \exists x.\psi$, then a witness forest for ϕ should record a binding of x that satisfies ψ, as well as a witness forest that records information about the bindings associated with any existential quantifiers in ψ. If $\phi = \phi_1 \vee \phi_2$, then a witness forest for ϕ has to be a witness forest for either ϕ_1 or ϕ_2, because the information needed to satisfy ϕ_1 or ϕ_2 also satisfies $\phi_1 \vee \phi_2$.

A witness forest for $\phi = \phi_1 \wedge \phi_2$ has to contain the information that makes ϕ_1 true and the information that makes ϕ_2 true. Thus, the semantics joins a witness forest for ϕ_1 and a witness forest for ϕ_2 to obtain a witness forest for $\phi_1 \wedge \phi_2$. Similarly, for $\mathsf{AX}\psi$, the semantics collects, for each successor of the current state, a witness forest that makes ψ true in this successor. In both cases, we collect the bindings of the metavariables that make the subformulas true at the relevant states. On the other hand, since EX is related to a

[3] URL: `http://www.emn.fr/x-info/coccinelle/ctlv.tar.gz`

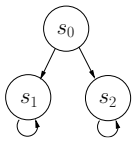

Figure 5. Illustration of finite unfolding

disjunction, a witness forest for $\mathsf{EX}\psi$ is a witness forest for ψ in one of the successors of the current state.

If $\phi = \mathsf{A}[\phi_1 \,\mathsf{U}\, \phi_2]$, then we can define a witness forest for ϕ by analogy with conjunction and AX. Then, the usual semantics of AU implies that a witness forest of ϕ collects as many witnesses as there are paths starting from the current state. This leads to an infinite number of witness forests, carrying redundant information. In order to collect only witness forests that are pertinent, we reason about a finite unfolding of the model when defining the semantics of AU, rather that reasoning about all the paths. Then, a witness forest of $\mathsf{A}[\phi_1 \,\mathsf{U}\, \phi_2]$ collects a witness forest of ϕ_1 for every state in the finite unfolding that is not a leaf, and a witness forest of ϕ_2 for every leaf. A witness forest for $\phi = \mathsf{E}[\phi_1 \,\mathsf{U}\, \phi_2]$ collects, for some path leading to a leaf, a witness forest for ϕ_1 in every non-leaf state in this path, and a witness forest for ϕ_2 in the leaf.

DEFINITION 9 (Finite paths). *Given a model and a state s, we define $\mathsf{FPath}(s)$ as the set of the finite paths starting from s. Given a finite path $\pi \in \mathsf{FPath}(s)$, $|\pi[$ is the length of π, i.e., $|\pi| + 1$ is the number of states in π, $\mathsf{prefix}(\pi)$ represents the set of the (finite) proper prefixes of π, and $\pi_{\leqslant n}$, for $n \leqslant |\pi|$, represents the prefix of π containing the first $n+1$ states (from $\pi[0]$ to $\pi[n]$).*

DEFINITION 10 (Finite unfolding). *Given a state s, a finite unfolding from s is a set $\Sigma \subseteq \mathsf{FPath}(s)$ where every path $\pi \in \Sigma$ satisfies the following constraints:*

- $\forall \pi' \in \mathsf{prefix}(\pi).\ \pi' \notin \Sigma$ *(no redundancy)*
- $\forall i \in 0..|\pi| - 1.\ \forall s' \in \mathsf{next}(\pi[i]).\ \exists \pi' \in \Sigma.\ \pi'_{\leqslant i} = \pi_{\leqslant i} \wedge s' = \pi'[i+1]$ *(full branching)*

The set of the finite unfoldings starting from s is denoted $\Pi(s)$. The set of the states that appear in some path of Σ is denoted $\overline{\Sigma}$.

Consider the model shown in Figure 5. Then $\{s_0\}$ and $\{s_0 \to s_1, s_0 \to s_2 \to s_2\}$ are two valid finite unfoldings starting from s_0. On the other hand, $\{s_0 \to s_1, s_0 \to s_2, s_0 \to s_2 \to s_2\}$ violates the first constraint of Definition 10, and $\{s_0 \to s_1\}$ does not explore the right branch, and thus does not satisfy the second constraint.

DEFINITION 11 (Semantics). *Given a model $(\mathsf{States}, \to, \mathsf{Label})$, a state $s \in \mathsf{States}$, a formula ϕ, an environment $\theta \in \mathsf{Env}^+$, and a witness forest $\Omega \in \mathsf{WitForest}^+$, the semantics is defined as the following relation:*

$$
\begin{array}{lcl}
s \models_{\theta,\emptyset} p(\overline{x}) & \Leftrightarrow & p(\theta(\overline{x})) \in \mathsf{Label}(s) \\
s \models_{\theta,\Omega_1 \uplus \Omega_2} \phi_1 \wedge \phi_2 & \Leftrightarrow & s \models_{\theta,\Omega_1} \phi_1 \wedge s \models_{\theta,\Omega_2} \phi_2 \\
s \models_{\theta,\Omega} \phi_1 \vee \phi_2 & \Leftrightarrow & s \models_{\theta,\Omega} \phi_1 \vee s \models_{\theta,\Omega} \phi_2 \\
s \models_{\theta,\emptyset} \neg\phi & \Leftrightarrow & \forall \Omega.\ s \not\models_{\theta,\Omega} \phi \\
s \models_{\theta,\{\langle s,x,v,\Omega\rangle\}} \exists x.\phi & \Leftrightarrow & s \models_{\theta[x\mapsto v],\Omega} \phi \\
s \models_{\theta,\Omega} \mathsf{AX}\phi & \Leftrightarrow & \exists (\Omega_{s'})_{s' \in \mathsf{next}(s)} \cdot \\
\multicolumn{3}{r}{\biguplus_{s' \in \mathsf{next}(s)} \Omega_{s'} = \Omega \ \wedge\ \forall s' \in \mathsf{next}(s).\ s' \models_{\theta,\Omega_{s'}} \phi} \\
s \models_{\theta,\Omega} \mathsf{EX}\phi & \Leftrightarrow & \exists s' \in \mathsf{next}(s).\ s' \models_{\theta,\Omega} \phi \\
s \models_{\theta,\Omega} \mathsf{A}[\phi_1 \,\mathsf{U}\, \phi_2] & \Leftrightarrow & \exists \Sigma \in \Pi(s).\ \exists (\Omega_{s'})_{s' \in \overline{\Sigma}} \cdot \\
\multicolumn{3}{r}{\forall \pi \in \Sigma.\ \pi[|\pi|] \models_{\theta,\Omega_{\pi[|\pi|]}} \phi_2 \wedge \forall 0 \leqslant j < |\pi|.\ \pi[j] \models_{\theta,\Omega_{\pi[j]}} \phi_1} \\
\multicolumn{3}{r}{\wedge \biguplus_{s' \in \overline{\Sigma}} \Omega_{s'} = \Omega} \\
s \models_{\theta,\Omega} \mathsf{E}[\phi_1 \,\mathsf{U}\, \phi_2] & \Leftrightarrow & \exists \Sigma \in \Pi(s).\ \exists \pi \in \Sigma. \exists (\Omega_i)_{i \in 0..|\pi|} \cdot \\
\multicolumn{3}{r}{\pi[|\pi|] \models_{\theta,\Omega_{|\pi|}} \phi_2 \wedge \forall 0 \leqslant j < |\pi|.\ \pi[j] \models_{\theta,\Omega_j} \phi_1} \\
\multicolumn{3}{r}{\wedge \biguplus_{j \leqslant |\pi|} \Omega_j = \Omega}
\end{array}
$$

