Conservative Extension in Positive/Negative Conditional Term Rewriting with Applications to Software Renovation Factories

Wan Fokkink*[1] and Chris Verhoef[2]

[1] University of Wales Swansea, Department of Computer Science, Singleton Park, Swansea SA2 8PP, UK,
w.j.fokkink@swan.ac.uk, fax: +44 1792 295708
[2] University of Amsterdam, Department of Computer Science, Programming Research Group, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands,
x@wins.uva.nl, fax: +31 20 5257490

Abstract. We transpose a conservative extension theorem from structural operational semantics to conditional term rewriting. The result is useful for the development of software renovation factories, and for modular specification of abstract data types.

1 Introduction

There is a strong link between the worlds of conditional term rewriting [30, 8] and of structural operational semantics (SOS) [35]. In fact, from a conceptual level they can be seen as identical. In both fields, terms are built from a set of function symbols. The binary relations on terms, rewrite steps and transitions, both are defined inductively by means of proof rules, called conditional rewrite rules or transition rules, respectively. Those rules, together with the validity, or non-validity, of a number of relations between terms, may imply the validity of another relation between terms.

There is one small distinction between both worlds. For a conditional term rewriting system (CTRS), provability is closed under context, in other words, if $s \to t$ is provable, then $Con[s] \to Con[t]$ is provable for every context $Con[]$. The set of transitions provable from a Transition System Specification (TSS) does not have to satisfy this characteristic, so in general a TSS cannot be expressed as a CTRS. However, the reverse transposition is possible, that is, for each CTRS there is an equivalent TSS. This transformation is obtained by adding context rules for all function symbols.

This correspondence was noted, but not exploited, by Groote and Vaandrager [28, Example 3.5]. They refrain from transposing their congruence format from TSSs to CTRSs, because it would not serve any practical purpose. Namely, although term rewriting and SOS theory are rooted on the same basis, their

* Supported by a grant from The Nuffield Foundation

aims are fundamentally different. Term rewriting seeks for termination and confluence of reductions, while in SOS theory in general behaviour is infinite and non-confluent. Usually, TSSs need to define a congruence relation with respect to a certain semantics, i.e., if two terms s and t are semantically equivalent, then $Con[s]$ and $Con[t]$ are semantically equivalent for all contexts $Con[]$. Several formats for TSSs have been developed which guarantee that they define a congruence relation for bisimulation semantics [28, 9, 3, 44, 17]. Most CTRSs from the literature do not fit these formats.

When a TSS is extended with new transition rules, the question arises whether or not such an extension influences the transitions of terms in the original domain. Usually, it is desirable that an extension is conservative, in the sense that the transitions for an original term are the same both in the original and in the extended TSS. Several formats have been developed which imply that an extended TSS is conservative over the original TSS [28, 9, 43, 20, 14]. Groote and Vaandrager [28, Theorem 7.6] proposed the first syntactic restrictions for an original TSS and its extension. Bol and Groote [9] adapted this conservativity format to the setting with negative conditions. Verhoef [43] proposed more general syntactic criteria, which were later on extended to a setting with inequalities [14]. In [20], Verhoef's format was transposed to higher-order languages.

This SOS notion of conservative extension is also useful in the realm of conditional term rewriting. Namely, if a CTRS $R_0 \oplus R_1$ is both confluent and an operational conservative extension of the CTRS R_0, then this extension is conservative in the classic sense. That is, then the CTRSs $R_0 \oplus R_1$ and R_0 induce exactly the same initial model for original terms. In this paper we exploit the link between TSSs and CTRSs to transpose the conservative extension theorem from the world of SOS to CTRSs. The conservativity result formulates syntactic requirements on the form of conditional rewrite rules in CTRSs R_0 and R_1, to ensure that the rewrite relation induced by R_0 on original terms is not affected by rewrite rules in R_1. It requires that each conditional rewrite rule in R_0 is *deterministic* [21]. Furthermore, each rewrite rule in R_1 should contain a fresh function symbol in its left-hand side.

The current paper arose from the final section in [18], where a simplified version of the conservativity format is transposed to the setting of conditional term rewriting. Simplifications are that we only treat first-order terms and that we do not allow the possibility that the left-hand side of a rewrite rule in the extension is an original term. We refrain from transposing the conservativity format to CTRSs in full generality for the sake of presentation, and to leave space to indulge in relevant applications. We refer to [19] where the SOS result has been transposed to higher-order CTRSs.

The conservativity format is applicable in the field of abstract data types, where there is a long tradition in specifying by means of modules of CTRSs. In abstract data types, modular specification means conservative extension, in our terminology. Namely, original modules fix the semantics of original terms, which should not be changed thereafter; new modules give meaning to fresh terms, which did not have a semantics before; see [5]. Our result is also applicable in the area of automated software engineering. In this paper we give a formal definition of a software renovation factory that may consist of numerous CTRSs. In order to build such a factory those CTRSs are combined, and then our result comes into play: it gives sufficient conditions so that the functionality of each separate component is not influenced in the presence of other components. This enables reuse of components and a component-based development of such factories. Of course, it cannot be demanded of an operator in a software renovation factory that she remembers our conservativity result upon adding a module to another component. It is, however, possible to implement this check; an automated check on determinism in the SOS world has been incorporated in the tool LATOS [29]. To demonstrate the use of our result, we provide examples from the literature, concerning term rewriting, abstract data types, and software renovation factories.

