The Semantics of "Semantic Patches" in Coccinelle: Program Transformation for the Working Programmer

Neil D. Jones[1] and René Rydhof Hansen[2]⋆

[1] DIKU (Computer Science Dept., University of Copenhagen, Denmark)
[2] Dept. of Computer Science, Aalborg University, Denmark

Abstract. We rationally reconstruct the core of the *Coccinelle* system, used for automating and documenting collateral evolutions in Linux device drivers. A denotational semantics of the system's underlying *semantic patch language* (SmPL) is developed, and extended to include variables. The semantics is in essence a higher-order functional program and so executable; but is inefficient and limited to straight-line source programs. A richer and more efficient SmPL version is defined, implemented by compiling to the temporal logic CTL-V (CTL with existentially quantified variables ranging over source code parameters and program points; defined using the staging concept from partial evaluation). The compilation is formally proven correct and a model check algorithm is outlined.

1 Introduction

A tedious, vital and frequently occurring software engineering job is to carry out *systematic updates to Device Driver code*, often referred to as *software evolution*. Many necessary changes are due to *collateral evolutions*: updates to a given driver that must be made as a consequence of current and substantial changes to library modules that the driver depends on. A change in the API of an external library procedure used by the given driver is a typical example; other common examples include changes in function signatures and data structures used by the driver. Finding all the places where collateral evolutions are needed and then performing the actual update even in a single driver is a non-trivial problem. Changes to accommodate a single library update may involve searching thousands of files and performing hundreds of code changes. This problem needs an automated solution, as it is too frequent and important to be left to inexperienced programmers with traditional text editing and update documentation. See [2, 3, 5–8].

The Coccinelle approach has demonstrated considerable pragmatic value. Coccinelle is an executable program transformer that has shown its utility, with satisfactory efficiency and expressivity, for large real application problems including

⋆ Supported by the *Coccinelle* project under the Danish Research Council for Technology and Production Sciences.

device driver code updates [17]. It develops and applies “Semantic Patch” notation, a concept that abstracts and generalises the practically well-established and frequently used “patches” well-known to the Linux kernel community. Semantic patches are described in the *semantic patch language* SmPL, a domain specific language inspired by the patch notation. In comparison with the usual Linux patches, SmPL is much more versatile and more firmly based in programming language semantics.

Coccinelle has several major components, including ways of *recognising* software patterns frequently occurring in source code (written in C or Java); means for *efficiently performing* the needed pattern recognition using a variant of the temporal logic CTL; and ways to *transform* the recognised code. See [17, 20] for more details and a wide range of applications.

Analysis: updating source code. The problem is to make consistent changes to a collection of source programs. An example is to change the way a central function or procedure is called, e.g., to add an extra argument to its parameter list. This requires changing both the function or procedure declaration, and all calls to it.

To avoid struggling from the outset with semantic details of programming languages such as C or Java we take a top-down approach to the problem of updating source programs. Transformation semantics is developed in a language-independent way, carefully side-stepping problems due to inessential but troublesome idiosyncrasies sometimes found in real languages. This approach is able to cope with real-world languages including C and Java [17, 20].

Linguistic tool: a transformation *language*, called SmPL in the Coccinelle system. A SmPL transformation consists of source language *patterns*, identifying the source language constructions to be changed; and *insertions and deletions,* marking the changes to be made.

System tool: a transformation *engine*. This has two inputs: the source program to be transformed, and the transformation. It produces as output an updated source program. The developments of this paper are based on the following assumptions supported by current practice:

1. We assume that the transformation only describes the part of the source program to be changed, as most of the source program will remain unchanged.
2. Source program insertions or deletions are mainly *order-preserving*, so major textual rearrangements are not needed.
3. There is a need for tokens with *large value ranges*, too large to be listed explicitly. A typical example is an identifier, for instance a variable name, a procedure name, or a constant.

It is essential that Coccinelle be *automatic* (run without human interaction) and *exhaustive* (find all possible places to apply a transformation). Further, the result of transformation should be predictable. Hence Coccinelle must also have a *minimally surprising semantics*, e.g., one free from unexpected pattern matches. As a corollary, Coccinelle must also detect *inconsistent transformation specifications* that perform different transformations, if read in different ways.

Contribution of the paper. This "theory-practice border" paper formalises an essential part of SmPL, thus providing a theoretical basis for what has already proven to be a pragmatic success. It is intended to clarify just what it is that semantic patches do (at least a subset of them), and to aid understanding some of the implementational and design challenges that are being met within the Coccinelle project.

Our main contribution is to rationally reconstruct the core of Coccinelle's semantic patch language SmPL, concisely and understandably clarifying a number of points in the core semantics. Our semantics compactly and explicitly describes a practical system, and has been implemented as a functional program.

Coccinelle has shown the utility of the temporal logic CTL [10] *as an intermediate language* to implement SmPL. (As with compiler intermediate languages, users need not know of or be aware of CTL.) In this paper we build *a theoretical bridge*, proving formally that the natural pattern-matching way to read SmPL patterns is equivalent to its CTL implementation.

Expressivity and efficiency of the SmPL patterns of [17] are quite satisfactory in practice. The notation is useful for working software engineers, as it does not require knowing temporal logic such as CTL formulas; or concepts from regular expressions, semantics, finite automata theory, or Prolog. Further, SmPL patterns are much more local than patterns in [11–14], with less emphasis on computational futures and pasts.

