Transformations for Grammar Adaptation*

Ralf Lämmel

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Email: `ralf@cwi.nl`
WWW: `http://www.cwi.nl/~ralf/`

Abstract. We employ grammar transformations for the adaptation of grammars. Starting from a few fundamental grammar transformations, and some supplementary notions like focus, constraint and yielder we derive an operator suite for grammar adaptation. Three major kinds of adaptations are separated, namely transformations for refactoring, construction and destruction. While refactoring is semantics-preserving in the narrow sense, transformations for construction and destruction require the consideration of relaxed notions of semantics preservation based on other grammar relations than equality of generated languages. The consideration of semantics and accompanying preservation properties is slightly complicated by the fact that we cannot insist on reduced and terminated grammars.

A derived transformational approach to grammar improvement has been successfully applied in grammar recovery projects. More generally, the grammar adaptations covered in the paper are also useful in grammar development including maintenance.

1 Introduction

We consider transformations facilitating the stepwise adaptation of grammars. These transformations model common schemes of adaptation such as refactoring, local changes like generalisation or removal of phrases. Indeed, the grammar transformations we focus on in this paper are meant to mimic adaptations which are performed by "grammar programmers" in grammar (re-) engineering manually otherwise. The amount of grammar programming should not be underestimated.

The transformations formalised in this paper proved to be invaluable in actual grammar recovery projects such as the recovery of the VS-COBOL-II grammar. Actually, this project led to the first publicly available, relatively correct and complete COBOL grammar [10]. The effort for this undertaking was relatively small compared to other known figures. The use of formal grammar transformations was the key to make this process accessible for tool support, traceable regarding the many steps and decisions involved, and measurable to allow for formal reasoning and claims like correctness and completeness. A global account on grammar recovery and grammar (re-) engineering from a software engineering perspective is given in [11].

* A full version of the paper appears as CWI technical report.

```
2.8.5 REDEFINES-clause

 ___ Format ______________________________________________________
|                                                                 |
| >>__level-number__________________REDEFINES__data-name-2____><  |
|                    |_data-name-1_|                              |
|                    |_FILLER______|                              |
|                                                                 |
|_________________________________________________________________|
```

Fig. 1. An incorrect syntax diagram from [9]

Let us present a concrete example. In recovering the VS-COBOL-II grammar [10] from the IBM references [9, 8], we were faced with a surprising number of errors and omissions.[1] It is actually the lack of (use of) formal methods that causes such important documents to be incorrect and incomplete. One incorrectness is shown in the syntax diagram in Figure 1. The diagram is supposed to define the syntax of a `REDEFINES`-clause which is an optional clause of a COBOL data item. For convenience, we also define the piece of syntax from the diagram in BNF notation:

```
REDEFINES-clause :=
  level-number (data-name | FILLER) REDEFINES data-name
```

The diagram or its integration in the overall grammar is incorrect because the diagram does not just define the structure of a `REDEFINES`-clause, but rather the structure of a data item *with* a `REDEFINES`-clause. This problem can be easily recognized by parsing approved code if the (prototype) parser is derived directly from the diagrams [11]. To correct the diagram or the grammar resp., we need to delete the phrase which is not concerned with the actual `REDEFINES`-clause. Using the operators for grammar adaptation from the paper, this removal is expressed as follows:

```
focus on REDEFINES-clause do
  delete level-number (data-name | FILLER)
```

The paper is structured as follows. In Section 2, we propose a variant of context-free grammars particularly useful to cope with incomplete, incorrect, evolving grammars. In Section 3, a framework for grammar transformations is developed. It offers transformation primitives and combinators. In Section 4, operators for grammar adaptations, e.g., the operator delete used in the example above, are derived in the framework.

Acknowledgement I am grateful to Jeremy Gibbons, Jörg Harm, Jan Heering, Merijn de Jonge, Paul Klint, Chris Verhoef, Joost Visser, and Dave Wile for encouraging discussions on the subject, and for helpful suggestions. Some results from this paper have been presented at the 54th IFIP WG2.1 ("Algorithmic Languages and Calculi") meeting in Blackheath, London, April, 3th–7th, 2000, and at the 2nd Workshop Software-Reengineering, Bad Honnef, May, 11th–12th, 2000.

[1] About 300 transformation steps were needed to derive [10] from [9, 8].

2 Grammar fragments

We derive a simple variant of context-free grammars as the object language for grammar transformations. The idea of grammar adaptation implies that we cannot assume reduced and terminated context-free grammars. We rather have to cope with incomplete and evolving grammars or grammar modules.

2.1 Standard context-free grammars

A context-free grammar G is usually considered as a quadruple $\langle N, T, s, P\rangle$, where N and T are the disjoint finite sets of nonterminals resp. terminals, $s \in N$ is called start symbol, and P is a finite set of productions or (context-free) rules with $P \subset N \times (N \cup T)^\star$. A production $\langle l, r\rangle \in P$ with $l \in N$ and $r \in (N \cup T)^\star$ is also written as $l \to r$. It is common to assume terminated and reduced context-free grammars, that is all nonterminals can be derived into terminal strings, and all nonterminals are reachable from the start symbol.

2.2 Deviation

We slightly deviate from the standard. We do not insist on the identification of a start symbol s because this is not sensible for incomplete grammars or grammar modules. Furthermore, we assume that nonterminals and terminals can be separated by the convention that they originate from different universes of grammar symbols, that is $\mathcal{U}_N$ for nonterminals, and $\mathcal{U}_T$ for terminals. By this, we do not need to make N and T explicit for a given set P of productions. Thus, from here on, a grammar is represented by its productions. We call this deviation *grammar fragments*. We do not require grammar fragments to be terminated, that is, there can be nonterminals which are used, but not defined. These nonterminals are usually called *bottom nonterminals*. The domains of all context-free productions $\mathcal{R}$ (or rules for short) and grammar fragments $\mathcal{G}$ are defined as follows:[2]

$$
\begin{aligned}
\mathcal{R} &= \mathcal{U}_N \times (\mathcal{U}_N \cup \mathcal{U}_T)^\star \\
\mathcal{G} &= \mathcal{P}_{\mathit{fin}}(\mathcal{R})
\end{aligned}
$$

We use γ and δ possibly indexed or primed to range over $\mathcal{G}$. The terms $\mathcal{T}(\gamma) \subset \mathcal{U}_T$ and $\mathcal{N}(\gamma) \subset \mathcal{U}_N$ are used to denote terminals and nonterminals of γ. These and other relevant sets of nonterminals are defined in Figure 2.