Note that the semantics does not allow any witness to be satisfied under a negation. This facilitates the proof of the soundness and completeness of the model checking algorithm. Indeed, a negation turns the nature of an existential quantification to a universal quantification, for which witnesses would not make any sense. However, one could justify the interest of a witness for an existential quantifier that appears under an even number of negations. We do not consider a special treatment for this case. Because of this constraint on negation, we must include both AX and EX in the logic explicitly, because a definition of *e.g.*, EX in terms of AX and negation, would preclude witnesses under EX.

5.3 A model checking algorithm for CTL-VW

The algorithm SAT for CTL-VW takes as arguments a model and a formula and returns a set of triples of the form $(s, \theta, \Omega) \in \mathsf{States} \times \mathsf{Env}^{\pm} \times \mathsf{WitForest}$. The new definitions of the element-level functions of Figure 3 are as follows:

$$
\begin{array}{l}
\mathsf{inj}(s,\theta) = (s,\theta,\emptyset) \\
(s_1,\theta_1,\Omega_1) \bigsqcap (s_2,\theta_2,\Omega_2) = \\
\quad (s_1, \theta_1 \sqcap \theta_2, \Omega_1 \uplus \Omega_2),\ \text{if } s_1 = s_2 \wedge \theta_1 \sqcap \theta_2 \neq \bot \\
\mathsf{negone}(s,\theta,\Omega) = \\
\quad \{(s',\emptyset,\emptyset) \mid s' \in \mathsf{States} - \{s\}\} \cup \{(s,\theta',\emptyset) \mid \theta' \in \neg\theta\} \\
\mathsf{existsone}(x,(s,\theta,\Omega)) = \\
\quad (s, \theta - [x \mapsto v], \{\langle s,x,v,\Omega\rangle\}),\ \text{if } \theta^+(x) = v \\
\quad (s, \theta - [x \not\mapsto V], \{\langle s,x,V,\Omega\rangle\}),\ \text{if } \theta^-(x) = V \\
\quad (s,\theta,\{\langle s,x,\emptyset,\Omega\rangle\}),\ \text{otherwise} \\
\mathsf{shift}(s_1,T,s_2) = \{(s_2,\theta,\Omega) \mid (s_1,\theta,\Omega) \in T\} \\
\mathsf{same}(T_1,T_2) = \mathsf{triple\_red}(T_1) = \mathsf{triple\_red}(T_2) \\
\text{where } \mathsf{triple\_red}(T) = \{(s,\theta,\mathsf{wit\_red}(\Omega)) \mid (s,\theta,\Omega) \in T\}
\end{array}
$$

These definitions augment the corresponding definitions for CTL-V to maintain the witnesses. As there are no existential quantifiers under a predicate, the function inj just uses an empty witness forest. The function $\bigsqcap$ takes the join of the provided witness forests, if the provided states and environments are compatible. The function existsone considers the same

cases as for CTL-V, but rather than discarding the binding of the quantified variable converts it to a witness, encapsulating the current witness forest. The function shift is analogous to the CTL-V definition. Finally, the function same, used to identify a fixed point, first converts all witness forests in its arguments to their underlying sets, using wit_red, and then compares the obtained results.

It remains to consider the definition of negone. This function negates the state component, and then the environment component, as in the CTL-V definition, but completely discards the witness component. This strategy corresponds to the semantics of negation presented in Section 5.2, where the subformula should not be satisfiable at the given state and environment for any witness forest.

5.4 Examples

As in Section 4.3, we consider the models of Figure 2 and the formulas of Figure 4. In checking the formula $f(x) \wedge \mathsf{AX}(g(y) \wedge \mathsf{AX}(h(x,y)))$ with respect the model in Figure 2b, the result for the subformula $g(y) \wedge \mathsf{AX}(h(x,y))$ is

$$\{(2, [x \mapsto 1, y \mapsto 2], \emptyset), (4, [x \mapsto 1, y \mapsto 3], \emptyset)\}$$

Model checking for the enclosing AX fails as before, because the environments are incompatible. Similarly, in checking the formula $f(x) \wedge \mathsf{AX}(\exists y.(g(y) \wedge \mathsf{AX}(h(x,y))))$, with respect to the same model, the same result is obtained for $g(y) \wedge \mathsf{AX}(h(x,y))$, but now for $\exists y.(g(y) \wedge \mathsf{AX}(h(x,y)))$, the result is

$$\{(2, [x \mapsto 1], \{\langle 2, y, 2, \emptyset\rangle\}), (4, [x \mapsto 1], \{\langle 4, y, 3, \emptyset\rangle\})\}$$

in which the information about the various bindings of y is still available. The result of the enclosing AX is then

$$\{(1, [x \mapsto 1], \{\langle 2, y, 2, \emptyset\rangle, \langle 4, y, 3, \emptyset\rangle\})\}$$

which is in turn the result for the entire formula. This result includes the possible bindings of y and the states at which those bindings are applicable.