Related work. Directly related work on software updating includes [4, 11–14, 16–18] by university groups at Nantes (Muller, Padioleau, ...), Copenhagen (Lawall, Hansen, Jones, ...); Oxford (De Moor, Lacey,...); and Stony Brook (Liu, Stoller,...). Papers [4, 16] apply regular expressions to program transformation. Paper [17] is a practice-oriented description of Coccinelle's semantic patches; and [11–14] apply CTL to program transformation. Compared with [11, 13, 15, 19], the focus of Coccinelle is not compiler optimisation, but software updating. Coccinelle is intentionally *not semantics-preserving*, in contrast to compiler or program transformer works such as [11, 14, 15]. The reason: Coccinelle may be used to change program functionality, or to fix or to detect bugs.

Papers [11, 13] use notation $\mathrm{C} \Rightarrow \mathrm{C}'$ **if** ϕ where C is a pattern, C' is a replacement for C, and the enabling condition for applying the rewrite is given by a formula ϕ expressed in the temporal logic CTL-FV. Here ϕ may refer to the computational past or future, relative to the occurrence of C.

For reasons of efficiency and usability by a broad software engineering community, Coccinelle does not require familiarity with the sometimes rather subtle nuances of temporal logic. Instead, Coccinelle uses patterns with variables and the "..." operator (explained later) to localise transformation sites.

In our experience, enabling conditions for program transformation seem more naturally expressed using Coccinelle patterns than by using general CTL formulas ϕ. In principle SmPL may be less expressive than CTL, e.g., it's not clear how to express conditions for some classical compiler optimisations such as constant propagation or live variables. However, if desired, such effects can easily be achieved by using Coccinelle's general scripting framework, discussed in [20].

The Stratego transformation system [1] is less semantics- based than [11, 14, 15] but more powerful as a rewriting engine, allowing separation of the rewrite rules from strategies for their application.

Structure of the paper. The data of a program transformer is a *source program.* A Core-SmPL transformation maps a source program into a target program. Its semantics is first written in the style of denotational semantics or functional programming. For simplicity, a source programs is initially just a *linear sequence of abstract syntax trees*, each attributes such as syntactic type, lexical infomation (e.g., a procedure name or constant value), or application of a value operator (e.g., +, - or assignment).

A more general and realistic source program is a *control flow graph* or CFG: a finite directed graph with program control points as nodes, and whose branching expresses control flow transfers: control divergence, convergence, and loops.

The initial Core-SmPL semantics is extended to such source programs in a perhaps unexpected way: the temporal logic CTL is used as an intermediate language, invisible to the user. This use of CTL is formally proven equivalent to the denotational semantics for programs with linear structure.

A semantic extension is to add pattern (meta-)variables to Core-SmPL, significantly extending its expressivity. The full paper [9] has more details, proofs, and a model checking algorithm for the extended CTL-V.

2 Core-SmPL: A Core Language for Semantic Patches

In this section we introduce Core-SmPL, a rational reconstruction of the core of SmPL, and show how it can be used to search for code patterns and to transform programs. In the terminology of the Coccinelle project such specifications are called *semantic patches* which is also the name we adopt in the following.

Syntax of source and target programs. We begin with a "linear source program" as a working abstraction of "source program". Later, it will be extended to include not just linear sequencing, but an arbitrary control flow graph or CFG with tests, divergence, convergence and loops.

Definition 1. *A* ground term *is a tree structure built from operators. A* linear source program *is a sequence of ground terms. Syntax is straightforward:*

$$
\begin{array}{ll}
S ::= G_1 G_2 \cdots G_n & \textit{A program is a sequence of ground terms} \\
G ::= op(G_1, \ldots, G_k) & k = arity(op) : \textit{Op.'s with right numbers of arguments.}
\end{array}
$$

A ground term is a variable-free tree structure built by operators from leaves. Technically a leaf is a 0-ary operator, and may be: a programming language constant; a name, e.g., a program variable or a function name; or a keyword without arguments. Nonleaf operators have positive arities, i.e., 1 or more arguments. Example nonleaf operators include `+`*,* `-`*,* `:=` *(assignment) or* `if`*. For compactness in presentation and examples, we write sequences (inputs to and outputs from our program transformer) without separators, and in infix notation.*

A table that summarises the operators and arities used in the examples:[3]

Operator	a	b	c	d	e	f	{	}	distance	rate	time	step	+	*	:=
Arity	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2

Symbols from a fixed alphabet such as $\texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{step}$ above are a special case: operators with arity 0. A program with only 0- ary operators is a string over a finite alphabet, as studied for decades in formal language and automata theory.

In real programming languages such as C or Java, the terms are subclassified into syntactic categories such as expression, command, or function declaration; but such distinctions will not be needed in this paper. (Such a classification would be called a *grammar* in compiler terminology, or a *signature* in algebra.)

Definition 2. *A* general source program*, or* CFG*, is a binary relation* $\rightarrow$ *on a finite set of* control states *(i.e., program points), each labelled by a ground term.*

The concrete syntax used for semantic patches in the Cocinelle system is similar to but extends the notation used by the `patch` program to specify a program transformation. This patch notation is the de facto standard for communicating proposed changes and updates among the Linux Kernel developers.

$P ::= \varepsilon$	Pattern that matches the empty sequence of terms
$\mid\ EP$	A match for E followed by a match for P
$E ::= T$	Pattern that matches a term T
$\mid\ (P_1 '\|' P_2)$	Match P_1 or P_2
$\mid\ ...$	Match a sequence of zero, one, or more arbitrary terms
$\mid\ -T$	Delete one T: match it, but do not copy it to the output
$\mid\ +T$	Insert T in the output sequence (no matching occurs)
$T ::= x$	A term is like a ground term, but may contain variables
$\mid\ op(T_1, \ldots, T_k)$	$k = arity(op)$: Must have the right numbers of arguments.
$x\ ::= variable$	A pattern variable

Figure 1: Syntax of Core-SmPL semantic patches

The pattern "`...`" matches *any sequence of terms.* This common pattern may be familiar from the patch notation used in the output of the `diff` utility. The variables appearing in a term T not to be confused with source or target program variables; they are *pattern variables* used for matching, essentially the variables or parameters used in [13, 11, 4, 16].