The bottom nonterminals $\perp(\gamma)$ capture some important properties of a grammar fragment γ. In grammar development, a bottom nonterminal might correspond to a nonterminal lacking a definition due to the incompleteness of the grammar. Alternatively, a bottom nonterminal might indicate a connectivity problem, that is its intended definition is given

[2] $\mathcal{P}_{\mathit{fin}}(x)$ denotes the finite subsets of x.

$$\begin{aligned}
\mathcal{D}(\gamma) &= \{n \in \mathcal{U}_N \mid n \to u \in \gamma, u \in (\mathcal{U}_N \cup \mathcal{U}_T)^\star\}\\
\mathcal{U}(\gamma) &= \{n' \in \mathcal{U}_N \mid n \to u\ n'\ v \in \gamma, n' \in \mathcal{U}_N, u, v \in (\mathcal{U}_N \cup \mathcal{U}_T)^\star\}\\
\mathcal{N}(\gamma) &= \mathcal{D}(\gamma) \cup \mathcal{U}(\gamma)\\
\mathcal{T}(\gamma) &= \{t \in \mathcal{U}_T \mid n \to u\ t\ v \in \gamma, t \in \mathcal{U}_T, u, v \in (\mathcal{U}_N \cup \mathcal{U}_T)^\star\}\\
\bot(\gamma) &= \mathcal{U}(\gamma) \setminus \mathcal{D}(\gamma)\\
\top(\gamma) &= \mathcal{D}(\gamma) \setminus \mathcal{U}(\gamma)\\
\emptyset(\gamma) &= \{n \in \mathcal{N}(\gamma) \mid \ \nexists w \in (\bot(\gamma) \cup \mathcal{T}(\gamma))^\star.\ n \Rightarrow_\gamma^* w\}
\end{aligned}$$

Fig. 2. defined nonterminals $\mathcal{D}(\gamma)$, used nonterminals $\mathcal{U}(\gamma)$, nonterminals $\mathcal{N}(\gamma)$, terminals $\mathcal{T}(\gamma)$, bottom nonterminals $\bot(\gamma)$, top nonterminals $\top(\gamma)$, looping nonterminals $\emptyset(\gamma)$

using a different nonterminal symbol. There is also the notion of *top nonterminals* which is somewhat dual to bottom nonterminals. As Figure 2 details, the set $\top(\gamma)$ of top nonterminals consists of all nonterminals defined but not used in γ. In a way, top nonterminals provide an indication to what extent a grammar fragment is reduced and connected. The set $\emptyset(\gamma)$ of looping nonterminals will be explained later.

2.3 Semantics

The common semantics for context-free grammars is based on derivation ($\Rightarrow$ etc.). The language $\mathcal{L}(G)$ generated by a common context-free grammar $G = \langle N, T, s, P\rangle$ is usually defined as follows:

$$\mathcal{L}(G) = \{w \in T^\star \mid s \Rightarrow_G^* w\},$$

that is as the set of terminal strings derivable from the dedicated start symbol. Of course, one can also consider the terminal strings derivable from an arbitrary nonterminal n. We are going to denote the semantics of n w.r.t. a grammar fragment γ as $[\![n]\!]_\gamma$. Adopting the idea of generated terminal strings, we get a first approximation of the ultimate denotation of n, namely:

$$[\![n]\!]_\gamma \supseteq \{w \in \mathcal{T}(\gamma)^\star \mid n \Rightarrow_\gamma^* w\}$$

It easy to acknowledge that terminal strings are not sufficient for grammar fragments since we do not require grammar fragments to be terminated. The set of terminal strings generated from a non-terminated nonterminal n is simply the empty set. In that sense, terminal strings can only provide a lower bound for the ultimate denotation of a nonterminal n. Consider, for example, the incomplete grammar γ_1 in Figure 3. Indeed, b and thereby s generate the empty set of terminal strings. Too little information is conveyed into the terminal strings. This problem can be solved by taking sentential forms into account as opposed to terminal strings. As we will see, sentential forms correspond to an upper bound for the ultimate denotation. Thus, we have the following:

$$[\![n]\!]_\gamma \subseteq \{w \in (\mathcal{N}(\gamma) \cup \mathcal{T}(\gamma))^\star \mid n \Rightarrow_\gamma^* w\}$$

$$
\begin{array}{lcl}
\gamma_1 & & \gamma_2 \\
\begin{array}{l} s \to a\ b \\ a \to \epsilon \\ a \to \text{“A”}\ a \end{array}
&
\begin{array}{rcl}
[\![b]\!]^{\emptyset}_{\gamma_1} = \emptyset & = & [\![b]\!]^{\emptyset}_{\gamma_2} = \emptyset \\
[\![b]\!]^{\perp(\gamma_1)}_{\gamma_1} = \{b\} & \supset & [\![b]\!]^{\perp(\gamma_2)}_{\gamma_2} = \emptyset \\
[\![b]\!]^{\perp(\gamma_1)\cup\emptyset(\gamma_1)}_{\gamma_1} = \{b\} & \subset & [\![b]\!]^{\perp(\gamma_2)\cup\emptyset(\gamma_2)}_{\gamma_2} = \{b, \text{”B”}\ b, \ldots\}
\end{array}
&
\begin{array}{l} s \to a\ b \\ a \to \epsilon \\ a \to \text{“A”}\ a \\ b \to \text{”B”}\ b \end{array}
\end{array}
$$

Fig. 3. Illustration of incomplete grammars

Sentential forms are too sensitive. They do not even provide a basis to state the semantics preservation of fold/unfold modulations. Such modulations are obviously very useful in refactoring a grammar before more involved adaptations are performed. We need to restrict the sentential forms that only particular nonterminals are *observable*. It is sensible to require at least the bottom nonterminals to be observable. Then we can still derive all “interesting” sentential forms for γ_1 in Figure 3. To be slightly more flexible regarding observable nonterminals, we consider an augmented variant of the semantic function with a superscript for the observable nonterminals:

$$[\![n]\!]^{\psi}_{\gamma} =^{\text{def}} \{w \in (\psi \cup \mathcal{T}(\gamma))^{\star} \mid n \Rightarrow^{*}_{\gamma} w\}, \text{where } \psi \subseteq \mathcal{N}(\gamma)$$

