Goethe University, Frankfurt
Department 12: Computer Science and Mathematics
Institute of Computer Science

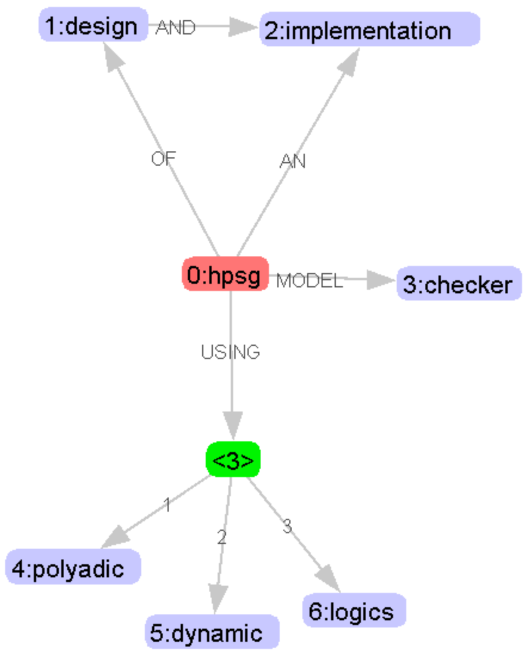
Master Thesis

Design and Implementation of an HPSG Model Checker using Polyadic Dynamic Logics

William B. Blacoe

Supervisor:
Prof. Dr. Manfred Schmidt-Schauß
Artificial Intelligence and Software Technology

June 30th, 2011



Abstract

We present an interactive graphical Java implementation of an HPSG Model Checker. The Model Checker's input consists of a type hierarchy, a set of feature structured HPSG derivations and a set of grammar principles. The feature structured mo-dels are polyadic Kripke structures. Søgaard and Lange [2009] have identified an extension of dynamic logic for such structures whose model checking complexity is in P: Polyadic propositional dynamic logic. Formulas of this logic are used to express linguistic constraints on the models. The input components are entered by way of a GUI, including the ability to import LKB and TRALE output structures. Type hierarchies and models can be viewed and manipulated via graph visualisation. The evaluation of consistency among formulas and models can be overtly analysed step by step.

Contents

Chapter 1

Introduction

The language faculty of the human brain has been an ongoing field of research for decades. As one of the cognitive sciences, linguistics has proven continually difficult to formalise and understand analytically. Fortunately, other disciplines such as neuroscience and psychology help in discovering and describing its workings. Computer science is another wonderful aid in this endeavour. Its compatibility with linguistics comes from treating the human brain as a computational device, a symbol manipulator. Thus, logical and mathematical descriptions are applied to mental processes concerned with language production and comprehension. Put simply, the former may be viewed as a parse from meaning to sound and the latter vice versa.

The present work is not concerned with parsing, but rather with model checking. Nevertheless the notion of the latter is very much based on the former. In this introduction we will first address natural language parsing by means of head-driven phrase structure grammar (henceforth HPSG, Pollard and Sag [1994]). Next, the logics for analysing those structures will be polyadic propositional dynamic logics (PPDL_2), as put forth by Søgaard and Lange [2009]. We will illustrate their interaction comprehensively with several examples. Finally, we will name the further chapters in this work which will show us how the programming language Java was used to implement our model checker.

The result will be a graphical tool for creating linguistic structures and formal grammar principles which describe them. Their mutual consistency can then be calculated by an efficient[1] model checker. From a logical standpoint, they are consistent iff the defined principles are valid in every state of those structures. Our tool makes this process and the material involved transparent, due to a step-by-step analysis and a full visualisation. This lends the program didactical value and an application for debugging and developing related computer-linguistic software, e.g. a parser.

1.1 HPSG

We picked HPSG for dealing with computational-linguistic tasks because it has evolved in a highly formalised fashion in the literature. HPSG has been used for creating formal theories on human language syntax. But, in principle, it also allows for any other kind of information to be encoded in its structures, including phonological, morphological, semantical or pragmatical information. The attribute-value matrix (henceforth AVM) is the typical representation

[1]see Theorem 5.5 *The model checking problem for $PPDL_2$ is in P* in Søgaard and Lange [2009] for the formal proof.

of HPSG feature structures. In what follows we show several AVMs together with their corresponding model to provide the reader with an immediate comparison (see chapter 5 for more details on our visualisation of feature structures). This section explains HPSG principles via AVMs, as is usual in HPSG literature, while section 1.2 works with the concurrent models.

To start out with an example, consider the AVM in figure 1.1 (left), which represents the word "snores". At the top the italic *word* tells us that this structure is of type *word*. Underneath that, its attributes PHON, SYN and SEM are listed (always upper-case), representing respectively some very simple phonological, syntactic and semantic descriptions of "snores". Each attribute's value (positioned to its right) is then a structure itself, i.e. either a type, a list of structures, or an AVM. PHON's value is a singleton list containing the type *snores* and SYN's value is an AVM of type *syn*, etc.

$$\begin{bmatrix} \textit{word} & \\ \text{PHON} & \langle \textit{snores} \rangle \\ \text{SYN} & \begin{bmatrix} \textit{syn} & \\ \text{HEAD} & \textit{verb} \\ \text{NUM} & \textit{singular} \\ \text{PER} & \textit{3rd} \\ \text{SUBCAT} & \langle [\text{SYN}\ [\text{HEAD}\ \textit{noun}]] \rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \textit{predication} & \\ \text{INDEX} & s \\ \text{REL} & \textit{snore\_rel} \\ \text{ARG} & \langle i \rangle \end{bmatrix} \end{bmatrix}$$