Some Core-SmPL semantics examples. $\mathcal{T}[\![P]\!](in)$ is the set of target programs that can be obtained by applying pattern P to transform source program in. In general, $\mathcal{T}[\![P]\!](in) = \{out_1, out_2, \ldots, out_n\}$ means that pattern P can transform source program in into any one target program in the set $\{out_1, out_2, \ldots, out_n\}$.

[3] Braces {, } delimit groups of (well-nested!) commands or statements.

Examples with only 0*-ary operators and no pattern variables.* A special case of a source or target program is a string of symbols (i.e., 0-ary operators) over a finite alphabet A. The first example recognises strings over an alphabet $A \supseteq \{\mathtt{a}, \mathtt{b}\}$. The pattern `...abab...` matches strings that contain `abab` as a substring. Viewed as a string transformer, pattern `...abab...` computes the identity transformation on strings that contain `abab` as a substring. It yields the empty set if applied to strings of other forms.

The pattern `...a-ba-b+e+f...` also matches source program strings containing `abab`, but the target string is constructed by deleting the two matched `b`'s from the source, and inserting symbols `e,f` just after the matched part `abab`.

Examples: $\mathcal{T}[\![Pattern]\!]$ *(Source- program) = set of transformed programs.*

1. $\mathcal{T}[\![$`...abab...`$]\!]$ `(abcd)` = $\emptyset$
2. $\mathcal{T}[\![$`...abab...`$]\!]$ `(cababababd)` = {`cababababd`}
3. $\mathcal{T}[\![$`...a-ba-b+e+f...`$]\!]$ `(cababd)` = {`caaefd`}
4. $\mathcal{T}[\![$`...a-ba-b+e+f...`$]\!]$ `(cababgababd)` = {`caaefgaaefd`}
5. $\mathcal{T}[\![$`...a-ba-b+e+f...`$]\!]$ `(cabababd)` = {`caaefababd, cabaaefabd, cababaaefd`}

Discussion. For software updating it is important that *all matches* are detected (e.g., if a function's calling mode is to be changed it is vital that all calls be changed to the new format). Example 1 does not match, so the semantics yields the empty set on input `abcd`. Example 2 has three matches in all, but no transformation occurs due to the absence of `+` or `-`. Thus the output is a singleton set, containing only the input sequence. Example 3 removes two `b`'s and adds `ef`. In Example 4 two patterns `abab` are discovered; for each, two `b`'s are removed, and `ef` is added. In examples 3 and 4 all matches are found and the transformation results are well-defined since unique.

Example 5 is problematic as three patterns `abab` are discovered, two of them overlapping. As a result there are in all *three possible* transformed programs. The Coccinelle system only transforms in case $n = 1$ in output $\{out_1, out_2, \ldots, out_n\}$, i.e., the effect of the transformation must be uniquely defined.

Examples with pattern variables and k-ary operators. Pattern variables are used to "remember" bits and pieces of the source program and, as it later will be seen, to match positions in the input program. Pattern variables are needed to express realistic source language patterns that contain possibly unbounded data such as function names, parameter names or constants. The Core-SmPL semantic patch notation allows (meta-) variables whose values come from such ranges, and allow testing the source program for equality of such values.

The source language term `distance := rate * time` can be matched with pattern x `:=` y `*` z by an environment that binds pattern variables x, y, z to corresponding bits of the source program, e.g.

$$env = [x \mapsto \texttt{distance}, y \mapsto \texttt{rate}, z \mapsto \texttt{time}]$$

$$\mathcal{T}[\![x \texttt{ := } y\texttt{*}z]\!](\texttt{distance := rate * time}) = \{\texttt{distance := rate*time}\}$$
$$\mathcal{T}[\![x \texttt{ := } x\texttt{+}y]\!](\texttt{distance := distance + step}) = \{\texttt{distance := distance + step}\}$$
$$\mathcal{T}[\![x \texttt{ := } x\texttt{*}y]\!](\texttt{distance := rate * time}) = \emptyset$$
(the empty set, since $\texttt{distance} \neq \texttt{rate}$)

3 Core-SmPL: Executable Transformation Semantics (without pattern variables)

We formalise the meaning of semantic patches by a directly executable semantics for Core-SmPL. This resembles a matcher for regular expressions over strings of terms, extended with *tree transformation* and *variable bindings*. We first develop the semantics for a simplified source language where there are no pattern variables, and a program is simply a string of ground terms, e.g., symbols. We will later generalise to allow variables in patterns, and programs with control transfers such as conditionals and loops.

The Core-SmPL semantic patch semantics is built by adding a transformation component to a string matcher written in continuation-passing style. Its input is a finite term string *in* from the set $GroundTerm^*$, the set of finite strings of ground terms. Its output is the set of all outputs corresponding to *in*: a set $out \subseteq GroundTerm^*$. The set *out* is empty if *in* does not match the pattern.

In the domain definitions of Figure 2 c is a continuation and a pattern meaning is an input-output transformation defined using continuation transformers.

$$in \in In = GroundTerm^* \quad out \in Out = 2^{GroundTerm^*}$$
$$c \in Cont = In \to Out$$
$$\mathcal{T}[\![_]\!] : P \to Cont$$
$$\mathcal{P}[\![_]\!] : P \to Cont \to Cont$$
$$\mathcal{E}[\![_]\!] : E \to Cont \to Cont$$

Figure 2: Semantic value domains

Figure 3 contains evaluation rules in a continuation- passing style denotational semantics. This formulation enables a natural and straightforward formalisation of searches for *all* possible matches for a given pattern. In addition, such a formulation lends itself to implementation in a functional language and indeed we have made such a prototype implementation.