5.5 Soundness and completeness

As for CTL-V, the CTL-VW model checking algorithm may return an environment with negative bindings. We thus extend the η function of Section 4.4 to take witnesses into account.

Let ϕ be a formula, and $T \in \mathcal{P}(\mathsf{States} \times \mathsf{Env}^{\pm} \times \mathsf{WitForest})$. Let us define $\eta_\phi(T) = \{(s, \theta, \Omega) \in \mathsf{States} \times \mathsf{Env}^{+}_{\phi} \times \mathsf{WitForest}^{+} \mid \exists (s', \theta', \Omega') \in T.\ s = s' \wedge \theta \sqsubseteq \theta' \wedge \Omega \sqsubseteq \Omega'\}$.

THEOREM 3 (Soundness). $\forall s, \theta, \Omega.$

$$\theta(s, \theta, \Omega) \in \eta_\phi(\mathrm{SAT}(\phi)) \implies s \models_{\theta,\Omega} \phi$$

Proof sketch The proof is by structural induction on ϕ. We present the case $\phi = \mathsf{A}[\phi_1 \,\mathsf{U}\, \phi_2]$. We first define a bounded semantics for AU, and a bounded SAT algorithm for AU. $s \models_{\theta,\Omega} \mathsf{A}[\phi_1 \,\mathsf{U}\, \phi_2]_{\leqslant N}$ iff there is a finite unfolding Σ whose maximum length is less than or equal to N, and such that the semantic definition of AU is satisfied with Σ. $\mathsf{SAT}_{\mathtt{AU}}(\phi_1, \phi_2)_{\leqslant N}$ corresponds to N iterations of the algorithm described in $\mathsf{SAT}_{\mathtt{AU}}(\phi_1, \phi_2)$. The following lemma relates the bounded semantics to the bounded algorithm.

LEMMA 1. *Let* $\phi = \mathsf{A}[\phi_1 \,\mathsf{U}\, \phi_2]$. *Then,* $\forall s, \theta, \Omega, N$

$$(s, \theta, \Omega) \in \eta_\phi(\mathsf{SAT}_{\mathtt{AU}}(\phi_1, \phi_2)_{\leqslant N}) \implies s \models_{\theta,\Omega} \mathsf{A}[\phi_1 \,\mathsf{U}\, \phi_2]_{\leqslant N}$$

We omit the proof of this lemma, also done by induction.

Let $(s, \theta, \Omega) \in \eta_\phi(\mathsf{SAT}_{\mathtt{AU}}(\phi_1, \phi_2))$. Then, $\exists N.\ (s, \theta, \Omega) \in \eta_\phi(\mathsf{SAT}_{\mathtt{AU}}(\phi_1, \phi_2)_{\leqslant N})$. By Lemma 1, $s \models_{\theta,\Omega} \mathsf{A}[\phi_1 \,\mathsf{U}\, \phi_2]_{\leqslant N}$, so $s \models_{\theta,\Omega} \mathsf{A}[\phi_1 \,\mathsf{U}\, \phi_2]$. □

The completeness theorem cannot be stated as directly as the soundness theorem. Indeed, if ϕ contains AU (resp. EU), a witness forest accepted by the semantics may contain more witnesses than are generated by $\mathsf{SAT}_{\mathtt{AU}}$ (resp.$\mathsf{SAT}_{\mathtt{EU}}$) in reaching a fixed point. In this case, the witness forest is not returned by the algorithm in its exact form, but the algorithm produces a shorter and equivalent (in the sense of $\simeq$) witness forest.

THEOREM 4 (Completeness). $\forall s, \theta, \Omega.$

$$\theta \in \mathsf{Env}^{+}_{\phi} \wedge s \models_{\theta,\Omega} \phi \implies \exists \Omega' \simeq \Omega.\ (s, \theta, \Omega') \in \eta_\phi(\mathrm{SAT}(\phi))$$

Proof sketch The proof is done by induction. In the AU case, we use a lemma analogous to Lemma 1. □

To conclude, we consider the relationship between CTL-VW and CTL-V. CTL-VW is a conservative extension of CTL-V: the collection of witnesses does not have any impact on the satisfiability of a formula.

PROPERTY 1

$$\forall s, \theta.\, s \underset{\textit{CTL-V}}{\models_{\theta}} \phi \iff \exists \Omega \in \mathsf{WitForest}^{+}.\ s \underset{\textit{CTL-VW}}{\models_{\theta,\Omega}} \phi$$

6. Applying CTL-VW in the context of Coccinelle

In this section, we define a core SmPL language, that is sufficient to treat the example presented in Section 2.1 (Figure 1), present its translation into CTL-VW, describe the matching and transformation process, and then present a few benchmarks. Many more examples of the use of SmPL and the associated performance are available in our previous work [18, 20]. The section concludes with a brief example illustrating the benefit of being able to mix existential and universal path quantification, as provided by temporal logics based on CTL.

For the purposes of this presentation, we have slightly simplified the semantics of SmPL, in that our encoding in CTL-VW allows dots ("...") to match any path, while

in SmPL dots represent the shortest path between terms matching the preceding and following patterns. This shortest path constraint can be encoded straightforwardly in CTL-VW. Furthermore, it is possible to drop the shortest path constraint in full SmPL by annotating the dots with `when any`.

6.1 Syntax of a simplified SmPL

The syntax of the transformation specification part of a semantic patch is quite complex, because - and + annotations can be freely mixed. In a trasformation specification, the combination of the unannotated code and the - code represents the pattern to match against, and the combination of the unannotated code and the + code represents the code to generate. Each must have the structure of valid C code. Expressing these constraints is, however, unrelated to the use of CTL-VW. Instead, we define the grammar of patterns to match against, and assume that the elements of these patterns are implicitly annotated with the transformation to perform. For example, in the type_ref semantic patch of Figure 1, the pattern `return -`C`;` would be annotated to indicate that `of_node_put(`*n*`);` should be inserted before it.