The semantics restricted to sentential forms consisting solely of bottom nonterminals and terminals is then denoted by $[\![n]\!]^{\perp(\gamma)}_{\gamma}$. It will turn out that this a better lower bound than the first one based on just terminal strings given above, but it is not yet quite right what we need. Thus, we have the following:

$$[\![n]\!]_{\gamma} \supseteq [\![n]\!]^{\perp(\gamma)}_{\gamma}$$

Refer back to Figure 2 for the simple definition of bottom nonterminals and all other sets of nonterminals relevant for grammar fragments. We can also read $[\![n]\!]^{\psi}_{\gamma}$ as the semantics of n according to γ assuming the nonterminals in ψ are observable. There is a problem with observing bottom nonterminals, only. Consider the grammars in Figure 3 for that purpose. γ_2 originated from γ_1 by just adding one rule defining b. Only observing bottom nonterminals, the extended grammar γ_2 generates the empty set from b, although the original grammar γ_1 does not. The empty set for γ_2 is caused by the particular definition of b in γ_2, that is the definition is non-terminating or looping because there is no base case for b. Thereby, b does not generate strings over $\perp(\gamma_2) \cup \mathcal{T}(\gamma_2)$. We resolve that problem by observing also the set $\emptyset(\gamma)$ for a grammar fragment, say the set of all looping nonterminals as defined in Figure 2. Finally, we are in the position to define the denotation of n w.r.t. γ in the ultimate way as follows:

$$[\![n]\!]_{\gamma} =^{\text{def}} [\![n]\!]^{\perp(\gamma)\cup\emptyset(\gamma)}_{\gamma}$$

This semantics assumes the minimum of observable nonterminals without causing the denotation of some nonterminal to be empty.

2.4 Reasoning

We are interested in comparing grammars, and in characterising the properties of grammar transformations. For this purpose, we define certain relations on $\mathcal{G}$ starting from the equivalence of grammar fragments. Finally, the grammar relations are employed for defining *relaxed* forms of semantics-preservation.

Definition 1 (Equivalence ($\equiv$)). *Given two grammars γ, γ', we say that $\gamma \equiv \gamma'$, or γ and γ'* are equivalent, *if for all $n \in \mathcal{N}(\gamma) = \mathcal{N}(\gamma')$ it holds that $[\![n]\!]_\gamma = [\![n]\!]_{\gamma'}$.*

This relation is useful to characterize fold/unfold modulations. All grammar relations will be exemplified in Figure 4. As for equivalence, it holds $\gamma_1 \equiv \gamma_2$. The way in which the other grammars γ_3–γ_6 differ from γ_1 goes beyond equivalence. Indeed, equivalence is often too restrictive, that is we need weaker relations on two grammars γ and γ':

- γ and γ' are only equivalent modulo renaming.
- $[\![n]\!]_\gamma$ cannot be related to $[\![n]\!]_{\gamma'}$ for some n.
- It does not hold $[\![n]\!]_\gamma = [\![n]\!]_{\gamma'}$, but $[\![n]\!]_\gamma \subseteq [\![n]\!]_{\gamma'}$ for some n.
- Some nonterminals from $\bot(\gamma)$ are defined in γ'.

These relaxations are formalised in the following definitions.

γ_1	γ_2	γ_3	γ_4	γ_5	γ_6
$x \to y$	$x \to z$	$x \to y$	$x \to z$	$x \to y$	$x \to y$
$y \to z$	$y \to z$	$y \to z'$		$y \to z$	$y \to z$
				$y \to z'$	$z \to \epsilon$

Fig. 4. Illustration of grammar relations

Definition 2 (Equivalence modulo renaming ρ ($\stackrel{\rho}{\equiv}$)). *We say that $\gamma \stackrel{\rho}{\equiv} \gamma'$, or γ and γ'* are equivalent modulo renaming ρ, *if ρ is a bijective function of type $\rho : \mathcal{N}(\gamma) \to \mathcal{N}(\gamma')$, and for all $n \in \mathcal{N}(\gamma)$ it holds that $\rho([\![n]\!]_\gamma) = [\![\rho(n)]\!]_{\gamma'}$.*

In Figure 4, it holds that $\gamma_1 \stackrel{id[z'/z]}{\equiv} \gamma_3$.[3] As for notation, in $\rho([\![n]\!]_\gamma)$, the renaming function ρ is used lifted to sets of strings over $\bot(\gamma) \cup \emptyset(\gamma) \cup \mathcal{T}(\gamma)$.

Definition 3 (Equivalence except π ($\stackrel{\pi}{\equiv}$)). *We say that $\gamma \stackrel{\pi}{\equiv} \gamma'$, or γ and γ'* are equivalent except $\pi \subseteq \mathcal{N}(\gamma) \cup \mathcal{N}(\gamma')$, *if for all $n \in (\mathcal{N}(\gamma) \setminus \pi) = (\mathcal{N}(\gamma') \setminus \pi)$ it holds that $[\![n]\!]_\gamma = [\![n]\!]_{\gamma'}$.*

In Figure 4, it holds that $\gamma_1 \stackrel{\{y\}}{\equiv} \gamma_4$. In the case of equivalence except π, we do not relate the denotations w.r.t. two grammars for the nonterminals in π at all. This is a sensible choice if π are auxiliary or superfluous nonterminals.

[3] *id* denotes the identity function.

Definition 4 (Embedding ($\overset{\leq}{\equiv}$)). *We say* $\gamma \overset{\leq}{\equiv} \gamma'$*, if for all* $n \in \mathcal{N}(\gamma) \subseteq \mathcal{N}(\gamma')$ *it holds that* $[\![n]\!]_\gamma \subseteq [\![n]\!]_{\gamma'}^{\emptyset(\gamma)\cup\perp(\gamma')\cup\perp(\gamma')}$*. We also say that* γ is embedded in γ'.