Nonterminal P stands for "pattern" and G stands for any ground term. To avoid ambiguity we use ML-like notations to write inputs to and outputs from our program transformer: the empty sequence is represented as [], and $G :: in$ represents the result of putting ground term G at the start of input string *in*.

- I starts the transformation, with an initial continuation c_0 that will copy any input that may remain.
- II and III resemble a regular expression matcher, expressed using continuation semantics (it's easy to add a rule for P^* in a way similar to "...").

– III checks to see that the first ground term in the input sequence is G. If so, continuation c is applied to the remaining input, and G is added to each output term sequence. If not, no output is produced.
– IV. Deletion works just as $\mathcal{E}[\![G]\!]\ c\ in$ in group II, except that term G is not added to the output sequence. Insertion: term G is added to the output sequence. (No matching is done.)

$$
\begin{array}{ll}
I: & \\
\mathcal{T}[\![P]\!] & = \mathcal{P}[\![P]\!]\ c_0 \quad \underline{\text{where}}\ \ c_0\ in = \{in\} \\
\\
II: \textbf{Sequences of things} & \\
\mathcal{P}[\![\varepsilon]\!]\ c\ in & = (c\ in) \\
\mathcal{P}[\![\ E\ P\ ]\!]\ c & = \mathcal{E}[\![E]\!]\ (\mathcal{P}[\![P]\!]\ c) \\
\\
III: \textbf{Single things} & \\
\mathcal{E}[\![G]\!]\ c\ [] & = \emptyset \\
\mathcal{E}[\![G]\!]\ c\ (G' :: in) & = \underline{\text{if}}\ G = G'\ \underline{\text{then}}\ \{G :: out \mid out \in (c\ in)\}\ \underline{\text{else}}\ \emptyset \\
\\
\mathcal{E}[\![\ P_1 \mid P_2\ ]\!]\ c\ in & = (\mathcal{P}[\![P_1]\!]\ c\ in) \cup (\mathcal{P}[\![P_2]\!]\ c\ in) \\
\\
\mathcal{E}[\![...]\!]\ c\ in & = (c\ in)\ \cup \{G :: out \mid G :: in' = in \text{ and } out \in (\mathcal{E}[\![...]\!]\ c\ in')\} \\
\\
IV: \textbf{Deletion, insertion} & \\
\mathcal{E}[\![-G]\!]\ c\ [] & = \{\} \\
\mathcal{E}[\![-G]\!]\ c\ (G :: in) & = \underline{\text{if}}\ G = G'\ \underline{\text{then}}\ (c\ in)\ \underline{\text{else}}\ \emptyset \\
\mathcal{E}[\![+G]\!]\ c\ in & = \{G :: out \mid out \in (c\ in)\}
\end{array}
$$

Figure 3: Semantic evaluation rules

4 A practically better approach: compiling SmPL to CTL

The semantics above explains the meanings of SmPL patterns, and can be executed. However Figure 3 applies only to abstract syntax trees, as is usual in denotational semantics. In effect, it makes the unrealistic assumption that a source program is one long ground term sequence.

It also suffers efficiency problems: matching as above is essentially “top-down”, repeatedly checking the same goals in slightly different contexts due to non-linear uses of argument c. Pattern expression matching can be complex and time-consuming, especially if universal path quantification is used (see [4, 16]).

Because of these and other problems, Coccinelle instead uses instead a two-step approach: SmPL patterns are translated into the temporal logic CTL. This happens “under the hood”: users need not know anything about CTL, model checking, etc. We will argue the equivalence of the denotational semantics with the more indirect CTL-based version after a quick review of CTL.

CTL is defined in terms of *transition systems*: directed graphs able naturally to express program control flow graphs (CFGs) with flow divergence, convergence and loops. Compiling into CTL thereby also allows a smooth extension to program control flow graphs, an extension done less systematically in [4, 16].

An immediate advantage is performance: model checkers are known to be fast, with a well-developed theory and practice. Since model checking is done bottom-up, repeated computation is avoided. A further advantage is that the interaction between universal and existential quantification over paths is well-defined in temporal logic, e.g., it does not in principle require extra work to generalise to patterns with alternating path quantifiers.

A final advantage is flexibility: the same CTL language can be used as an intermediate language with different translation schemes. This makes it easier to adapt the Coccinelle approach to applications other than updating and transformation, e.g., bug finding [20].

In the remainder of this paper we mainly focus in using CTL model checking to search for program patterns rather than program transformation. This is motivated by the way Coccinelle works: first model checking is used to find all the relevant program points and then the transformations are performed afterwards. This has proven to be a simple way to avoid ambiguous transformations. It also has the practical advantage that it is significantly simpler to formulate the correctness statements without the transformation component. Extension of CTL (e.g., with transformation operators '$+$' and '$-$' giving judgements of the form $\mathcal{M}, s \models \phi \rightarrow \mathcal{M}'$) will be described in a subsequent publication.

Compiling SmPL into CTL. We now translate SmPL into CTL instead of executing. To save space we do not repeat the standard semantics of CTL but refer instead to [10]. We will prove that the Core-SmPL semantics of Section 3 is a *symbolic composition* of this transformation semantics with the CTL semantics. For now we use classical CTL without variables, so the T appearing below is an atomic proposition in AP: a ground term as in Definition 2. We show later how to allow variables in CTL terms, an idea also used in [13, 11]. To simplify the correctness formulation, we do not here account for transformation by `+` or `-`.