We study positive/negative CTRSs [31], which may contain negative conditions of the form $s \neg Dt$ for relations D, to express that there does not exist a relation sDt. We give meaning to such negative conditions using three-valued stable models [37, 22] from logic programming. Van de Pol [36] used instances of this semantic notion to provide negative answers to three open questions in term rewriting with priorities [4].

2 A Conservative Extension Theorem

2.1 Conditional Rewrite Rules

Definition 1. *A (single-sorted) signature Σ consists of a countably infinite set $\mathcal{V}$ of variables, and a non-empty set of function symbols f with fixed arities.*

A function symbol of arity zero is called a *constant*.

Definition 2. *Let Σ be a signature. The collection of* (open) terms $s, t, \ldots$ *over Σ is defined as the least set satisfying:*

- *each variable from $\mathcal{V}$ is a term;*
- *if function symbol f has arity n, and $t_1, ..., t_n$ are terms over Σ, then $f(t_1, ..., t_n)$ is a term over Σ.*

A term is called *closed* if it does not contain any occurrences of variables.

We assume a signature Σ, and a set $\mathcal{D}$ of relation symbols. The symbols in $\mathcal{D}$ represent binary rewrite relations between closed terms over Σ. Following Kaplan [31] we study positive/negative CTRSs, which may contain negative conditions of the form $s \neg Dt$, meaning that the relation sDt is not valid.

Definition 3. *For closed terms s, t over Σ, and $D \in \mathcal{D}$, sDt is called a* positive rewrite step*, and $s \neg Dt$ is called a* negative rewrite step.

The standard rewrite relation is the one-step relation $\rightarrow$. But we will also encounter its transitive-reflexive closure $\twoheadrightarrow$, the join $\downarrow$, and the equality sign $=$.

Definition 4. *A* (positive/negative) conditional rewrite rule *is of the form* $p \Leftarrow C$*, where.*

- *the* conclusion *p is of the form sDt;*
- *C is a (possibly empty) set of* conditions *of the form sDt or* $s\neg Dt$*;*

with s and t open terms. A (positive/negative) conditional term rewriting system *(CTRS) is a set of positive/negative conditional rewrite rules.*

We extend the notion of a *deterministic* conditional rewrite rule [21] to the setting with negative conditions.

Definition 5. *For a conditional rewrite rule* $p \Leftarrow C$*, the deterministic variables in this rule are defined inductively as follows.*

– *All variables in the left-hand side of p are deterministic.*
– *If sDt is a positive condition in C, and all variables in s are deterministic, then all variables in t are also deterministic.*

A conditional rewrite rule is called deterministic *if all its variables are so.*

Definition 6. *A* proof *from a CTRS R for a closed rewrite rule* $p \Leftarrow C$ *(which contains only closed terms) consists of an upwardly branching tree in which all upward paths are finite, where the nodes of the tree are labelled by positive and negative rewrite steps, such that:*

– *the root has label p,*
– *if some node has label q, and K is the set of labels of nodes directly above this node, then*
  1. *either* $K = \emptyset$*, and* $q \in C$*,*
  2. *or* $q \Leftarrow K$ *is a closed substitution instance of a rewrite rule in R.*

2.2 Conservative Extension

We define a notion of (operational) conservative extension for CTRSs, which is related to an equivalence notion for TSSs in [24, 17]: two TSSs are equivalent if they prove exactly the same rewrite rules N/τ where N contains only negative transitions.

Definition 7. *Let* Σ_0 *and* Σ_1 *be signatures. Their sum (or union)* $\Sigma_0 \oplus \Sigma_1$ *is* well-defined *if each function symbol and each variable in* $\Sigma_0 \cap \Sigma_1$ *has the same functionality in both signatures.*

We assume two CTRSs R_0 and R_1 over $(\Sigma_0, \mathcal{D}_0)$ and $(\Sigma_1, \mathcal{D}_1)$ respectively, where $\Sigma_0 \oplus \Sigma_1$ is well-defined. Their sum (or union) is denoted by $R_0 \oplus R_1$.

Definition 8. $R_0 \oplus R_1$ *is a* conservative extension *of* R_0 *if for each closed rewrite rule* $p \Leftarrow C$ *with*

- *C contains only negative conditions;*

- *the left-hand side of p is a term over Σ_0;*
- *there exists a proof from $R_0 \oplus R_1$ for $p \Leftarrow C$;*

then there exists a proof from R_0 for $p \Leftarrow C$.

We give an example of an extension that is *not* conservative.

Example 1. Σ_0 consists of the constant a, and R_0 of the rewrite rule $a \to x \Leftarrow x \to x$. Furthermore, Σ_1 consists of the constant b, and R_1 of the rewrite rule $b \to b$. Clearly the rewrite step $a \to b$ is valid in $R_0 \oplus R_1$, but not in R_0. Since a is an original term, $R_0 \oplus R_1$ is not a conservative extension of R_0.

Note that the CTRS R_0 in Example 1 is not deterministic, because the variable x in the rewrite rule in R_0 is not deterministic (see Definition 5).