In our simplified SmPL, the transformation specification part of a semantic patch has the form of a sequence S, as defined by the following grammar:

$$
\begin{array}{rcrcl}
S & \in & \text{Sequences} & ::= & E\,S \mid \varepsilon \\
E & \in & \text{Elements} & ::= & T \mid D \mid \texttt{(}S \mid S\texttt{)} \\
T & \in & \text{Terms} & ::= & \textit{Atomic} \mid \texttt{if (}\textit{exp}\texttt{)}\ T \\
D & \in & \text{Dots} & ::= & \texttt{...} \mid D\ \texttt{when != }S
\end{array}
$$

A Sequence should not contain consecutive dots (D), but we do not complicate the grammar with this constraint. *Atomic* is an arbitrary atomic term, such as an assignment or function call, and *exp* is an arbitrary expression. *Atomic* and *exp* may contain metavariables. We consider conditionals with only one branch, because that is all that is required for our example. The treatment of a conditional with two branches is similar.

6.2 Quantification of metavariables

The first step in the translation of SmPL to CTL-VW is to introduce existential quantifiers to delimit the scope of each metavariable. For this, we extend the syntax of Sequences, Elements, and Terms to include existential quantifiers, as shown below:

$$
\begin{array}{rcrcl}
S & \in & \text{Sequences} & ::= & E\,S \mid \varepsilon \mid \exists x.S \\
E & \in & \text{Elements} & ::= & T \mid D \mid \texttt{(}S \mid S\texttt{)} \mid \exists x.E \\
T & \in & \text{Terms} & ::= & \textit{Atomic} \mid \texttt{if (}\textit{exp}\texttt{)}\ T \mid \exists x.T \\
D & \in & \text{Dots} & ::= & \texttt{...} \mid D\ \texttt{when != }S
\end{array}
$$

The scope of a metavariable is the smallest Sequence, Element, or Term that contains all references to it, including references in + code. Inserting existential quantifiers according to this strategy is straightforward. For the semantic patch in Figure 1, the scope of the metavariable n extends around the entire semantic patch, but the scope of the other metavariables is only the immediate containing Term. In particular, the scope of the metavariable C in the modified term `return -`C`;` is local to the pattern itself, *i.e.*, $\exists C.$ `return -`C`;`, allowing it to match a return of any negative constant within the different control-flow paths. This property is essential, as the matched code may need to return an error code for many reasons, returning a different value in each case (see Figure 8).

$$
\begin{array}{rcl}
\mathcal{C}_s[\![E\ S]\!]\ a & = & \mathcal{C}_e[\![E]\!]\ (\mathcal{C}_s[\![S]\!]\ a) \\
\mathcal{C}_s[\![\varepsilon]\!]\ a & = & a \\
\mathcal{C}_s[\![\exists x.S]\!]\ a & = & \exists x.\mathcal{C}_s[\![S]\!]\ a \\
\\
\mathcal{C}_e[\![T]\!]\ a & = & \mathcal{C}_t[\![T]\!]a \\
\mathcal{C}_e[\![D]\!]\ a & = & \mathsf{A}[\mathcal{G}_d[\![D]\!]\ a\ \mathsf{U}\ a] \\
\mathcal{C}_e[\![(S_1 \mid S_2)]\!]\ a & = & \mathcal{C}_s[\![S_1]\!]\ a \vee (\neg(\mathcal{C}_s[\![S_1]\!]\ a) \wedge \mathcal{C}_s[\![S_2]\!]\ a) \\
\mathcal{C}_e[\![\exists x.E]\!]\ a & = & \exists x.\mathcal{C}_e[\![E]\!]\ a \\
\\
\mathcal{C}_t[\![\textit{Atomic}]\!]a & = & (\textit{Atomic} \wedge \exists \_v.\_v = \text{“}\textit{Atomic}\text{”}) \wedge \mathsf{AX}\ a \\
\mathcal{C}_t[\![\texttt{if (}\textit{exp}\texttt{)}\ T]\!]a & = & (\texttt{if (}\textit{exp}\texttt{)} \wedge \exists \_v.\_v = \text{“}\texttt{if (}\textit{exp}\texttt{)}\text{”}) \wedge \\
 & & \mathsf{AX}((\mathsf{trueBranch} \wedge \mathsf{AX}(\mathcal{G}_t[\![T]\!])) \vee \\
 & & \quad \mathsf{fallThrough} \vee (\mathsf{after} \wedge \mathsf{AXAX}\ a)) \wedge \mathsf{EX}(\mathsf{after}) \\
\mathcal{C}_t[\![\exists x.T]\!]a & = & \exists x.\mathcal{C}_t[\![T]\!]a \\
\\
\mathcal{G}_s[\![E]\!]\ a & = & \mathcal{G}_e[\![E]\!]\ a \\
\mathcal{G}_s[\![E\ S]\!]\ a & = & \mathcal{G}_e[\![E]\!]\ (\mathcal{G}_s[\![S]\!]\ a) \\
\mathcal{G}_s[\![\varepsilon]\!]\ a & = & \mathsf{true} \\
\mathcal{G}_s[\![\exists x.S]\!]\ a & = & \exists x.\mathcal{G}_s[\![S]\!]\ a \\
\\
\mathcal{G}_e[\![T]\!]\ a & = & \mathcal{G}_t[\![T]\!] \\
\mathcal{G}_e[\![D]\!]\ a & = & \mathsf{A}[\mathcal{G}_d[\![D]\!]\ a\ \mathsf{U}\ a] \\
\mathcal{G}_e[\![(S_1 \mid S_2)]\!]\ a & = & \mathcal{G}_s[\![S_1]\!]\ a \vee (\neg(\mathcal{G}_s[\![S_1]\!]\ a) \wedge \mathcal{G}_s[\![S_2]\!]\ a) \\
\mathcal{G}_e[\![\exists x.E]\!]\ a & = & \exists x.\mathcal{G}_e[\![E]\!]\ a \\
\\
\mathcal{G}_t[\![\textit{Atomic}]\!] & = & \textit{Atomic} \wedge \exists \_v.\_v = \text{“}\textit{Atomic}\text{”} \\
\mathcal{G}_t[\![\texttt{if (}\textit{exp}\texttt{)}\ T]\!] & = & (\texttt{if (}\textit{exp}\texttt{)} \wedge \exists \_v.\_v = \text{“}\texttt{if (}\textit{exp}\texttt{)}\text{”}) \wedge \\
 & & \mathsf{AX}((\mathsf{trueBranch} \wedge \mathsf{AX}(\mathcal{G}_t[\![T]\!])) \vee \\
 & & \quad \mathsf{fallThrough} \vee \mathsf{after}) \\
\mathcal{G}_t[\![\exists x.T]\!] & = & \exists x.\mathcal{G}_t[\![T]\!] \\
\\
\mathcal{G}_d[\![\texttt{...}]\!]\ a & = & \mathsf{true} \\
\mathcal{G}_d[\![D\ \texttt{when != }S]\!]\ a & = & \mathcal{G}_d[\![D]\!]\ a \wedge \neg(\mathcal{G}_s[\![S]\!]\ a)
\end{array}
$$