Compilation is defined in Figure 4 using functions $\mathcal{T}_{ctl} : P \rightarrow CTL$, $\mathcal{P}_{ctl}[\![_]\!] : P \rightarrow K \rightarrow CTL$ and $\mathcal{E}_{ctl}[\![_]\!] : E \rightarrow K \rightarrow CTL$ (note that the CTL and K are also used as types in the figure). Data structure $k \in K$ is related to the continuation functions of the executable semantics of Section 3. For pragmatic reasons, the K data structure distinguishes between two kinds of continuations, denoted `tail` and `after`, representing respectively continuations that are final and continuations that need further work. We defer detailed explanation to [9].

Correctness of the compilation to CTL. We now argue the translation correct by relating the executable semantics of Section 3 to CTL satisfaction of a translated term. As we only consider patterns P without $+$ or $-$, the net semantic effect of $\mathcal{T}[\![P]\!]\ in$ is to transform input in into either $\{in\}$ or $\emptyset$. To state correctness we first define a link between input sequences and transition systems.

$$
\begin{array}{lll}
k : K = \texttt{tail} \mid \texttt{after}\ CTL & & \\
 & & \\
\mathcal{T}_{ctl}[\![P]\!] & = \mathcal{P}_{ctl}[\![P]\!]\ \texttt{tail} & \\
 & & \\
\mathcal{P}_{ctl}[\![\varepsilon]\!]\ \texttt{tail} & = \texttt{true} & \\
\mathcal{P}_{ctl}[\![\varepsilon]\!]\ (\texttt{after}\ \phi) & = \phi & \\
 & & \\
\mathcal{P}_{ctl}[\![E\ P]\!]k & = \mathcal{E}_{ctl}[\![E]\!](\texttt{after}(\mathcal{P}_{ctl}[\![P]\!]k)) & \\
 & & \\
\mathcal{E}_{ctl}[\![G]\!]\ \texttt{tail} & = G & \text{ground term } G \text{ regarded as atomic prop.} \\
\mathcal{E}_{ctl}[\![G]\!]\ (\texttt{after}\ \phi) & = G \wedge AX\phi & \\
 & & \\
\mathcal{E}_{ctl}[\![P_1 \mid P_2]\!]k & = (\mathcal{P}_{ctl}[\![P_1]\!]k) \vee (\mathcal{P}_{ctl}[\![P_2]\!]k) & \\
 & & \\
\mathcal{E}_{ctl}[\![...]\!]\ \texttt{tail} & = AF\ \texttt{exit} & \text{end of the input } (\texttt{exit} \text{ is in Definition 3}) \\
\mathcal{E}_{ctl}[\![...]\!]\ (\texttt{after}\ \phi) & = AF\ \phi & \text{all future states must satisfy } \phi
\end{array}
$$

Figure 4: Translation from SmPL into CTL

Definition 3. *Let $in = G_1G_2 \ldots G_n$ be a linear source program: a sequence of ground terms. The corresponding transition system (Figure 5) is denoted $\widehat{in}$. This has states $1, 2, \ldots, n, n{+}1$ with labels $L(1) = \{G_1\}, \ldots, L(n) = \{G_n\}, L(n{+}1) = \{\texttt{exit}\}$ and transitions $\{1 \rightarrow 2, \ldots, n \rightarrow n+1, n+1 \rightarrow n+1\}$.*

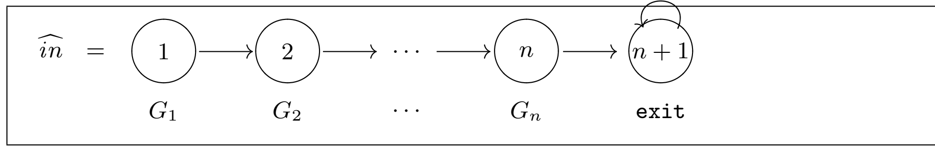

Figure 5: Model for a linear string as source program

Theorem 1. *For any linear source program in and pattern P without $+, -$ or variables, we have $\mathcal{T}[\![P]\!]\ in = \{in\}$ if and only if $\widehat{in}, 1 \models \mathcal{T}_{ctl}[\![P]\!]$.*

A definition aids stating a sufficiently strong induction hypothesis:

Definition 4. *Relation $c \approx k$ holds if $k = \texttt{tail}$ and $\forall in\ (c(in) = \{in\})$, or if $k = \texttt{after}\ \phi$ and $\forall in\ (\ c(in) = \{in\}$ if and only if $\widehat{in}, 1 \models \phi\ )$.*

We prove this by structural induction on P. The desired result follows by structural induction on P, E, using the definitions of $\mathcal{P}, \mathcal{P}_{ctl}, \mathcal{E}, \mathcal{E}_{ctl}$ and the following. See the full paper [9] for detailed proof.

Theorem 2. *If $c \approx k$ it holds that $\forall P.\ (\ \mathcal{P}[\![P]\!]\ c \approx \texttt{after}(\mathcal{P}_{ctl}[\![P]\!]\ k)\ )$ and $\forall E.\ (\ \mathcal{E}[\![E]\!]\ c \approx \texttt{after}(\mathcal{E}_{ctl}[\![E]\!]\ k)\ )$.*

Relating regular expressions and CTL. A natural question: can the translation be extended to allow an arbitrary regular expression in place of P? Alas, there seems to be no natural way to translate a general regular pattern P^* into CTL.