The relation holds for γ_1 and γ_5 in Figure 4. The above definition essentially says that γ is embedded in γ' if γ' generates more than γ, i.e., $[\![n]\!]_\gamma \subseteq [\![n]\!]_{\gamma'}$. However, the definition also observes the looping nonterminals from γ as far as the denotation of γ' is concerned. This is done to enforce a homogeneous regarding the extension of the definition of looping nonterminals. Note that we cannot say that γ is embedded in γ' if γ' defines bottom nonterminals of γ. On the other hand, embedding is not excluded if γ' defines some new nonterminals not at all present in γ.

Definition 5 (Prefix ($\overset{\Rightarrow^*}{\equiv}$)). *We say* $\gamma \overset{\Rightarrow^*}{\equiv} \gamma'$*, or* γ is a prefix of γ' *if for all* $n \in \mathcal{N}(\gamma) \subseteq \mathcal{N}(\gamma')$ *it holds that*

$$\forall x \in [\![n]\!]_\gamma .\ \exists y \in [\![n]\!]_{\gamma'} .\ x \Rightarrow^*_{\gamma'} y.$$

This means that strings from $[\![n]\!]_\gamma$ can be completed into strings from $[\![n]\!]_{\gamma'}$ using $\Rightarrow^*_{\gamma'}$. Note the different mode of extension compared to embeddings. Bottom nonterminals from γ can be defined in γ', that is they occur in the sentential forms in $[\![n]\!]_\gamma$, and so they will be expanded by $\Rightarrow^*_{\gamma'}$ to derive strings from $[\![n]\!]_{\gamma'}$. A syntactical operation which can be characterised by this semantic notion is context-free substitution. The relation holds for γ_1 and γ_6 in Figure 4.

The above grammar relations can be employed to define various preservation properties. We consider partial functions on grammars, say partial *grammar transformers*. The notation $Rel(f)$ is used to denote the relation given by the partial function $f : \tau \nrightarrow \tau$ for some type τ, that is:

$$Rel(f) = \{\langle x, y\rangle \in \tau \times \tau \mid y = f(x)\}$$

Definition 6 (preservation properties). *A function* $f : \mathcal{G} \nrightarrow \mathcal{G}$ *is*

- strictly semantics-preserving *if* $Rel(f) \subseteq^4 \equiv$*,*
- semantics-preserving modulo renaming ρ *if* $Rel(f) \subseteq \overset{\rho}{\equiv}$*,*
- semantics-preserving except nonterminals π *if* $Rel(f) \subseteq \overset{\pi}{\equiv}$*,*
- increasing *if* $Rel(f) \subseteq \overset{\leq}{\equiv}$*,*
- decreasing *if* $Rel(f) \subseteq \overset{\leq}{\equiv}^{-1}$*,*
- defining *if* $Rel(f) \subseteq \overset{\Rightarrow^*}{\equiv}$*,*
- rejecting *if* $Rel(f) \subseteq \overset{\Rightarrow^*}{\equiv}^{-1}$*.*

Note that for symmetric grammar relations, there is one corresponding preservation property, whereas for asymmetric grammar relations, there are two preservation properties.

The ultimate goal is to come up with transformation operators which separate these various roles in transformations, and to show that an operator suite structured in that way also embodies operators for grammar adaptation in an orthogonal and exhaustive way.

[4] Since all our grammar relations do actually not correspond to functions, only $Rel(f) \subset \ldots$ is feasible

3 The transformation framework

In this section, we define a simple framework for grammar transformations. The developed transformation primitives and combinators provide the basic operators to describe useful grammar adaptations. We give a denotational semantics of the operators using partial grammar transformers as denotations.

3.1 Primitives

All the grammar adaptations we have in mind are derivable from the syntactical domain Trafo_0 of primitives defined in Figure 5. There are the constant operators id the semantics of which is the identity function on grammars, fail modelling the undefined grammar transformation, and reset denoting the grammar transformer returning the empty set of productions. There are primitives to add and to subtract a rule from the given grammar. The operator substitute serves for the substitution of nonterminals in the productions. Finally the operator replace replaces phrases by other phrases.

$$
\begin{aligned}
\mathit{Trafo}_0 &::= \mathsf{id} \mid \mathsf{fail} \mid \mathsf{reset} \mid \mathsf{add}\ \mathit{Rule} \mid \mathsf{sub}\ \mathit{Rule} \\
&\quad\ \mid\ \mathsf{substitute}\ \mathit{Nt}\ \mathsf{by}\ \mathit{Nt} \mid \mathsf{replace}\ \mathit{Phrase}\ \mathsf{by}\ \mathit{Phrase} \\
\mathit{Rule} &:= \mathit{Nt} \rightarrow \mathit{Phrase} \\
\mathit{Nt} &:= \mathcal{U}_N \\
\mathit{Phrase} &:= (\mathcal{U}_N \cup \mathcal{U}_T)^\star
\end{aligned}
$$

Fig. 5. Syntax of the transformation primitives

In Figure 6, the simple interpretation of the primitives is given by recursive functions in the style of denotational semantics. For brevity, we omit the verbose definition of substitute and replace. The given semantics does not rely on *cpos* as semantic domains because general recursion is not involved. We use partial grammar transformers as denotations where the undefined value corresponds to failure. There are trivial semantic functions which are concerned with the interpretation of rules ($\mathcal{IR}$), nonterminals ($\mathcal{IN}$), and phrases ($\mathcal{IP}$). We call the elements of the corresponding syntactical domains *Rule*, *Nt* and *Phrase* *yielders* because the interpretation of these domains potentially facilitates symbolic rules, nonterminals and phrases rather than just concrete ones. This concept will be explored later.
The primitives do not satisfy any convenient preservation properties except a few trivial ones. The denotations $\mathcal{IT}_0$ $[\![\mathsf{id}]\!]$ and $\mathcal{IT}_0$ $[\![\mathsf{fail}]\!]$, for example, are strictly semantics-preserving. For suitably restricted applications of the other operators non-trivial preservation properties could be stated. Consider, for example, the transformation substitute n by n'. The substitution is semantics-preserving modulo renaming if n' is fresh in