Figure 6. A simplified translation of semantic patches to CTL-VW

6.3 Translation to CTL-VW

Figure 6 defines the translation of a pattern into CTL-VW. The entry point of the translation is the function $\mathcal{C}_s$, which takes as arguments a Sequence and a formula describing how to match the remainder of the semantic patch. Initially, a transformation specification S is translated as $\mathcal{C}_s[\![S]\!]\,\mathsf{true}$.[4]

The translation contains two sets of rules: 1) the $\mathcal{C}$ rules $\mathcal{C}_s$, $\mathcal{C}_e$, and $\mathcal{C}_t$, for translating Sequences, Elements, and Terms, respectively, at the top level, and 2) the $\mathcal{G}$ rules $\mathcal{G}_s$, $\mathcal{G}_e$, $\mathcal{G}_t$, for Sequences, Elements, Terms, and Dots, respectively, when these occur under a `when` clause. The connection between the two sets of rules is made by the rule $\mathcal{G}_d$, which is used in the translation of "`...`" and processes each of the associated `when` clauses. The difference between the two sets of rules is only in the use of the argument a, describing the rest of the SmPL code. For the $\mathcal{C}$ rules, a represents a pattern at the

[4] We define `true` as an abbreviation for $p() \vee \neg p()$, for an arbitrary predicate $p()$.

same level as the pattern being translated, which thus must be matched after the current pattern, while for the $\mathcal{G}$ rules, a represents the code that follows the associated dots, and thus only serves to delimit any dots that appear at the end of the `when` code. In particular, the rule $\mathcal{G}_t$ for Terms does not have an argument a, because a Term cannot end in dots.

We examine in more detail the $\mathcal{C}$ translation rules for dots, disjunctions, atomic patterns, and conditionals. These rules illustrate the main concepts of the translation process.

Dots Dots represent a sequence of arbitrary code, possibly constrained by `when` clauses, along a control-flow path. In the translation, the end of this sequence is indicated by the formula a that describes the rest of the SmPL code. Such a delimited path can be expressed by an "until" path operator, AU or EU. Our simplified language only supports universal quantification over paths, and thus we use AU.[5] The left argument of AU is constructed using the rule $\mathcal{G}_d$, which creates a formula checking that none of the patterns in the `when` clauses are matched within the path. The right argument of AU is the formula a matching the rest of the semantic patch.

Disjunction A disjunction $(S_1 \mid S_2)$ matches S_1 if possible, and otherwise S_2. The translation reflects the priority of S_1 over S_2 by encoding S_2 as $\neg(\mathcal{C}_s[\![S_1]\!]\, a) \wedge \mathcal{C}_s[\![S_2]\!]\, a$. This translation duplicates the previous processing of S_1. In the implementation, we use instead a "sequential disjunction" operator, that uses the negation of the previously computed result of processing S_1, and thus eliminates this duplication. The translation of a disjunction propagates the formula a separately into the translation of each branch. In this way, the rest of the semantic patch is matched at a point starting from the end of the code matching S_1 or S_2.

Atomic An atomic pattern may involve a transformation, as illustrated in line 13 of our type_ref semantic patch (Figure 1). Thus, the matching process must remember information about the position at which each atomic pattern matches. To collect this information, which is our fourth requirement for Coccinelle (Section 2.2), we use witnesses. Specifically, we introduce a new existentially quantified metavariable $\_v$ and create a predicate "$=$" that simply matches this variable to a textual representation of the atomic pattern, including any annotations about the transformation required. For example, the pattern `return -`C`;` with the annotation that `of_node_put(`n`);` should be inserted before it would be translated as follows:

$$\texttt{return -}C\texttt{;} \wedge \exists \_v.\_v = \texttt{"+ of\_node\_put(}n\texttt{);return -}C\texttt{;"}$$

The witnesses for $\_v$ will contain the current state and the binding of $\_v$ to the textual representation of the atomic pattern, indicating where and how to perform the transformation.

[5] Coccinelle automatically converts AU to AW, which can be defined using EU and negation, when the source program is found to contain a loop. AW can accept a path that goes around a loop infinitely without satisfying ϕ_2, but AW is implemented much less efficiently than AU.

$$
\begin{array}{l}
\exists n.(n \texttt{ = of\_find\_node\_by\_type(...)} \wedge \exists \_v.\_v = \ldots) \wedge \\
\quad \mathsf{AXA}[\mathsf{true}\ \mathsf{U} \\
\qquad ((\texttt{if (}n\texttt{ == NULL)} \mid \texttt{if (NULL == }n\texttt{)} \mid \texttt{if (!}n\texttt{)}) \wedge \exists \_v.\_v = \ldots) \wedge \\
\qquad \mathsf{AX}((\mathsf{trueBranch} \wedge \mathsf{AX}(\exists S.(S \wedge \exists \_v.\_v = \ldots))) \vee \\
\qquad\quad \mathsf{fallThrough} \vee \\
\qquad\quad (\mathsf{after} \wedge \mathsf{AXAX}(\mathsf{A}[\neg(\texttt{of\_node\_put(}n\texttt{);} \wedge \exists \_v.\_v = \ldots) \wedge \\
\qquad\qquad\qquad\qquad \neg(\exists n1.\exists f1.(n1 \texttt{ = } f1\texttt{(}n\texttt{,...)}) \wedge \exists \_v.\_v = \ldots) \wedge \\
\qquad\qquad\qquad\qquad \neg(\exists E1.(E1 \texttt{ = } n \wedge \exists \_v.\_v = \ldots)) \\
\qquad\qquad\qquad\qquad \mathsf{U} \\
\qquad\qquad\qquad\qquad \exists C.(\texttt{return -}C\texttt{;} \wedge \exists \_v.\_v = \ldots) \mid \\
\qquad\qquad\qquad\qquad (\texttt{of\_node\_put(}n\texttt{);} \wedge \exists \_v.\_v = \ldots) \mid \\
\qquad\qquad\qquad\qquad \exists n2.\exists f2.(n2 \texttt{ = } f2\texttt{(}n\texttt{,...)} \wedge \exists \_v.\_v = \ldots) \mid \\
\qquad\qquad\qquad\qquad \exists E2.(E2 \texttt{ = } n \wedge \exists \_v.\_v = \ldots) \mid \\
\qquad\qquad\qquad\qquad \exists E2.(\texttt{return } E2\texttt{;} \wedge \exists \_v.\_v = \ldots)])))\ \wedge \\
\qquad\quad \mathsf{EX}(\mathsf{after})]
\end{array}
$$