Theorem 1. *Assume two CTRSs R_0 and R_1 over $(\Sigma_0, \mathcal{D}_0)$ and $(\Sigma_1, \mathcal{D}_1)$ respectively, where $\Sigma_0 \oplus \Sigma_1$ is well-defined. Under the following conditions, $R_0 \oplus R_1$ is a conservative extension of R_0.*

1. *R_0 is deterministic.*
2. *For each rewrite rule in R_1, the left-hand side of its conclusion contains a function symbol from $\Sigma_1 \backslash \Sigma_0$.*

Proof. This result follows almost directly from a similar result for TSSs; see [20, Theo. 3.20]. We only need to resolve the distinction in the notion of provability for TSSs and CTRSs. For this reason we introduce for each function symbol f of arity n, and for each argument $i \in \{1, ..., n\}$ of f, a so-called context rule

$$f(x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n) \to f(x_1, ..., x_{i-1}, y, x_{i+1}, ..., x_n) \;\Leftarrow\; x_i \to y.$$

We make two observations.

1. For each original function symbol f, the context rules for its arguments are all deterministic. Namely, since $x_1, ..., x_n$ occur in the left-hand side of the conclusion it is deterministic. Moreover, since the variable x_i is deterministic, the condition $x_i \to y$ makes that y is also deterministic.
2. For each fresh function symbol f, the context rules for its arguments all contain the fresh function symbol f in their source.

So if we add the context rules for original function symbols to R_0, and the context rules for fresh function symbols to R_1, then R_0 and R_1 still comply with the syntactic requirements that were formulated in the theorem.

Owing to the extra context rules, we can use the provability notion from the SOS world, where closure under context is not taken for granted. hence, we can apply the conservative extension result for TSSs [20, Theo. 3.20] to conclude that $R_0 \oplus R_1$ is a conservative extension of R_0.

2.3 Three-Valued Stable Models

When there are negative conditions around, it is no longer straightforward to define a sensible rewrite relation. We consider some well-established notions from logic programming [37, 22]. See [24] for a thorough overview of possibilities to give meaning to negative conditions.

The notion of a three-valued stable model was introduced by Przymusinski [37] in logic programming. It consists of two disjoint collections of positive rewrite steps: intuitively, $\mathtt{T}$ contains the *true* rewrite steps, while $\mathtt{U}$ contains the rewrite steps of which it is *unknown* whether or not they are true. All positive rewrite steps outside $\mathtt{T} \cup \mathtt{U}$ are considered to be *false*. These intuitions are made precise in the definition of a three-valued stable model.

Definition 9. *The collections* $(\mathtt{T}, \mathtt{U})$ *of positive rewrite steps are a* three-valued stable model *for a CTRS R, if the following two requirements hold.*

1. *The elements of* $\mathtt{T}$ *are exactly those positive rewrite steps sDs' for which there exists a closed rewrite rule $tDt' \Leftarrow C$ such that:*
   - *$s = Con[t]$ and $s' = Con[t']$ for some context $Con[]$,*
   - *there exists a proof from R for $tDt' \Leftarrow C$,*
   - *C contains only negative rewrite steps,*
   - *for each $s\neg Dt \in C$ we have $sDt \notin \mathtt{T} \cup \mathtt{U}$.*
2. *The elements of* $\mathtt{T} \cup \mathtt{U}$ *are exactly those positive rewrite steps sDs' for which there exists a closed rewrite rule $tDt' \Leftarrow C$ such that:*
   - *$s = Con[t]$ and $s' = Con[t']$ for some context $Con[]$,*
   - *there exists a proof from R for $tDt' \Leftarrow C$,*
   - *C contains only negative rewrite steps,*
   - *for each $s\neg Dt \in C$ we have $sDt \notin \mathtt{T}$.*

Example 2. The CTRS that consists of the rewrite rules $a \to b \Leftarrow a \not\to c$ and $a \to c \Leftarrow a \not\to b$ allows several three-valued stable models: $(\{a \to b\}, \emptyset)$ and $(\{a \to c\}, \emptyset)$ and $(\emptyset, \{a \to b, a \to c\})$.

If $R_0 \oplus R_1$ is a conservative extension of R_0, then each three-valued stable model for R_0 can be obtained by restricting a three-valued stable model for $R_0 \oplus R_1$ to the positive rewrite steps that have a closed original term as left-hand side. This theorem follows immediately from similar results for TSSs, Theorems 3.24 and 3.25 in [20], by the introduction of context rules.

Theorem 2. *Let R_0 be a CTRS over Σ_0. For three-valued stable models* $(\mathtt{T}, \mathtt{U})$ *for $R_0 \oplus R_1$, we define*

$$\begin{aligned} \mathtt{T}|\Sigma_0 &= \{sDt \in \mathtt{T} \mid s \text{ a closed term over } \Sigma_0\} \\ \mathtt{U}|\Sigma_0 &= \{sDt \in \mathtt{U} \mid s \text{ a closed term over } \Sigma_0\}. \end{aligned}$$

If $R_0 \oplus R_1$ is a conservative extension of R_0, then:

1. *if* $(\mathtt{T}, \mathtt{U})$ *is a three-valued stable model for $R_0 \oplus R_1$, then* $(\mathtt{T}|\Sigma_0, \mathtt{U}|\Sigma_0)$ *is a three-valued stable model for R_0;*

2. *each three-valued stable model for R_0 is of the form* $(\mathtt{T}|\Sigma_0, \mathtt{U}|\Sigma_0)$*, with* $(\mathtt{T}, \mathtt{U})$ *a three-valued stable model for* $R_0 \oplus R_1$*.*

According to Przymusinski [37], each CTRS allows a unique three-valued stable model $(\mathtt{T}, \mathtt{U})$ for which the set $\mathtt{U}$ of unknown positive rewrite steps is maximal. Furthermore, Przymusinski showed that this model coincides with the *well-founded* model of Van Gelder, Ross and Schlipf [22]. The next corollary follows from Theorem 2.