Types
$\mathcal{IT}_0$: $\mathrm{Trafo}_0 \to (\mathcal{G} \nrightarrow \mathcal{G})$
$\mathcal{IR}$: $\mathit{Rule} \to (\mathcal{G} \nrightarrow \mathcal{R})$
$\mathcal{IN}$: $\mathit{Nt} \to (\mathcal{G} \nrightarrow \mathcal{U}_N)$
$\mathcal{IP}$: $\mathit{Phrase} \to (\mathcal{G} \nrightarrow (\mathcal{U}_N \cup \mathcal{U}_T)^\star)$

Primitives
$\mathcal{IT}_0$ $[\![\mathsf{id}]\!]\ \gamma = \gamma$
$\mathcal{IT}_0$ $[\![\mathsf{fail}]\!]\ \gamma =$ undefined
$\mathcal{IT}_0$ $[\![\mathsf{reset}]\!]\ \gamma = \emptyset$
$\mathcal{IT}_0$ $[\![\mathsf{add}\ R]\!]\ \gamma = \gamma \cup \{r\}$ where $r = \mathcal{IR}\ [\![R]\!]\ \gamma$
$\mathcal{IT}_0$ $[\![\mathsf{sub}\ R]\!]\ \gamma = \gamma \setminus \{r\}$ where $r = \mathcal{IR}\ [\![R]\!]\ \gamma$
$\mathcal{IT}_0$ $[\![\mathsf{substitute}\ N\ \mathsf{by}\ N']\!]\ \gamma = \cdots$
$\mathcal{IT}_0$ $[\![\mathsf{replace}\ P\ \mathsf{by}\ P']\!]\ \gamma = \cdots$

Yielders
$\mathcal{IR}\ [\![N \to P]\!]\ \gamma = (\mathcal{IN}\ [\![N]\!]\ \gamma) \to (\mathcal{IP}\ [\![P]\!]\ \gamma)$
$\mathcal{IN}\ [\![n]\!]\ \gamma = n$
$\mathcal{IP}\ [\![p]\!]\ \gamma = p$

Fig. 6. Semantics of the transformation primitives

the given grammar. Indeed, the operators for grammar adaptation we are going to derive correspond to restricted combinations of the primitives, so that each adaptation operator can be characterised to be semantics-preserving in one sense for all possible grammars.

3.2 Combinators

In Figure 7, the syntax for combinators is defined. Since the semantic model will be different for primitives and combinators, we establish a new syntactical domain *Trafo* which contains the domain of primitives Trafo_0. There are the following combinators. Firstly, there is construct to restrict the focus for a certain transformation (focus). Secondly, there is simple way to sequence two transformations in the sense of functional composition. Thirdly, there is a form of conditional to choose between two transformations.

$$
\begin{aligned}
\mathit{Trafo} ::= {} & \mathit{Trafo}_0 \\
| {} & \mathsf{focus\ on}\ \mathit{Nt}\ \mathsf{do}\ \mathit{Trafo} \\
| {} & \mathit{Trafo}; \mathit{Trafo} \mid \mathsf{if}\ \mathit{Cond}\ \mathsf{then}\ \mathit{Trafo}\ \mathsf{else}\ \mathit{Trafo} \\
\mathit{Cond} ::= {} & \mathsf{effective}\ \mathit{Trafo} \\
| {} & \mathsf{fresh}\ \mathit{Nt} \mid \mathsf{bottom}\ \mathit{Nt} \mid \mathsf{defined}\ \mathit{Nt} \\
| {} & \mathit{Phrase}\ \mathsf{covers}\ \mathit{Phrase}
\end{aligned}
$$

Fig. 7. Syntax of the transformation combinators

The semantics of the combinators is defined in Figure 8. We use a new semantic function $\mathcal{IT}$ for the denotations of *Trafo*. As its type points out, we stick to another semantic model than for primitives. There is an additional parameter for the current focus of a transformation. The focus is described by a set of nonterminals. The first equation details how primitives are interpreted in the extended model. Actually, the given grammar is restricted to the nonterminals in the focus before the primitive interpreter function $\mathcal{IT}_0$ is applied. Restriction of a grammar γ to some nonterminals $\psi \subseteq \mathcal{D}(\gamma)$ is denoted by $\gamma_{/\psi}$, and it is defined as follows:

$$\gamma_{/\psi} = \{n \rightarrow u \in \gamma \mid n \in \psi\}$$

Types

$\mathcal{IT}$: *Trafo* $\rightarrow \mathcal{P}_{fin}(\mathcal{U}_N) \rightarrow (\mathcal{G} \not\rightarrow \mathcal{G})$

$\mathcal{IC}$: *Cond* $\rightarrow \mathcal{P}_{fin}(\mathcal{U}_N) \rightarrow (\mathcal{G} \not\rightarrow \{false, true\})$

Combinators

$\mathcal{IT}\ [\![T_0]\!]\ \psi\ \gamma = (\mathcal{IT}_0\ [\![T_0]\!]\ \gamma_{/w}) \cup \gamma_{/\mathcal{D}(\gamma)\setminus\psi}$ for $T_0 \in \mathit{Trafo}_0$

$\mathcal{IT}\ [\![\mathsf{focus\ on}\ N\ \mathsf{do}\ T]\!]\ \psi\ \gamma = \mathcal{IT}\ [\![T]\!]\ \{n\}\ \gamma$ where $n = \mathcal{IN}\ [\![N]\!]\ \gamma$

$\mathcal{IT}\ [\![T; T']\!]\ \psi\ \gamma = \mathcal{IT}\ [\![T']\!]\ \psi\ (\mathcal{IT}\ [\![T]\!]\ \psi\ \gamma)$

$\mathcal{IT}\ [\![\mathsf{if}\ C\ \mathsf{then}\ T\ \mathsf{else}\ T']\!]\ \psi\ \gamma = \begin{cases} \mathcal{IT}\ [\![T]\!]\ \psi\ \gamma, & \text{if } c = true \\ \mathcal{IT}\ [\![T']\!]\ \psi\ \gamma, & \text{if } c = false \end{cases}$

where $c = (\mathcal{IC}\ [\![C]\!]\ \psi\ \gamma)$

Conditions

$\mathcal{IC}\ [\![\mathsf{effective}\ T]\!]\ \psi\ \gamma = \gamma \neq \gamma'$ where $\gamma' = \mathcal{IT}\ [\![T]\!]\ \psi\ \gamma$