Figure 7. CTL-VW translation of the semantic patch type_ref. The right argument of the equality constraint on $\_v$ is elided in each case, for conciseness.

Conditionals The translation of a conditional is determined by the representation of a conditional in a Coccinelle control-flow graph. For conciseness, we omit further details. Note, however, that the translation of the header of a conditional introduces a metavariable $\_v$, as in the translation of an atomic pattern, thus making it possible to find and transform the matched code.

Figure 7 shows the result of translating the semantic patch type_ref into CTL-VW. The formula uses the "sequential disjunction" operator, denoted as $\mid$, that was described above. The pattern `n == NULL` has been expanded into a variety of ways to make a NULL test, using Coccinelle's *isomorphisms* [20]. For type_ref, the size of the formula is comparable to the size of the semantic patch. In general, however, subformulas can be duplicated 1) in the branches of a disjunction, 2) when a `when` clause ends in dots, and 3) in full SmPL to implement the shortest path constraint, although in the latter case, only the atomic terms immediately preceding and following the dots are duplicated. In practice, however, we have found that it is the complexity of the source code rather than the size of the formula that has an impact on the performance [20].

6.4 Matching and transformation

As illustrated by the CTL-VW formula in Figure 7, a formula resulting from the translation always has the following properties: 1) All metavariables are existentially quantified, 2) The existential quantifiers for the introduced metavariables $\_v$ recording the positions of the atomic terms are always innermost, and 3) The translation of an atomic term, including the quiantifier of its $\_v$ variable, is nested within quantifiers for all of its free metavariables. The first point implies that in the result of matching the CTL-VW formula against the source code, the environment component of each triple is always empty. The second and third points imply that the witnesses are trees in which the leaves are the bindings of the $\_v$ metavariables, representing the code to transform, and the path from the root of a witness to a given leaf contains the

```
np = of_find_node_by_type(NULL, "smu");                                  1
if (np == NULL)                                                          2
        return −ENODEV;                                                  3
printk(KERN_INFO "SMU: Driver %s %s\n", VERSION, AUTHOR);                4
if (smu_cmdbuf_abs == 0) {                                               5
        printk(KERN_ERR "SMU: Command buffer not allocated !\n");        6
        return −EINVAL;                                                  7
}                                                                        8
smu = alloc_bootmem(sizeof(struct smu_device));                          9
if (smu == NULL) {                                                       10
        return −ENOMEM;                                                  11
}                                                                        12
... // unrelated straightline code                                      13
smu−>of_node = np;                                                       14
```

Figure 8. An extract of the function smu_init in the file drivers/macintosh/smu.c

state: 1
environment: $\emptyset$
witnesses: $\{\langle 1, n, \texttt{np},$
$\{\langle 1, \_v, \texttt{n = of\_find\_node\_by\_type(...)}, \emptyset\rangle,$
$\langle 2, \_v, \texttt{if (}n\texttt{ == NULL)}, \emptyset\rangle,$
$\langle 3, S, \texttt{return -ENODEV;}, \{\langle 3, \_v, S, \emptyset\rangle\}\rangle,$
$\langle 7, C, \texttt{EINVAL}, \{\langle 7, \_v, \texttt{return -}C\texttt{;}, \emptyset\rangle\}\rangle,$
$\langle 11, C, \texttt{ENOMEM}, \{\langle 11, \_v, \texttt{return -}C\texttt{;}, \emptyset\rangle\}\rangle,$
$\langle 14, E2, \texttt{smu->of\_node}, \{\langle 14, \_v, E2\texttt{ = }n, \emptyset\rangle\}\rangle\}$

Figure 9. Result of applying the semantic patch of Figure 1 to the function smu_init. The state numbers correspond to the line numbers in Figure 8

bindings of the free metavariables that may be involved in the transformation process.

As a concrete example, Figure 8 shows an extract of the function smu_init in the file drivers/macintosh/smu.c,[6] which is matched by the type_ref semantic patch, and Figure 9 shows the single triple that results from the matching process. In Figure 9, we have replaced the state numbers generated by Coccinelle by the corresponding line numbers in Figure 8 for easy reference. The result shows that the various atomic terms of the semantic patch match on lines 1, 2, 3, 7, 11, and 14. Only the pattern `return -`C`;` encapsulates a transformation, as shown in Figure 1: adding the code `of_node_put(`n`);`. There are two matches of this pattern, on lines 7 and 11. For the former, tracing up from the $\_v$ to the root of the witness gives the binding of C to `EINVAL` and n to `np`. For the latter, this gives the binding of C to `ENOMEM` and n to `np`. In each case, this is sufficient information needed to carry out the transformation.

[6] This code comes from http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary, using the version of Linux from just before the commit bad5232ba266ae2c666c17be236152fb2d8ada3b, in which the first of the patches based on this semantic patch was accepted. This version dates from June, 2008. To simplify the example, we have added the braces around the return on line 11, so that it is valid to add code before this statement. Coccinelle automatically adds such braces, when needed, but for simplicity this issue is not addressed in the translation presented in Figure 6.

Based on the result of the matching process, Coccinelle collects the set of sequences of bindings leading from the root of a witness to each of its leaves that contains a transformation. Coccinelle only accepts a set of transformations if the states involved are disjoint and are only reachable from the states associated with the set of triples. These conditions help avoid ambiguity in the transformation process. If the set of transformations is acceptable, code is generated accordingly, in our case adding a call to `of_node_put` above lines 7 and 11 of Figure 8. Otherwise, Coccinelle aborts.