Corollary 1. *Let R_0 be a CTRS over Σ_0. If $R_0 \oplus R_1$ is a conservative extension of R_0, and $(\mathtt{T}, \mathtt{U})$ is the well-founded model for $R_0 \oplus R_1$, then $(\mathtt{T}|\Sigma_0, \mathtt{U}|\Sigma_0)$ is the well-founded model for R_0.*

Two other semantic notions are related to three-valued stable models:

- A *stable model* [23] is a three-valued stable model $(\mathtt{T}, \emptyset)$.
- A CTRS is *complete* [24] if its well-founded model is a stable model.

Suppose that the CTRS $R_0 \oplus R_1$ is complete; i.e., it has a well-founded model of the form $(\mathtt{T}, \emptyset)$. Furthermore, let $R_0 \oplus R_1$ be a conservative extension of R_0; item 1 in Theorem 2 implies that $(\mathtt{T}|\Sigma_0, \emptyset)$ is the well-founded model for R_0. If $R_0 \oplus R_1$ is confluent, then it follows that $R_0 \oplus R_1$ is conservative over R_0 in the classic sense from logic. That is, if the rewrite rules in $R_0 \oplus R_1$ and R_0 are taken to be equations, then both systems induce the same equations between original terms.

Remark 1. A positive CTRS induces a unique initial model of rewrite steps, which together constitute a minimal model for the CTRS. A rewrite step is in the initial model of a CTRS if and only if there exists a constructive proof for it. This is also the case for the rewrite steps in a three-valued stable model.

For a positive/negative CTRS, a *quasi-initial model* [31] of equations (instead of rewrite steps) is also required to constitute a minimal model for the CTRS. A quasi-initial model is not necessarily unique: the positive/negative CTRS R_0 that consists of the single rule $a \to c \Leftarrow a \neq b$ allows two quasi-initial models, $\{a = c\}$ and $\{a = b\}$. There does not exist a constructive proof for $a = b$; this contrasts with the semantics for positive CTRSs.

Theorem 2 does not hold if we replace "three-valued stable" by "quasi-initial". A counter-example is the extension of R_0 with the constant d and the CTRS R_1 that consists of the rewrite rules $d \to b$ and $d \to c$; this yields a unique quasi-initial model $\{a = b = c = d\}$. However, according to Theorem 1, $R_0 \oplus R_1$ is a conservative extension of R_0. The CTRS R_0 allows a unique three-valued stable model $(\{a \to c\}, \emptyset)$, and $R_0 \oplus R_1$ allows a unique three-valued stable model $(\{a \to c, d \to b, d \to c\}, \emptyset)$.

3 Application to CTRSs of Type III

A *CTRS of type III* consists of conditional rewrite rules of the form

$$s \to t \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n.$$

$\twoheadrightarrow$ denotes the transitive-reflexive closure of the one-step relation $\rightarrow$. It is defined by two rewrite rules, which we add explicitly to each CTRS of type III:

$$x \twoheadrightarrow x \qquad \text{and} \qquad x \twoheadrightarrow z \Leftarrow x \rightarrow y,\ y \twoheadrightarrow z.$$

The first rule is clearly deterministic. In the second rule, x occurs in the left-hand side of the conclusion, so it is deterministic. Then y is deterministic by condition $x \rightarrow y$, so z is deterministic by condition $y \twoheadrightarrow z$.

On a rewrite rule $p \Leftarrow C$ we often see the following three requirements.

A. The left-hand side of p is not a single variable.
B. Variables in the right-hand side of p also occur in the left-hand side of p.
C. Variables in C also occur in p.

Criteria A and B are natural in the unconditional case, because then they are essential in order to obtain termination. According to Middeldorp [33, page 114], criterion C is often imposed "due to severe technical complications". We left out these criteria, because our results do not require to impose them. Criteria A and B would even be a hindrance, because the two rewrite rules that define the relation $\twoheadrightarrow$ do not satisfy these criteria.

We give an example of an extension of a CTRS of type III, taken from [15], to demonstrate the use of our conservativity result.

Example 3. The CTRS N_0 implements addition on natural numbers. It assumes the constant 0, the unary successor function S, and the binary addition function A. Its standard rules are

$$\begin{aligned} A(0,x) &\rightarrow x \\ A(S(x),y) &\rightarrow S(A(x,y)) \end{aligned}$$

The two rules in the CTRS N_0 are clearly deterministic, because they do not have conditions, and they satisfy criterion B.

The CTRS N_1 implements the Fibonacci numbers. It assumes 0 and S and A, together with the unary Fibonacci function Fib. The rules of N_1 are

$$\begin{aligned} \mathit{Fib}(0) &\rightarrow (0, S(0)) \\ \mathit{Fib}(S(x)) &\rightarrow (z, A(y,z)) \Leftarrow \mathit{Fib}(x) \twoheadrightarrow (y,z) \end{aligned}$$

The second rule in N_1 is considered difficult, because it does not satisfy criterion B: the variables y and z do not occur in the left-hand side of its conclusion. Nevertheless, since the left-hand sides of the conclusions of the two rules in N_1 contain the fresh function symbol Fib, and since N_0 is deterministic, Theorem 1 yields that $N_0 \oplus N_1$ is a conservative extension of N_0. The first rule in N_1 is clearly deterministic. In the second rule, x occurs in the left-hand side of the conclusion of the rule, so it is deterministic. Then condition $\mathit{Fib}(x) \twoheadrightarrow (y,z)$ makes that y and z are deterministic. So the second rule in N_1 is also deterministic.