$\mathcal{IC}\ [\![\mathsf{fresh}\ N]\!]\ \psi\ \gamma = n \notin \mathcal{N}(\gamma)$ where $n = \mathcal{IN}\ [\![N]\!]\ \gamma$

$\mathcal{IC}\ [\![\mathsf{bottom}\ N]\!]\ \psi\ \gamma = n \in \bot(\gamma)$ where $n = \mathcal{IN}\ [\![N]\!]\ \gamma$

$\mathcal{IC}\ [\![\mathsf{defined}\ N]\!]\ \psi\ \gamma = n \in \mathcal{D}(\gamma)$ where $n = \mathcal{IN}\ [\![N]\!]\ \gamma$

$\mathcal{IC}\ [\![P\ \mathsf{covers}\ P']\!]\ \psi\ \gamma = u \Rightarrow^*_\gamma v$ where $u = \mathcal{IP}\ [\![P]\!]\ \gamma, v = \mathcal{IP}\ [\![P']\!]\ \gamma$

Abbreviations

T effectively = if effective T then T else fail

guard C = if C then id else fail

guard $\neg$ C = if C then fail else id

Fig. 8. Semantics of the transformation combinators

The interpretation of the focus-construct also details that the remainder of the grammar outside of the focus is preserved. Note that the actual semantics of the focus-construct allows conditions to be evaluated w.r.t. the complete grammar and not just w.r.t. the focused fragment. This is essential for the formulation of the grammar adaptations. The interpretation of sequential composition and the conditional is straightforward. Conditions used to form conditionals are concerned with the kinds of nonterminals (fresh, bottom, and defined), with the effectiveness of transformations (effective), and with the relation on phrases if one covers the other (cover).

A condition of the form effective T evaluates to *true* if the result of the transformation is defined and the resulting grammar is different from the given one. The evaluation returns *false* if the resulting grammar is equal to the given one. The Boolean value of the condition is not defined if the transformation is not defined. The condition fresh N tests if a nonterminal N is neither defined nor used, similarly for bottom nonterminals (bottom N) and defined nonterminals (defined N).[5]

Finally note the convenient abbreviations defined in Figure 8. T effectively is used to scope a transformation T to be effective. guard C and guard $\neg$ C

[5] As for orthogonality, guard bottom is equivalent to guard $\neg$ fresh; guard $\neg$ defined.

model guards with a positive condition C or a negative condition $\neg\, C$ as trivial forms of a conditional. Guards are convenient to be placed as pre- and postconditions of transformations.

3.3 Yielders

The way how we established the syntax and semantics for primitives and combinators, we prepared for symbolic rules, nonterminals, and phrases, say yielders. The evaluation of such yielders returns concrete rules, nonterminals, and phrases. One reason for this approach is to attempt a homogeneous semantics, where all kinds of syntactical domains of the underlying object language are subject to explicit semantic domains and semantic functions. In the sequel, we illustrate one useful extension relying on this generality. Regarding the actual operator suite to be represented in the sequel, this extension is convenient in the context of unfolding.

Syntax extension

$$\mathit{Phrase} ::= \cdots \mid \mathsf{rhs\ of}\ \mathit{Nt}$$

Semantics extension

$$\mathcal{IP}\ [\![\mathsf{rhs\ of}\ N]\!]\ \gamma = \begin{cases} u, & \text{if } \gamma_{/n} = \{n \to u\}, u \in (\mathcal{N}(\gamma) \cup \mathcal{T}(\gamma))^{\star} \\ \text{undefined}, & \text{otherwise} \end{cases}$$
$$\text{where } n = \mathcal{IN}\ [\![N]\!]\ \gamma$$

Fig. 9. An application of yielders

The form of yielder defined in Figure 9 suggests another form of phrases. The yielder $\mathsf{rhs\ of}\ N$ denotes the RHS of the context-free rule for the nonterminal N. Indeed, this construct is useful for a convenient definition of unfolding, where a nonterminal is replaced (in terms of the primitive $\mathsf{replace}$) by the RHS of the nonterminal.

4 The operator suite

Operators for grammar adaptation are defined in the sequel. These operators are meant to mimic the roles one naturally encounters while adapting grammars manually. The definition of all the operators covered in this section are shown in Figure 10. The operators can be subdivided into three groups, namely refactoring, construction, and destruction. These groups are discussed in some detail below. Various properties of the operators such as semantics preservation, reversibility, orthogonality, and completeness will be pointed out.

4.1 Refactoring

The group of refactorings in Figure 10 are useful to improve or to tune the structure of grammars so that comprehensibility is improved, or subsequent adaptations are easier to perform. These operators cover the

Refactoring		
rename N to N'	=	guard fresh N'; (substitute N by N' effectively); guard fresh N
fold P to N	=	guard fresh N; (replace P by N effectively; add $N \to P$)
unfold N	=	replace N by rhs of N effectively
introduce N as P	=	guard fresh N; (add $N \to P$ effectively)
eliminate N	=	reject N; guard fresh N
Construction		
generalise P to P'	=	guard P' covers P; (replace P by P' effectively)
include P in N	=	guard defined N; (add $N \to P$ effectively)
resolve N as P	=	guard bottom N; (add $N \to P$ effectively)
unify N to N'	=	guard bottom N; guard $\neg$ fresh N'; replace N by N'; guard fresh N
Destruction		
restrict P to P'	=	guard P covers P'; (replace P by P' effectively)
exclude P from N	=	sub $N \to P$ effectively; guard defined N
reject N	=	guard defined N; focus on N do reset
separate N as N'	=	guard $\neg$ fresh N; guard fresh N'; replace N by N'; guard $\neg$ fresh N
delete P	=	guard $\neg$ P covers ϵ; (replace P by ϵ effectively)

Fig. 10. Operator suite for grammar adaptation

usual fold/unfold modulations discussed for various formalisms elsewhere [14]. The operators fold and unfold support folding and unfolding for grammars in a fundamental manner. Introduction and elimination of "superfluous" nonterminals is facilitated by the operators introduce and eliminate. Renaming, that is consistent substitution of nonterminals by others, is facilitated by the rename operator. Refactoring operators are strictly semantics-preserving or at least semantics-preserving in an only slightly relaxed sense. This is discussed later in detail.