6.5 Experiments

The type_ref semantic patch shown in Figure 1 transforms four files in the Linux kernel: arch/powerpc/platforms/-pseries/nvram.c, drivers/macintosh/smu.c, drivers/macintosh/-therm_pm72.c, and drivers/video/fsl-diu-fb.c.[7] Figure 10 summarizes the sizes of these files and the performance of Coccinelle when applying the type_ref semantic patch to them. All experiments were performed on a 1.4GHz uniprocessor Centrino laptop with 635MB of RAM. Execution times are the average of five runs. The result of applying the semantic patch to nvram.c and smu.c has been integrated into the Linux kernel. The result for fsl-diu-fb.c has been validated by a Linux developer. We observed that the result of applying the semantic patch to therm_pm72.c is probably a false positive, due to an interprocedural effect that the semantic patch does not take into account. Nevertheless, the code structure in this case is typical, and thus it also serves as a representative example.

The files range in size from 149 to over 2000 lines of C code, including comments and whitespace. In each case there is a single function that is relevant to the semantic patch, and this function ranges in size from 27 to 94 lines. Coccinelle skips over irrelevant functions quickly, essentially only parsing them, and thus the CTL-VW algorithm is only applied to the relevant functions. Because the files are much larger than the relevant functions in this case, the time for parsing dominates the overall running time for most files. We can nevertheless get a sense of the performance of the CTL-VW algorithm in practice, by considering its performance on the relevant functions, in terms of both time and space usage.

We consider two implementations of the model checking algorithm for CTL-VW: an unoptimized one that closely follows the algorithm presented in Section 5.3, and the optimized one that is implemented in the Coccinelle tool. The optimizations affect both the encoding in CTL-VW and the execution of the algorithm, and are summarized as follows:

Optimizations to the translation These optimizations seek to reduce the number of witnesses, and thus the amount of information that the algorithm has to propagate.

- Some metavariables, such as *n1*, *E1*, and *f1* in Figure 1, only occur under a negation. The CTL-VW algorithm

[7] This code comes from the same source as mentioned in Footnote 6.

File	total lines	lines in the relevant function	unoptimized total time	unoptimized CTL time	unoptimized steps	unoptimized triples	optimized total time	optimized CTL time	optimized steps	optimized triples
nvram.c	149	27	0.309	0.009	77+29	541	0.306	0.005	67+14	261
therm_pm72.c	2279	32	1.466	0.010	77+29	678	1.457	0.006	67+12	276
fsl-diu-fb.c	1721	57	1.226	0.039	77+47	1093	1.205	0.011	67+27	532
smu.c	1323	94	7.707	6.818	77+81	2270	0.888	0.043	67+61	1176

Figure 10. The performance of applying the semantic patch of Figure 1 to the files that it affects in Linux. Times are in seconds. Steps is the number of CTL operators considered in processing the formula added to the number of steps involved in computing fixpoint iterations for the operator AU. Triples is the sum of the number of triples in the result of processing each CTL operator. The unoptimized and optimized variants both use an implementation of disjunction that does not duplicate the processing of subformulas.

drops witnesses created under a negation, so as an optimization, we augment the translation of Figure 6 to quantify such metavariables using a variant of $\exists$ that does not create any witnesses.

- Some patterns, such as *n1 = f1(n,...)*, are used in matching, but are not affected by the transformation. In this case, we do not need to introduce a metavariable $\_v$ to record where the pattern has matched, thus eliminating a further set of witnesses.

Optimizations to the algorithm These optimizations seek to reduce the number of triples that are manipulated.

- The CTL-VW formulas generated by our translation algorithm often have the form $\phi_1 \wedge \mathsf{AX}(\phi_2)$ (see Figure 7). Information about the processing of ϕ_1 is propagated into the processing of ϕ_2, to ensure that the states considered for the subformulas of ϕ_2 are reachable from the states at which ϕ_1 is satisfied, and the environments considered for the subformulas of ϕ_2 are compatible with the environments that satisfy ϕ_1.
- The path operator AU is implemented by an incremental algorithm that considers only the newly added triples on each iteration. Various optimizations are also integrated into the implementations of conjunction and negation, as compared to the specifications shown in Figure 3.
- There is no need to keep track of multisets of witnesses in the implementation, as these are just a device to facilitate the proofs of soundness and completeness; the set of witness is sufficient to support matching and transformation.

As shown in Figure 10, the combination of these optimizations reduces the CTL-VW processing time by up to over 150 times, due to reduced memory requirements, and the sum of the number of triples manipulated at each step by around 50%. In other work [20], we have created over 60 semantic patches based on collateral evolutions that have taken place in Linux 2.5 and Linux 2.6. In applying these semantic patches to a total of over 5800 relevant Linux files on a 3.4GHz uniprocessor Pentium 4 PC with 1024MB of RAM, the time for applying a semantic patch to a relevant file is rarely more than 0.5 seconds. Thus, our optimizations for CTL-VW model checking provide acceptable performance for interactive use without resorting to efficient but more complex encoding strategies such as BDDs [5].

6.6 An extension: Mixing path quantifiers

A advantage of CTL is the ability to mix universal and existential quantification over paths within a single formula. To conclude, we consider how full SmPL can take advantage of this facility. By default, SmPL uses universal quantification when a semantic patch performs transformation, as indicated by `-` and `+` annotations, and existential quantification when a semantic patch performs only searching (making it instead a *semantic match*), in which lines are annotated with `*` to indicate items of interest. These conventions can be overridden, by indicating `forall` or `exists` at the beginning of the rule. But the quantifier used for individual dots can also be controlled locally using `when forall` or `when exists`.

The semantic match shown in Figure 11 mixes path quantifiers. This semantic match searches for cases where there exists a dereference of the result of the Linux kernel memory allocation function `kmalloc` without first checking that the result is valid. As the semantic match marks lines using `*`, paths are existentially quantified by default. In lines 7-10, however, we would like to identify a conditional that always aborts if the allocated value is `NULL`, as subsequent dereferences are known to be safe. Thus, the dots in the body of this conditional are annotated with `when forall`.