We extend $N_0 \oplus N_1$ with the following two standard module N_2 for a binary equality function eq, which decides whether or not two natural numbers are syntactically equal.

$$\begin{array}{l} eq(x,x) \rightarrow true \\ eq(x,y) \rightarrow false \;\Leftarrow\; eq(x,y) \not\twoheadrightarrow true \end{array}$$

Since the left-hand side of the conclusions of both rules contain the fresh function symbol *eq*, and since $N_0 \oplus N_1$ is deterministic, it follows from Theorem 1 that $N_0 \oplus N_1 \oplus N_2$ is conservative over $N_0 \oplus N_1$.

In a *CTRSs of type III$_n$* (also called 'normal'), conditions are conjuncts of expressions $s \twoheadrightarrow t$ where t is a closed normal form. In particular, terms at the right-hand sides of conditions are closed, so it follows that the only deterministic variables in a type III_n rule are the ones that occur in the left-hand side of its conclusion. Hence, a rule of type III_n is deterministic if all its variables occur in the left-hand side of its conclusion, that is, if it satisfies criteria B and C.

In a *CTRSs of type II* (also called 'join'), conditions are conjuncts of expressions $s \downarrow t$, which denote that s and t reduce to the same term. This can be formulated in type III style: $s \twoheadrightarrow y$ and $t \twoheadrightarrow y$, where y is a fresh variable. Since the y is fresh, again the only deterministic variables in a type II rule are the variables that occur in the left-hand side of its conclusion. Hence, a rule of type II is deterministic if it satisfies criteria B and C.

Finally, in a *CTRSs of type I* (also called 'semi-equational'), conditions are conjuncts of expressions $s = t$, which denote that s rewrites to t if the rewrite rules may be applied both from left to right and from right to left. The following example shows that the syntactic criteria from Theorem 1 are not sufficient to ensure that an extension of a CTRS of type I is conservative.

Example 4. Let $\Sigma_0 = \{a, b\}$ and $\Sigma_1 = \{a, b, c\}$, where a, b, c are constants. Let R_0 consist of the single rule $a \rightarrow b \Leftarrow a = b$. Furthermore, let R_1 consist of the two rules $c \rightarrow a$ and $c \rightarrow b$.

R_0 is deterministic, and even satisfies criteria A, B, and C. Also, the left-hand side of the conclusions of the rules in R_1 contain the fresh function symbol c. However, $a \rightarrow b$ is provable from $R_0 \oplus R_1$, but not from R_0. Since a is an original term, $R_0 \oplus R_1$ is not a conservative extension of R_0.

4 Application to Software Renovation Factories

One way of looking at renovating a software system is to consider it as an annotated abstract syntax tree (AST) that needs to be manipulated. This manipulation can be rewriting. This idea underlies the following definition of a software renovation factory (this definition is implicitly assumed in [11] where it is shown how to generate useful rewrite systems from a context-free grammar).

Definition 10. *A* software renovation factory *is a set of software renovation assembly lines. A* software renovation assembly line *is an ordered set of (renovation) components. A* (renovation) component *is a positive/negative CTRS.*

We explain what our theorem and software renovation factories have in common (the reader is referred to [10, 16] for a quick introduction to the field of reverse engineering and system renovation). Code that needs to be renovated is first parsed, resulting in an abstract syntax tree (AST). Renovation of code amounts to conditionally rewriting the AST to a desired normal form. Then the AST is unparsed, resulting in renovated code. To renovate code, it is customary to combine existing renovation components. This can be done sequentially in an assembly line by applying components in a fixed order, or simultaneously by taking the sum of components, or as a combination of these two. Our theorem is important for the simultaneous combination of components, which amounts to taking the sum of positive/negative CTRSs. The question that arises is whether the sum is conservative over the separate components, i.e., is the functionality of an extended component the same as before?

We give an example, and apply our theorem to it, to ensure that the combination of components does not influence the behaviour of the separate components. The example uses COBOL (Common Business Oriented Language) [1]. It focuses on a many-sorted TRS; the conservative extension theorem in this paper generalizes to a many-sorted setting without any complications; see [18, 20].

In the example below we follow [39] in departing from the standard prefix notation for terms (see Definition 2). For example, a function symbol with three arguments can be defined as `IF Boolean THEN Statement ELSE Statement END-IF -> Statement`; this notations resembles Backus Naur Forms [2]. The name of the function symbol `IF _ THEN _ ELSE _ END-IF` is interspersed with its domain that would be `Boolean` $\times$ `Statement` $\times$ `Statement` in the standard notation. The form that we used here is known as distributed fix operators, distfix operators, or mixfix operators; these names are due to Mosses and Goguen [25]. The terms over a signature of distfix operators are constructed as usual. In [32, p. 202] an elegant correspondence between many-sorted terms and Backus Naur Forms is made, illustrating the natural connection of universal algebra and formal language definitions. In [32, p. 210] the syntax of while programs is discussed where the connection between distributed fix operations and terms in prefix notation is elegantly illustrated.