There are some specialties we should point out. Unfolding is restricted to nonterminals defined by exactly one rule. This is implied by the use of rhs of ... in the definition of unfold. Also, the role of unfolding and rule elimination is separated. Thereby unfolding can be performed in a focus. Note that all the operators check that they actually affect the given grammar (sometimes indirectly). Note also the way in which guards are placed as pre- and postconditions to restrict the nonterminals used in the various refactoring combinators.

Let us discuss, for example, the pre- and postconditions of rename N to N'. The precondition guard fresh N' makes sure that N is not renamed to a nonterminal N' which is not fresh in the given grammar. Thereby, it is avoided that defined nonterminals N and N' are confused. This requirement is common for renaming. On the other hand, the postcondition guard fresh N makes sure that *all* occurrences of N are substituted. The necessity of the condition is implied by the focus construct. The condition makes sure that the renaming is not disqualified by applying the operator accidentally in a focus.

4.2 Construction

The second group of operators in Figure 10 facilitates construction in a rather broad sense, that is extension, completion and others. Let us discuss the various operators in some detail.

generalise P to P' The phrase P is replaced by the more general phrase P'. This is one convenient way to extend the generated language. This operation is particularly interesting if extended BNF expressiveness was considered.

include P in N Another straightforward way to extend the generated language is to add a rule for a nonterminal N already defined in the given fragment. The precondition makes sure that indeed there is a definition to be extended. Compared to the operator generalise, the operator include works at the rule level and not at the level of phrases inside rules.

resolve N as P This operator is used to provide the definition for a bottom nonterminal N. Note that both include and define essentially add rules, but due to the guards the roles of increasing and defining transformations cannot be confused when applying these operators.

unify N to N' The bottom nonterminal N is unified with (or "connected" to) the nonterminal N' in the sense that N is replaced by N' as illustrated in Figure 11. The precondition also makes sure that N' is not fresh because otherwise renaming and unification are confused. The postcondition documents that unification should not be applied in a focus. This restriction is needed because local unification of nonterminals can hardly be characterised in a convenient way as far as preservation properties are concerned.

$$
\begin{array}{l} x \to \ldots b \ldots \\ y \to \ldots b \ldots \\ z \to \ldots \end{array}
\xrightarrow{\text{unify } b \text{ to } z}
\begin{array}{l} x \to \ldots z \ldots \\ y \to \ldots z \ldots \\ z \to \ldots \end{array}
$$

$$
\begin{array}{l} x \to \ldots b \ldots \\ y \to \ldots b' \ldots \\ z \to \ldots b' \ldots \end{array}
\xrightarrow{\text{unify } b \text{ to } b'}
\begin{array}{l} x \to \ldots b' \ldots \\ y \to \ldots b' \ldots \\ z \to \ldots b' \ldots \end{array}
$$

Fig. 11. Illustration of the operator unify

The operators for construction are useful in grammar development to complete and to connect a grammar. Missing rules are added by include. Too restrictive phrases are generalised with generalise. Missing definitions of nonterminals are established via define. Nonterminal symbols which should become the same are unified or connected via unify.

4.3 Destruction

The third group of operators in Figure 10 is useful for destructing grammars. Again, destruction is meant here in a broad sense. The destruction group provides essentially the inverse operators of the construction group.[6] The operator restrict replaces a phrase by a more restrictive phrase. The operator exclude excludes one rule for a given nonterminal. The postcondition makes sure that exclude behaves as a decreasing

[6] Exception: The operator delete has no counterpart in the construction group.

rather than a rejecting operator. The latter role is solely modelled by the operator reject, and again accordingly enforced by its postcondition. The operator separate is the inverse of the operator unify from the construction group. Thus, the operator separate "breaks up connections" in the grammar, that is it replaces nonterminal occurrences by a fresh nonterminal as illustrated in Figure 12. The postcondition ensures that the separation does not degenerate to a renaming. In fact, the postcondition can only be met if the operator separate is applied in a focus. Finally, the operator delete removes a phrase from the rules. It has been illustrated in the introduction. The precondition of delete is insightful because it makes sure that this removal cannot be done in a more modest way that is by restricting the phrase of concern to ϵ via the more disciplined operator restrict.

$x \to \dots z \dots$	focus on y do	$x \to \dots z \dots$
$y \to \dots z \dots$	separate z as b	$y \to \dots b \dots$
$z \to \dots$	$\Longrightarrow$	$z \to \dots$

Fig. 12. Illustration of the operator separate

As the operators for construction are useful in grammar development to complete and to connect a grammar, the destruction operators are useful for correction or revision. Too general phrases can be restricted. Superfluous rules or definitions can be cancelled. Accidentally unified phrases can be separated by introducing new nonterminals in certain occurrences. Refactoring steps usually precede construction and destruction.

4.4 Discussion

Let us discuss the properties of the various operators. The semantics-preservation properties are summarized in Figure 13. The figure also shows the approximate inverse operation for each operator (except for delete). The operators from the refactoring group are semantics-preserving in a narrow sense, say strictly semantics-preserving or semantics-preserving modulo renaming or except a nonterminal. The two groups for construction and destruction are not semantics-preserving in the narrow sense. Several operators experience the grammar relations for embeddings and prefixes, that is the corresponding transformations are increasing, decreasing, defining or rejecting.[7]
The distribution of the various preservation properties in Figure 13 documents a kind of orthogonality of the operator suite. Most operators experience exactly one kind of preservation property in a pure way except for the operators unify, separate, and delete. All grammar relations or preservation properties resp. are covered by the pure operators. The impure operations have been introduced for convenience. In fact, reconstructions in terms of the pure operators can be provided. These reconstructions are also useful to understand the exact preservation properties

[7] Proofs of the properties in Figure 13, and of the correctness of some other subsequent statements are not very instructive. We prefer a semi-formal discussion.