7. Related Work

Lacey and De Moor write CTL-FV formulas to specify compiler optimizations [16]. In this setting, a CTL formula describes properties of the context of a term that allow an optimization such as constant propagation to be applied to the term. Åberg *et al.* used CTL to specify the transformations required to integrate the run-time system of the Bossa scheduling framework into the Linux kernel source code [1]. In both cases, the state representing code that is affected by the transformation is the one that satisfies the entire formula, so there

```
@kmalloc_ref@ expression x,E; identifier fld; statement S; @@

  x = kmalloc(...)
  ... when != x = E
      when != x->fld
(
  if ((x == NULL) || ...) {
    ... when forall
    return ...;
  } else S
|
* x->fld
)
```

Figure 11. A semantic patch that mixes universal and existential path quantification

is no need for witnesses. Furthermore, the specifications are fairly simple, so it is possible to write CTL code directly. De Moor *et al.* observed that full CTL was not always necessary for such specifications, and proposed universal regular path queries, which are based on regular expressions [9]. Such queries, however, do not permit mixing universal and existential path quantifiers and also do not permit specifying transformations within formulas.

Other approaches to bug finding that take program control-flow into account include Metal [10] and SDV [2]. Metal is based on specifications expressed as state machines. The expressiveness of state machines and CTL is incomparable. SDV focuses mainly on eliminating false positives by taking possible run-time values into account. Coccinelle does not currently address this issue.

Bohn *et al.* also propose a variant of CTL with universal and existentially quantified variables [4]. Their goal is to reason about properties involving variables that range over very large or infinite domains, and they provide a syntactic approach to model checking that allows constraints about such variables to be simplified in *ad hoc* ways. Their semantics is somewhat different than ours in that environments are explicit in the semantics of state operators, but integrated into the labelling function of a specialized version of the initial model in the semantics of path operators. Our use of constructive negation can be viewed as a restricted version of their more general predicates on variable values.

Since SmPL semantic patches can be translated into CTL-VW, they could also be written using CTL-VW directly. Doing so, however, would be very tedious for complex semantic patches, as already illustrated by Figure 7. The difficulty of creating CTL specifications has been recognized in other areas. Corbett *et al.* propose a high level language for describing desired properties of Java programs, for use with the Bandera model checking framework [8].

Jones and Hansen present a translation of a toy version of SmPL into CTL-V [13]. They concentrate on matching, and do not provide support for transformation, as is enabled by CTL-VW. They implement CTL-V by translation into CTL, at the cost of increasing the size of the formula exponentially in the size of Val.

8. Conclusion

In this paper, we have identified four requirements for a logic that is to serve as the foundation of a control-flow based program matching language, and we have incrementally derived from CTL a logic, CTL-VW, that meets these requirements. In practice, we have found the decision to base the implementation of Coccinelle on an extension of CTL very beneficial, as it naturally separates the specification of the semantics of the program matching language, represented by the translation into the logic, from the implementation, represented by the model checking algorithm. Indeed, in the case of Coccinelle, we iterated many times over the language semantics, but modified the implementation of the model checking algorithm only rarely, to improve performance. Our extensive experiments [20] show that the approach is efficient enough for practical use, on a standard PC.

A limitation of the formalization of CTL-VW presented here is that it does not allow for collecting witnesses under negation. While this feature is not needed in the examples we have considered in practice with Coccinelle, there could be other contexts in which such witnesses would be useful. We want to extend the formalization to accommodate them. Currently, only the proof of soundness and completeness of the CTL-V model checking algorithm has been validated by a proof assistant. We plan to validate the proof for the CTL-VW model checking algorithm as well.

Availability Coccinelle is available from the following URL:
`http://www.emn.fr/x-info/coccinelle/`.

References

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 196–204, Montreal, Canada, Oct. 2003. IEEE.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 73–85, Leuven, Belgium, Apr. 2006.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[4] J. Bohn, W. Damm, O. Grumberg, H. Hungar, and K. Laster. First-Order-CTL model checking. In *Foundations of Software Technology and Theoretical Computer Science*, number 1530 in Lecture Notes in Computer Science, pages 283–294, Chennai, India, Dec. 1998.

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[6] D. Chan. Constructive negation based on the completed database. In *Fifth International Conference and Symposium on Logic Programming*, pages 111–125, Seattle, WA, Aug. 1988.

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

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proceedings of the SPIN Software Model Checking Workshop*, number 1885 in Lecture Notes in Computer Science, pages 205–223, Stanford, CA, USA, Aug. 2000.

[9] O. de Moor, D. Lacey, and E. Van Wyk. Universal regular path queries. *Higher Order and Symbolic Computation*, 16:15–35, 2003.

[10] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Diego, CA, Oct. 2000.

[11] A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2–3):151–238, 1992.

[12] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[13] N. Jones and R. R. Hansen. The semantics of "semantic patches" in coccinelle: Program transformation for the working programmer. In *Fifth ASIAN Symposium on Programming Languages and Systems*, number 4807 in Lecture Notes in Computer Science, pages 303–318, Singapore, Nov. 2007.

[14] C. Kern and M. R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

[15] D. Lacey. *Program Transformation using Temporal Logic Specifications*. PhD thesis, Oxford University Computing Laboratory, 2003.

[16] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Compiler Construction, 10th International Conference, CC 2001*, number 2027 in Lecture Notes in Computer Science, pages 52–68, Genova, Italy, Apr. 2001.

[17] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Compiler optimization correctness by temporal logic. *Higher Order and Symbolic Computation*, 17(3):173–206, 2004.

[18] J. L. Lawall, J. Brunel, R. R. Hansen, H. Stuart, and G. Muller. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. Technical Report 08/1/INFO, Ecole des Mines de Nantes, Nantes, France, 2008.

[19] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, Jan. 2003. Unified Format section, `http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html`.

[20] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Eurosys 2008*, pages 247–260, Glasgow, Scotland, Mar. 2008.

[21] M. Sassa and S. Sahara. Validating correctness of compiler optimizer execution using temporal logic. In *Compiler Optimization meets Compiler Verification*, Budapest, Hungary, Apr. 2008.

[22] D. A. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *SAS '98: Proceedings of the 5th International Symposium on Static Analysis*, pages 351–380, London, UK, 1998. Springer-Verlag.

[23] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *International Conference on Software Engineering (ICSE)*, pages 172–181, Shanghai, China, May 2006.