5 Semantics of Core-SmPL with pattern variables

We now enrich Core-SmPL, extending the language of patterns to include pattern variables (essentially the parameters of [16, 4] or meta-variables of [11, 12, 14]). An *environment* parameter holds the values bound to pattern variables.

$$\begin{array}{l} \mathit{In} = \mathit{GroundTerm}^* \quad \mathit{Out} = 2^{\mathit{GroundTerm}^*} \\ c : \mathit{Cont} = \mathit{Env} \rightarrow \mathit{In} \rightarrow \mathit{Out} \quad (c \text{ is a continuation}) \\ \mathcal{T}[\![_]\!] : P \rightarrow \mathit{In} \rightarrow \mathit{Out} \quad \mathcal{P}[\![_]\!] : P \rightarrow \mathit{Cont} \rightarrow \mathit{Cont} \quad \mathcal{E}[\![_]\!] : E \rightarrow \mathit{Cont} \rightarrow \mathit{Cont} \end{array}$$

Figure 6: Semantic value domains for Core-SmPL with variables

The input to a Core-SmPL semantic patch is still a finite sequence $in = G_1 G_2 \ldots G_n \in \mathit{GroundTerm}^*$ of ground terms $G_i \in \mathit{GroundTerm}$, and the matcher output is a set of such sequences: a set $out \subseteq \mathit{GroundTerm}^*$, empty if in does not match the pattern. Pattern semantics has to be extended, though, to include bindings of pattern variables. Operations on environments: $env(T)$ denotes the result of replacing every pattern variable x in T by $env(x)$. $env(T)$ is defined only if every $env(x)$ is defined. Updating the environment env with env' is denoted by $env[env']$, i.e., $env[env'](x) = env'(x)$ if $x \in \mathrm{dom}(env')$ and $env[env'] = env(x)$ otherwise. (Note that $\mathrm{dom}(env[env']) = \mathrm{dom}(env) \cup \mathrm{dom}(env')$.)

Further (as in Prolog), $MGU(T_1, T_2)$ denotes the *most general unifier* of T_1, T_2. Notation: $MGU(T_1, T_2)$ equals "some env" where env is the most general unifier env if it exists, else $MGU(T_1, T_2)$ equals "fail". For SmPL, T_1 may contain pattern variables, but T_2 will always be a ground term. Here $\mathit{GroundTerm}^*$ and Term^* mean any finite sequence of ground terms and terms respectively.

- I starts, with empty variable environment env_0 and initial continuation c_0.
- II is just as before except for the extra environment parameter.
- III yields the empty output set on empty input. Otherwise, the first input ground term G is matched against pattern T (after applying the current environment to instantiate its pattern variables). If matching succeeds with env', new bindings found in env' are added to the current environment env. An example: pattern $x{:=}x{+}y$ is successfully matched against program input $\texttt{di} := \texttt{di} + \texttt{st}$ to give new environment bindings $[x \mapsto \texttt{di}, y \mapsto \texttt{st}]$:

$$\begin{array}{l} \mathcal{E}[\![x{:=}x{+}y]\!]\ c\ []\ (\texttt{di} := \texttt{di} + \texttt{st}){::}in = \\ \qquad \{(\texttt{di} := \texttt{di} + \texttt{st}){::}out \mid out \in c[x \mapsto \texttt{di}, y \mapsto \texttt{st}]\} \end{array}$$

- IV. Deletion and insertion are as for Core-SmPL, except the environment is applied to term T as in II.

An implementation. These rules have been implemented in a functional programming language, and gave the expected outputs on all this paper's examples. See the full paper [9] for details.

$$
\begin{array}{ll}
I: \quad \mathcal{T}[\![P]\!] & = \mathcal{P}[\![P]\!]\ c_0\ env_0\ \ \underline{\text{where}}\ \ \text{dom}(env_0) = \emptyset\ \ \text{and} \\
 & \quad c_0\ env\ in = \{in\} \\
\hline
II: \textbf{Sequences of things} & \\
\mathcal{P}[\![\varepsilon]\!]\ c\ env\ in & = c(in) \\
\mathcal{P}[\![\ E\ P\ ]\!]\ c & = \mathcal{E}[\![E]\!]\ (\mathcal{P}[\![P]\!]\ c) \\
\hline
III: \textbf{Single things} & \\
\mathcal{E}[\![T]\!]\ c\ env\ [] & = \{\} \\
\mathcal{E}[\![T]\!]\ c\ env\ (G :: in) & = \underline{\text{case}}\ MGU(env\ T, G)\ \underline{\text{of}} \\
 & \quad \text{fail} \quad\quad\ : \{\} \\
 & \quad \text{some}\ env' : \{G :: out \mid out \in (c\ env[env']\ in)\} \\
 & \\
\mathcal{E}[\![\ P_1 \mid P_2\ ]\!]\ c\ env\ in & = (\mathcal{P}[\![P_1]\!]\ c\ env\ in) \cup (\mathcal{P}[\![P_2]\!]\ c\ env\ in) \\
 & \\
\mathcal{E}[\![...]\!]\ c\ env\ in & = (c\ in) \cup \\
 & \quad \{G :: out \mid G :: in' = in\ \text{and}\ out \in (\mathcal{E}[\![...]\!]\ c\ env\ in')\} \\
\hline
IV: \textbf{Deletion, insertion} & \\
\mathcal{E}[\![-T]\!]\ c\ env\ [] & = \{\} \\
\mathcal{E}[\![-T]\!]\ c\ env\ (G :: in) & = \underline{\text{case}}\ MGU(env\ T, G)\ \underline{\text{of}} \\
 & \quad \text{fail} \quad\quad\ : \{\} \\
 & \quad \text{some}\ env' : c\ env[env']\ in \\
\mathcal{E}[\![+T]\!]\ c\ env\ in & = \{(env\ T) :: out \mid out \in (c\ env\ in)\}
\end{array}
$$

Figure 7: Semantic evaluation rules with variables

6 Semantics of CTL-V with pattern variables

The Coccinelle implementation translates SmPL patterns with variables into CTL-V: a CTL extension with quantified variables ranging over fragments from the source program's CFG. The correctness argument of Section 4 was expressed in terms of classical, variable-free, CTL, so some changes are necessary to express correctness of the more general SmPL with pattern variables.