Operator	Semantics Preservation	Inverse
Refactoring		
rename	modulo renaming	rename
fold	except introduced nonterminal	unfold; eliminate
unfold	strict	fold
introduce	except introduced nonterminal	eliminate
eliminate	except superfluous nonterminal	introduce; include; ...
Construction		
generalise	increasing	restrict
include	increasing	exclude
resolve	defining	reject
unify	essentially defining	separate
Destruction		
restrict	decreasing	generalise
exclude	decreasing	include
reject	rejecting	define; include; ...
separate	essentially rejecting	unify
delete	rejecting; defining	

Fig. 13. Properties of the derived operators

of the impure operators. For brevity, we only consider the reconstruction for unify:

$$\begin{array}{l}\mathsf{unify}\ N\ \mathsf{by}\ N' = \\ \ \ \mathsf{guard}\ \neg\ \mathsf{fresh}\ N';\ \mathsf{resolve}\ N\ \mathsf{by}\ N';\ \mathsf{unfold}\ N;\ \mathsf{eliminate}\ N\end{array}$$

Superficially, unify is a defining operator since it "connects" one nonterminal N to another N', that is N is defined in terms of N'. As we see, the transformation sequence above is indeed essentially a defining transformation. The nonterminal N is only defined temporarily. Its definition is eliminated just after unfolding the definition. The complete transformation sequence is not strictly defining because the result of the entire transformation has one nonterminal less than the input grammar of the transformation. Thereby, we deviate from the narrow definition of a defining operator.

Reconstruction of separate and delete in terms of the pure operators is slightly more tedious. This is also the reason why we did not use such a pure formulation in Figure 10.

Besides the kind of orthogonality discussed above, we can also state a kind of completeness of the operator suite w.r.t. the grammar relations in the sense that the operators in a sense exhaust these relations. Thus, we have the following:

$$\begin{array}{c}\text{if } \gamma\ \mathcal{R}\ \gamma' \text{ with } \mathcal{R} \in \{\equiv, \stackrel{\rho}{\equiv}, \stackrel{\pi}{\equiv}, \stackrel{\leq}{\equiv}, \stackrel{\Rightarrow^*}{\equiv}\}, \\ \text{then there is transformation } T \in \mathit{Trafo} \text{ so that } \mathcal{IT}\ [\![T]\!]\ \mathcal{D}(\gamma)\ \gamma = \gamma'.\end{array}$$

5 Concluding remarks

As far as common practice is concerned, there is little discipline in grammar engineering. Actual grammar development (including the development of new grammars, grammar maintenance, grammar recovery) is usually done in an informal and unstructured manner. Grammar development is often conceived as coding effort, as grammar (conflict) hacking. In our opinion, grammars should be regarded much more as real engineering artifacts [5]. On the other hand, grammars are not too much of a research topic anymore with a small number of exceptions [15, 3, 5, 4]. Research should supply the formal methods useful in grammar development. The present paper contributes in this context. The paper lays down the foundations of an adaptive style for grammar development. Adaptations of grammars are represented as well-studied program transformations. The ultimate grammar programmer probably does not write down the verbose transformations discussed in the paper, but he rather uses a simple grammar engineering environment providing not just support for interactive grammar transformations but also for grammar assessment, testing by parsing, coverage measurement and others.

To call one group of grammar adaptations refactorings, has been inspired by the idea of refactoring in (object-oriented) programming [12, 6]. Indeed, the intentions coincide, that is programs or grammars resp. are adapted in a semantics-preserving manner to improve their structure and to prepare subsequent extensions or even revisions. Recall that the other groups of adaptation operators discussed in the paper are in inherent manner not semantics-preserving in the narrow sense. The major body of research on transformational programming assumes semantics-preservation [13]. The important contribution of the present paper is the set of weaker preservation notions which are suitable to characterise corrections, extensions and others. It is conceivable that the adaptive approach developed for grammars in the present paper can also be instantiated in a meaningful manner for some programming languages. The style of adaptive grammar programming developed in the paper scales up to real grammar projects [11, 10], and it helps to actually make these projects feasible and predictable.

The present paper does not explore several dimensions in depth. Modular grammars [1] and expressiveness beyond simple BNF as for example in SDF [7, 17] are not considered in the formalisation. Grammar transformations are useful for several applications not covered by this paper. In [19], grammar transformations are used for the development of domain-specific languages starting from reusable syntax components. In [18], semantics-preserving grammar transformations are used to derive abstract syntaxes from concrete syntaxes. Grammar transformations are also involved in parser implementation to eliminate list constructs or to optimize grammars w.r.t. certain grammar classes [3, 2, 16].

References

1. S. R. Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, Faculty of En-

gineering, Department of Electronics and Computer Science, Mar. 1991.

2. J. Aycock and N. Horspool. Faster Generalized LR Parsing. In S. Jähnichen, editor, *Proceedings of the eight International Conference on Compiler Construction*, volume 1575 of *LNCS*, pages 32–46. Springer-Verlag, 1999.
3. D. Blasband. *Automatic Analysis of Ancient Languages*. PhD thesis, Free University of Brussels, 2000. Available via `http://www.phidani.be/homes/darius/thesis.html`.
4. M. Brand, M. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117, 1998. Available at http://adam.wins.uva.nl/~x/ref/ref.html.
5. M. de Jonge and J. Visser. Grammars as Contracts. to appear, 2000.
6. M. Fowler. *Refactoring—Improving the Design of Existing Code*. Addison Wesley, 1999.
7. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
8. IBM Corporation. *VS COBOL II Application Programming Language Reference*, Release 4, Document number GC26-4047-07 edition, 1993.
9. IBM Corporation. *VS COBOL II Reference Summary*, Release 4, Document number SX26-3721-05 edition, 1993.
10. R. Lämmel and C. Verhoef. VS COBOL II Grammar Version 1.0.3. `http://www.cwi.nl/~ralf/grammars`, 1999–2000.
11. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. Submitted, available at `http://www.cwi.nl/~ ralf/`, July 2000.
12. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
13. H. A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.
14. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.
15. M. Ruckert. Continuous Grammars. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 303–310, New York, NY, USA, 1999. ACM Press.
16. J. J. Sarbo. Grammar transformations for optimizing backtrack parsers. *Computer Languages*, 20(2):89–100, May 1994.
17. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Sept. 1997.
18. D. S. Wile. Abstract syntax from concrete syntax. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 472–480. ACM Press, 1997.
19. D. S. Wile. Integrating Syntaxes and their Associated Semantics. Draft, 1999.