Example 5. Suppose that a company moves to Japan and that they wish to migrate their mission-critical business software from MicroFocus COBOL to Fujitsu COBOL, since the local programmers are familiar with the latter dialect. One of the components that we introduce migrates MicroFocus specific `78` level constant definitions to the `SYMBOLIC CONSTANT` clause (a Fujitsu COBOL specific feature) of the `SPECIAL-NAMES` paragraph. In fact, in the first dialect, the declaration of constants is done in a certain subtree of the AST, and we need to move this information from this subtree to another subtree (the word 'move' is loosely phrased since we also have to modify the syntax of the declarations). An extra problem with the other subtree is that we may need to create it, since it may not yet be present in the original code. The move is implemented in the CTRS R_0 below. It contains five hand crafted rewrite rules that represent the above requirements, plus hundreds of rewrite rules that take care of traversal of the AST, which are generated automatically from the grammar. For detailed information on this generative technology we refer to [11].

Before explaining the rewrite rules, first we focus on notations. The function symbols `f_1`-`f_4` are generated automatically from the COBOL grammar (we renamed them for explanatory reasons). All other expressions that contain numerals are variables; the remaining expressions are terminals (or constant symbols in CTRS terminology). For example, `Ident-div1` is a variable that matches a complete `IDENTIFICATION DIVISION` of a COBOL program; `COMMENT2*` stands for zero or more COBOL comments; `Special-name1+` stands for one or more symbolic constants in a `SPECIAL-NAMES` section; and `VALUE` is a terminal representing the COBOL keyword `VALUE`. The five rewrite rules are:

```
[1] f_1(
    COMMENT1*
    Ident-div1
    Env-div1
    DATA DIVISION. COMMENT2*
      File-sec1
      WORKING-STORAGE SECTION. COMMENT3*
      Data-desc1*
      78 Id1 Dd-item1* VALUE Id2 Dd-item2*. COMMENT4*
        Dd-body1*
      Data-desc2*
      Link-sec1
    Proc-div1
    )^{ } =
    COMMENT1*
    Ident-div1
    f_2(Env-div1)^{ Id1 Id2 }
    DATA DIVISION. COMMENT2*
      File-sec1
      WORKING-STORAGE SECTION. COMMENT3*
      Data-desc1*
      Data-desc2*
      Link-sec1
    Proc-div1
[2] f_2( )^{ Id1 Id2 } = ENVIRONMENT DIVISION.  f_3( )^{ Id1 Id2 }
[3] f_3( )^{ Id1 Id2 } = CONFIGURATION SECTION.  f_4( )^{ Id1 Id2 }
[4] f_4( )^{ Id1 Id2 } = SPECIAL-NAMES. Id1 IS Id2.
[5] f_4(Special-name1+)^{ Id1 Id2 } = Special-name1+ Id1 IS Id2
```

First notice that R_0 is unconditional: for explanatory reasons we did not impose any conditions on the term rewriting rules of this example. From a software renovation point of view this is unrealistic. However, since we wish to illustrate our conservativity result rather than develop a software renovation factory in this paper, we keep the renovation components as simple as possible.

We explain the above rewrite rules in detail. Function symbol `f_1` takes as input a MicroFocus COBOL program and has as output the desired Fujitsu

COBOL program; see the end of this section for a typical example of an original and a rewritten COBOL program. Rule [1] defines f_1. The argument of f_1 is a textual representation of an AST containing a pattern that matches a complete COBOL program. It can start with comments, then an IDENTIFICATION DIVISION (matched by Ident-div1), an ENVIRONMENT DIVISION (matched by Env-div1), a DATA DIVISION (specified in such detail that it matches the 78 level constant definitions that we wish to move to another subtree), and a PROCEDURE DEFINITION (matched by the variable Proc-div1). The output of f_1 shows that the first and last divisions are not modified: the parts COMMENT1* Ident-div1 and Proc-div1 are invariant. In the DATA DIVISION the 78 level constants are removed. The essential information, residing in variables Id1 and Id2, is stored in memory. This memory is simply a second argument of the function symbols f_1 – f_4, which we call an attribute, denoted using curly braces. The function f_2 that appears in the output of f_1 takes care of addition of the constants in the Fujitsu dialect. This is implemented in the rules [2]–[5]. If there is no ENVIRONMENT DIVISION in the COBOL program, then the variable Env-div1 matches an empty subtree, so that rule [2] creates a subtree with top-node ENVIRONMENT DIVISION and an empty subtree, which is handled by f_3. Rule [2] passes on the attributes to f_3. If the ENVIRONMENT DIVISION already exists, then rule [2] does not apply; in this case one of the generated rules for f_2 renames f_2 into f_3, and passes on the attributes to f_3 as well. We emphasize that whatever the initial situation was, we always end up in the situation that the next function is f_3. Rule [3] is similar to rule [2]; it creates a CONFIGURATION SECTION if it is not present. If the CONFIGURATION SECTION already exists, then rule [3] does not apply; in this case one of the generated rules for f_3 renames f_3 into f_4, and passes on the attributes to f_4 as well. Rule [4] is also similar to [2] and [3] in that it creates the SPECIAL-NAMES paragraph if it is not present. It also adds the Fujitsu specific SYMBOLIC CONSTANT clause and uses the removed variables that reside in the attributes. In the case that there was already such a paragraph, rule [5] matches those in the variable Special-name1+, copies them to the output, and adds the SYMBOLIC CONSTANT clause.

Since COBOL was intended to look like written English, sentences ending with separator periods were introduced. Such a separator period terminates the scope of *all* still open IF statements. Later on, in 1985, an explicit END-IF for IF statements in COBOL was introduced. Suppose that MicroFocus COBOL code is written by a programmer before 1985, and that we want to change the implicit separator periods in such programs into END-IF statements. This can be implemented by means of a CTRS R_1 with four handwritten rewrite rules (plus hundreds of generated ones). The first three rewrite rules handle the three possible ways to implicitly terminate a COBOL IF phrase (see [1]), respectively:

- by an END-IF phrase at the same level of nesting;
- by a separator period;
- if nested, by an ELSE phrase associated with an IF statement at a higher level of nesting.