CTL-V = Staged CTL with quantifiers, a variant intended to be especially suitable for program manipulation. One extension over classical CTL is (as in Definition 2) to allow atomic propositions *ap* to have full tree-structured terms as values. The idea is to extend traditional models by allowing a state to be decorated with pieces of source program information, e.g., possibly unbounded data such as function names, parameter names or constants.

These are referred to using *pattern variables* so only a term's top-level syntactic structure need be expressed: a CTL-V atomic proposition may be an arbitrary term, with or without variables. This generalises an approach seen in

[13, 12, 11]. (Variables used in a similar way are called *parameters* in [4, 16].) We generalise a bit to allow *explicit quantification*, with existential quantifiers appearing anywhere in a formula.

CTL-V syntax and its satisfaction relation. For brevity we just show how CTL-V pattern recognition works, and omit details of how the language and algorithms are extended to carry out program transformation. The development is intended only to clarify the CTL-V semantics, and does not at all account for efficiency issues (e.g., as done in the Coccinelle system).

Definition 5. *Let x range over Var, a set of* variables[4]. *A* CTL-V formula *is anything generated by the following context-free grammar, where $ap \in AP$ may be a term containing variables:*

$$\phi ::= ap \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid AX\phi \mid AF\phi \mid A(\phi U \phi) \mid EX\phi \mid \exists x\phi$$

By Definition 2 the CFG of a source program P is a binary relation $\rightarrow$ on states, each labelled by a single ground term G. A pattern-variable value will typically be a fragment of the source program P to be analysed. The set of all possible values is thus the set of all subterms of P , and so *a finite set.* We will henceforth denote this set by $Val = \{v_1, \ldots, v_m\}$.

Before starting with CTL-V-satisfaction and model checking, we need precisely to define the working of substitutions that bind pattern variables. A substitution binds the free variables of a CTL-V-formula ϕ to values in Val. A term atomic proposition T is true iff T can be unified with G.

Definition 6. *The* set of free variables $fv(\phi)$ *of CTL-V formula ϕ is defined as expected. A formula ϕ is* closed *if $fv(\phi) = \emptyset$. A* substitution *is a partial function $\theta : FinSet(Var) \rightarrow Val$ mapping a finite set of CTL-variables to values.*

Definition 7. *The* satisfaction *relation $\mathcal{M}, s \models_\theta \phi$ for CTL-V is defined inductively in Figure 8. ($\mathcal{M}$ is elided for brevity.)*

$s \models_\theta T$	iff	some $\theta = MGU(T, v)$ where $L(s) = \{v\}$
$s \models_\theta \neg\phi$	iff	not $s \models_\theta \phi$
$s \models_\theta \phi_1 \wedge \phi_2$	iff	$s \models_\theta \phi_1$ and $s \models_\theta \phi_2$
$s \models_\theta \phi_1 \vee \phi_2$	iff	$s \models_\theta \phi_1$ or $s \models_\theta \phi_2$
$s \models_\theta AX\phi$	iff	$\forall\sigma \in \mathbb{P}(s) \; . \; \sigma[1] \models_\theta \phi$
$s \models_\theta EX\phi$	iff	$\exists\sigma \in \mathbb{P}(s) \; . \; \sigma[1] \models_\theta \phi$
$s \models_\theta A(\phi_1 U \phi_2)$	iff	$\forall\sigma \in \mathbb{P}(s) \; . \; \exists j \geq 0 \; .$
		$[\forall k \; . \; 0 \leq k < j \;\Rightarrow\; \sigma[k] \models_\theta \phi_1] \wedge \sigma[j] \models_\theta \phi_2$
$s \models_\theta AF\phi$	iff	$\forall\sigma \in \mathbb{P}(s) \; . \; \exists j \geq 0 \; . \; \sigma[j] \models_\theta \phi$
$s \models_\theta \exists x\phi$	iff	$s \models_{\theta[x \mapsto v_1]} \phi$ or $\ldots$ or $s \models_{\theta[x \mapsto v_m]}$

Figure 8: CTL-V satisfaction relation

[4] These are names of pattern variables, not program variables.

Staging. The "silver bullets" of this approach: pattern (meta-)variables, quantification, and the use of two stages. The term "staging" comes from partial evaluation and refers to the *binding times*, i.e., the times at which various things are specified or computed. A key point is that source program-dependent values such as identifiers, although unbounded if one consider arbitrary programs, have a *bounded finite value range* **for any one source program**. Hence Val is a finite value set for the program about to be transformed.

Mapping CTL-V to CTL. Recall that $Val = \{v_1, \ldots, v_m\}$ and consider the following mapping from CTL-V to CTL:

$$[\![T]\!]\theta = \theta(T) \qquad [\![\phi \wedge \phi']\!]\theta = [\![\phi]\!]\theta \wedge [\![\phi']\!]\theta \qquad [\![\neg\phi]\!]\theta = \neg([\![\phi]\!]\theta)$$
$$[\![\exists x \phi]\!]\theta = [\![\phi]\!]\theta[x \mapsto v_1] \vee \ldots \vee [\![\phi]\!]\theta[x \mapsto v_m]$$

The following theorems establish the correctness of the above mapping and decidability of CTL-V model checking respectively:

Theorem 3. *For any $\mathcal{M}, s$ and θ that closes ϕ: $\mathcal{M}, s \models [\![\phi]\!]\theta$ iff $\mathcal{M}, s \models_\theta \phi$.*

Theorem 4. *It is decidable, given Kripke model $\mathcal{M} = (S, \rightarrow, L)$, state $s \in S$, substitution θ and CTL-V formula ϕ, whether $\mathcal{M}, s \models_\theta \phi$.*

In [9] we show a model check algorithm for CTL-V that works because of staging and the corollary finiteness of *Val*. It sidesteps some tricky algorithmic problems involved in an efficient way to implement $\neg\phi, \exists\phi$, as was necessary in [13, 16] (and is also done in Coccinelle).