An additional fourth rewrite rule removes separator periods. See [11, p. 150] for an elaborate discussion, explanation and implementation of the four rewrite rules constituting R_1, and [12] for more background on COBOL grammars.

```
[1] g_1(Bad-cond)^{Attr*} = g_4(Bad-cond)^{Attr*} END-IF
[2] g_2(IF L-exp Sent)^{Attr*} = IF L-exp g_3(g_2(Sent)^{Attr*}) END-IF.
[3] g_2(IF L-exp Cond-body ELSE Sent)^{Attr*} =
      IF L-exp g_1(Cond-body)^{Attr*}
      ELSE g_3(g_2(Sent)^{Attr*}) END-IF.
[4] g_3(Stat.) = Stat
```

Both R_0 and R_1 serve the purpose of uniformizing the code. It is useful to uniformize code before restructuring, since it decreases the number of possibilities in rewriting the AST in a later phase. For performance reasons we combine both uniformizing components R_0 and R_1. The question arises whether we can do this safely. This is the case indeed, since R_0 is deterministic, and each rewrite rule in R_1 contains a fresh function symbol from `g_1`-`g_3` at the left-hand side of its conclusion. Such uniformization techniques are common practice in software renovation factories; see [13, 38] for a factory approach where an elimination assembly line for an important class of legacy systems is implemented.