7 Relation to the Coccinelle System

We have made a rational reconstruction of the core of the Coccinelle system. We now briefly review how the real Coccinelle system differs from, and extends, our reconstruction. The most important difference: this paper does not cover the full semantic patch language (SmPL) implemented by Coccinelle.

Other differences are mainly concerned with implementation and issues relating to the underlying models, such as nesting of program structures and matching balanced braces. These particular issues are handled by adding a special atomic proposition, called `Paren`(x). The `Paren`(x) proposition is true at some state if the variable x equals the *current nesting level* of program braces. This makes it possible to constrain searches to specific function definition bodies or program block structures, e.g., to skip over the "then" branch of a conditional.

Efficiency issues. The Coccinelle system implements a number of optimisations in order to obtain acceptable execution times. These include use of constructive negation for a more efficient implementation of $\exists$ than in Definition 7; reducing the scope of quantifiers; and a number of low-level implementation techniques. Constructive negation directly encodes "negative information" about variable bindings, i.e., recording that a given variable must not be bound to a certain value. Reducing the scope of quantifiers has the effect of reducing the size and number of environments that have to be propagated by the algorithm.

In practise these optimisations have had a profound effect on execution times.

Transformation after model checking. In order to perform program transformations based on successful matches obtained by model checking, the Coccinelle system adds so-called *witness trees* to the CTL-V semantics. These record the variable bindings (substitutions) that led to successful matches. To do transformation some such structure is needed, to record variable bindings that are removed from a substitution when a quantified variable is bound to a value.

8 Conclusion

The *Coccinelle* system is a *well-established program transformer* currently being used by practitioners to automate and document collateral evolutions in Linux device drivers. We presented a compact, precise and self-contained semantics of Core-SmPL, in essence a rational reconstruction of the heart of the system. This gives it a solid foundation, one that motivates the structure of the Coccinelle framework, and justifies it theoretically.

Technically: we defined the semantics using continuation-passing style denotational semantics; made a prototype implementation in Haskell; translated SmPL to a novel implementation language (the temporal logic CTL); and formally proved the translation faithful to the denotational semantics. Partial evaluation's "staging" concept was used to define CTL-V, a CTL extension with existentially quantified variables that range over program points and source code parameters. This led to a more complex but practically more expressive and useful version of Core-SmPL. In the full paper [9] a model checking algorithm for CTL-V is outlined and exemplified on a string matching problem.

These results show a pleasing relation between theory and practice, and give descriptions of a complex working practical system. The descriptions are compact and (we hope) comprehensible to outsiders without previous experience with Coccinelle. Ideally, the insights gained here will be of benefit and perhaps even a guide to others with similar goals.

Acknowledgements. The authors wish to thank all the people involved with the *Coccinelle* project: Gilles Muller, Yoann Padioleau, Jesper Andersen, Henrik Stuart and, especially, Julia L. Lawall.

References

1. Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.16. Components for transformation systems. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*, Charleston, South Carolina, January 2006. ACM SIGPLAN.
2. Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 309–332, 2005.
3. Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Than. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13:3–30, 2001.

4. Oege De Moor, David Lacey, and Eric Van Wyk. Universal regular path queries. *Higher-order and Symbolic Computation*, 16(1-2):15–35, 2003.
5. Danny Dig and Ralph Johnson. How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.
6. Marc Fiuczynski, Robert Grimm, Yvonne Coady, and David Walker. Patch (1) considered harmful. In *Workshop on Hot Topics in Operating Systems*, 2005.
7. Marc E. Fiuczynski. Better tools for kernel evolution, please! *;LOGIN:*, 30(5):8–10, October 2006.
8. M.W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *International Conference on Software Management (ICSM)*, pages 131–142, 2000.
9. René Rydhof Hansen and Neil D. Jones. The semantics of semantic patches in Coccinelle: Program transformation for the working programmer (full paper). Project home page: www.emn.fr/x-info/coccinelle/.
10. Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.
11. 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.
12. David Lacey. *Program Transformation using Temporal Logic Specifications*. PhD thesis, Oxford University Computing Laboratory, 2003.
13. David Lacey and Oege de Moor. Imperative Program Transformation by Rewriting. In *Proc. International Conference on Compiler Construction, CC'01*, volume 2027 of *Lecture Notes in Computer Science*, pages 52–68. Springer Verlag, 2001.
14. David Lacey, Neil D. Jones, Eric Van Wyk, and Carl C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc. of Principles of Programming Languages, POPL'02*, volume 29, pages 283–294, 2002.
15. Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 220–231. ACM Press, 2003.
16. Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 219–230, 2004.
17. Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. In *PLOS '06: Proc. of workshop on Programming languages and operating systems*, page 10, 2006.
18. Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 59–71, 2006.
19. Bernhard Steffen. Optimal run time optimization proved by a new look at abstract interpretation. In *TAPSOFT, Vol.1*, pages 52–68, 1987.
20. Henrik Stuart, René Rydhof Hansen, Julia L. Lawall, Jesper Andersen, Yoann Padioleau, and Gilles Muller. Towards easing the diagnosis of bugs in OS code. In *Workshop on Linguistic Support for Modern Operating Systems (PLOS'07)*, 2007. To appear.