Below we provide an original COBOL program and its rewritten code, which both print the word `HAIKU`. We explain the code fragments, and show where the rewrite rules changed the original code at the left-hand side. The `78` level constant `CON` and its value `1` are moved from the `DATA DIVISION` to the `ENVIRONMENT DIVISION`. They appear in the `SYMBOLIC CONSTANT` clause of the paragraph called `SPECIAL-NAMES`. Indeed, the appropriate paragraph, section and division have been created. The syntax of constants in MicroFocus COBOL and Fujitsu COBOL differs; of course, the rewrite rules take care of that. The `IF` is terminated by the separator period after the first print statement `DISPLAY 'HAI'`. The rewrite system adds an explicit scope terminator `END-IF` in the rewritten code.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HAIKU.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 VAR-1.
  02 SUB-1 PIC X COMP-X.
78 CON VALUE 1.
  02 SUB-2 PIC X COMP-X VALUE CON.
PROCEDURE DIVISION.
  IF SUB-2 = 1
    DISPLAY 'HAI'.
  DISPLAY  'KU'.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HAIKU.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
  SYMBOLIC CONSTANT
    CON IS 1.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 VAR-1.
  02 SUB-1 PIC X COMP-X.
  02 SUB-2 PIC X COMP-X VALUE CON.
PROCEDURE DIVISION.
  IF SUB-2 = 1
    DISPLAY 'HAI'
  END-IF.
  DISPLAY  'KU'.
```

This transformation can be obtained automatically, using the implementation of the CTRSs R_0 and R_1. The CTRS R_1 has been implemented in [11]; the CTRS R_0 has been defined for the sake of this example, and has been implemented using the same technology.

Our theorem has also been applied in [38], where incrementally an algorithm was developed for eliminating very difficult `GO TO` statements from COBOL/CICS programs from a Swiss Bank. The use of the theorem was that already developed patterns for eliminating `GO TO`s could safely be extended with new patterns without distroying the original functionality. This important consequence of our theorem gives therefore rise to incremental development of software renovation factories. This is important since then we can heavily reuse already developed components which is cost-effective.

5 Related Work

The conservativity format for structural operational semantics has a direct application to term rewriting, as was noticed in [43]. It can help, for example, to obtain a simple completeness proof for the process algebra ACP [7]. In that paper, completeness of the equations for ACP is derived by means of a term rewriting analysis. The confluence proof of the TRS consists of about 400 cases. Completeness of the equations for ACP could also be obtained by the combination of a much simpler completeness result, a conservative extension result for the operational semantics, and an elimination result; see [43].

In general, studies on modular properties of term rewriting systems deal with the following question: given two (mostly unconditional) TRSs with a certain desirable property, such as confluence or termination, does the combination of these TRSs also satisfy this property? It is often assumed that the signatures of the two rewrite systems are disjoint, and that the variables in a rewrite rule all occur in its left-hand side. CTRSs that satisfy these requirements are automatically within our conservativity format. In this paper it is investigated whether the *full* rewriting relation is preserved for terms over only *one* of the signatures. The signatures of the original CTRS and its extension need not be disjoint. Toyama [40] showed that confluence is a modular property for TRSs [40], but that in general termination is not [41]. Klop and Barendregt gave a counter-example which shows that completeness is not modular, but Toyama, Klop and Barendregt [42] proved that completeness in combination with left-linearity is modular for TRSs. Ohlebusch [34] showed that if a combination of two TRSs does not terminate, then one of the TRSs is not $\mathcal{C}_{\mathcal{E}}$-terminating, while the other TRS is collapsing. (This generalizes a similar result for finitely branching TRSs of Gramlich [27]). Middeldorp [33] presented a panorama of positive and negative results on modular properties of CTRSs. For example, he showed that confluence constitutes a modular property for CTRSs. Gramlich [26] showed that his main results in [27] extend to CTRSs.

References

1. ANSI X3.23–1985. Programming Language – COBOL. American National Standards Institute, Inc, 1985.
2. J.W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings ICIP*, pp. 125–131. Unesco, Paris, 1960.
3. J. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In *Proc. CONCUR'93*, LNCS 715, pp. 477–492. Springer, 1993.
4. J. Baeten, J. Bergstra, J.W. Klop, and P. Weijland. Term-rewriting systems with rule priorities. *Theoretical Computer Science*, 67(2/3):283–301, 1989.
5. J. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison Wesley, 1989.
6. J. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37(2):335–372, 1990.
7. J. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3), 1984.
8. J. Bergstra and J.W. Klop. Conditional rewrite rules: confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.
9. R. Bol and J.F. Groote. The meaning of negative premises in transition system specifications. *Journal of the ACM*, 43(5):863–914, 1996.
10. M. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation – an annotated bibliography. *ACM SEN*, 22(1):57–68, 1997.
11. M. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *Proc. 4th Working Conference on Reverse Engineering*, pp. 144–155, 1997.
12. M. van den Brand, A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In *Proc. 2nd Workshop on the Theory and Practice of Algebraic Specifications, 1997. Workshops in Computing*, Springer, Available at http://www.springer.co.uk/ewic/workshops/.
13. M. van den Brand, A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy systems. In *Proc. 2nd Euromicro Conference on Software Maintenance and Reengineering*, pp. 11–19. IEEE Computer Society Press, 1998.
14. P. D'Argenio and C. Verhoef. A general conservative extension theorem in process algebras with inequalities. *Theoretical Computer Science*, 177:351–380, 1997.
15. N. Dershowitz, M. Okada, and G. Shivkumar. Confluence of conditional rewrite systems. In *Proc. CTRS'87*, LNCS 308, pp. 31–44. Springer, 1987.
16. A. van Deursen, P. Klint, and C. Verhoef. Research issues in renovation of legacy software. In *Proceedings ETAPS'99*, 1999. To appear.
17. W. Fokkink and R. van Glabbeek. Ntyft/ntyxt rules reduce to ntree rules. *Information and Computation*, 126(1):1–10, 1996.
18. W. Fokkink and C. Verhoef. A conservative look at term deduction systems with variable binding. Report 95-28, Eindhoven University of Technology, 1995.
19. W. Fokkink and C. Verhoef. An SOS message: conservative extension for higher-order positive/negative conditional term rewriting. Report P9715, University of Amsterdam, 1997.
20. W. Fokkink and C. Verhoef. A conservative look at operational semantics with variable binding. *Information and Computation*, 146(1):24–54, 1998.

21. H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. In *Proc. CTRS'92*, LNCS 656, pp. 430–437. Springer, 1993.
22. A. van Gelder, K. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
23. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Logic Programming Conference*, pp. 1070–1080. MIT Press, 1988.
24. R. van Glabbeek. The meaning of negative premises in transition system specifications II. In *Proc. ICALP'96*, LNCS 1099, pp. 502–513. Springer, 1996.
25. J. Goguen. Personal Communication, January 1993.
26. B. Gramlich. Sufficient conditions for modular termination of conditional term rewriting systems. In *Proc. CTRS'93*, LNCS 656, pp. 128–142. Springer, 1993.
27. B. Gramlich. Generalized sufficient conditions for modular termination of rewriting. *Applicable Algebra in Engin., Communic. and Comput.*, 5:131–158, 1994.
28. J.F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, 1992.
29. P. Hartel. LATOS – a lightweight animation tool for operational semantics. Report DSSE-TR-97-1, University of Southampton, 1997.
30. S. Kaplan. Conditional rewrite rules. *TCS*, 33(2):175–193, 1984.
31. S. Kaplan. Positive/negative conditional rewriting. In *Proc. CTRS'87*, LNCS 308, pp. 129–143. Springer, 1987.
32. K. Meinke and J. Tucker. Universal algebra. In *Handbook of Logic for Computer Science*, Volume I, pp. 189–411. Oxford University Press, 1993.
33. A. Middeldorp. Modular properties of conditional term rewriting systems. *Information and Computation*, 104(1):110–158, 1993.
34. E. Ohlebusch. On the modularity of termination of term rewriting systems. *Theoretical Computer Science*, 136(2):333–360, 1994.
35. G. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University, 1981.
36. J. van de Pol. Operational semantics of rewriting with priorities. *Theoretical Computer Science*, 200(1/2):289–312, 1998.
37. T. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13(4):445–463, 1990.
38. A. Sellink, H. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. In *Proceedings 3rd European Conference on Maintenance and Reengineering*. IEEE Computer Society Press, 1999.
39. A. Sellink and C. Verhoef. Native patterns. In *Proc. 5th Working Conference on Reverse Engineering*, pp. 89–103. IEEE Computer Society Press, 1998.
40. Y. Toyama. On the Church-Rosser property for the direct sum of term rewriting systems. *Journal of the ACM*, 34(1):128–143, 1987.
41. Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, 1987.
42. Y. Toyama, J.W. Klop, and H. Barendregt. Termination for direct sums of left-linear complete term rewriting systems. *Journal of the ACM*, 34(1):128–143, 1987.
43. C. Verhoef. A general conservative extension theorem in process algebra. In *Proc. PROCOMET'94*, *IFIP Transactions* A-56, pp. 149–168. Elsevier, 1994.
44. C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nordic Journal of Computing*, 2(2):274–302, 1995.