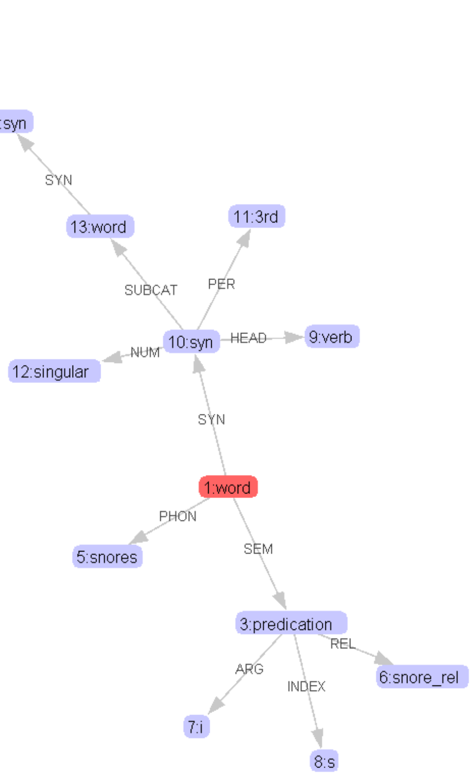


Figure 1.1: AVM for "snores" (left) and its corresponding model (right)

The phonological values in this chapter's examples are simply a list of types in the order in which they are spoken. Each is atomic, thus abbreviating its respective phonological material. The head of a word is a category token indicating the word's character. Number (NUM) and person (PER) values are important for syntactic agreement. Each word may use the SUBCAT attribute to specify a list of other structures it subcategorises or "fits with". In this case, the verb "snores" needs to go together with one word whose nature (or head) is noun.

But what are AVMs good for? Before we continue our introduction of HPSG, let us motivate its use by remembering some classical approaches to syntax. Figure 1.2 shows a classical derivation tree of the English sentence "Tim snores". Its leaves are the words of the sentence. Next, they are abstracted to their syntactical category (N for noun, V for verb).

They then each constitute a phrase on their own, i.e. noun phrase (NP) and verbal phrase (VP). These make up a full sentence (S), by means of the phrase structure rule S $\rightarrow$ NP VP.

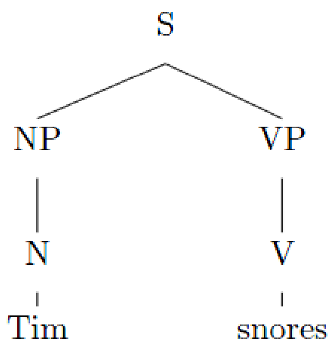


Figure 1.2: A classical syntax tree for “Tim snores”

There are two views of this process. The first one is that we start out with the root symbol S and successively apply phrase structure rules. When this process finishes we end up with leaf nodes labelled with words which, when strung together from left to right, yield a sentence. The second view taken is when we start out with the string of words and merge them to internal phrase nodes using the same phrase structure rules “backwards”. If there is a way of doing this which leads to a single sentence node S, the input is recognised as grammatical or well-formed.

When parsing, though, we do more than just get a “yes” or a “no” to the question whether an uttered sentence is well-formed. We translate it into a structure that represents its meaning. When a human hears a sentence, she looks up the words she hears in her mental lexicon. There she finds entries of so-called signs. A sign is an abstract collection of information about the sound and meaning of a word, besides some further formal information. Her brain then searches for a way to combine some set of signs in such a way that the phonological value of the resulting complex sign equals the heard sentence. In the process each sign's semantic content gets “dragged along” so that the SEM value of the resulting sign is the meaning of the uttered expression.

Figure 1.1 gives us the sign found in the lexicon for “snores” and figure 1.3 gives us the one for “Tim”. A parser would now look for some other sign that serves as a mother sign over the the two daughter signs just mentioned. We are not concerned here in what manner this may be achieved. Rather, we are interested in making sure that the selected derivation obeys all grammar rules. This is model checking. In this view, then, the rules **describe** existing derivations rather than **generating** them. If all rules are satisfied, the derivation is well-formed. To give some examples, consider sentences such as “Tim are snoring” and “Tim takes”. The former is ruled out because its subject does not agree with its verb for number, the latter because “takes” subcategorises at least two arguments, i.e. a subject and an object. To formalise this, the NUM value for “are snoring” is *plural* and the SUBCAT value for “takes” is ⟨[SYN [HEAD *noun*]], [SYN [HEAD *noun*]]⟩.

The mother sign that we are looking for is a phrase whose PHON value is the concatenation of its daughters' PHON values and whose HEAD value is identical with that of its head-daughter. Furthermore, it must not subcategorise any signs, i.e. it must expect no merges with further signs. Figure 1.4 shows an AVM with the minimal attributes required of a mother

$$\begin{bmatrix} \textit{word} & \\ \text{PHON} & \langle \textit{tim} \rangle \\ \text{SYN} & \begin{bmatrix} \textit{syn} & \\ \text{HEAD} & \textit{noun} \\ \text{SUBCAT} & \langle\ \rangle \\ \text{NUM} & \textit{singular} \\ \text{PER} & \textit{3rd} \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \textit{reference} & \\ \text{INDEX} & i \end{bmatrix} \end{bmatrix}$$

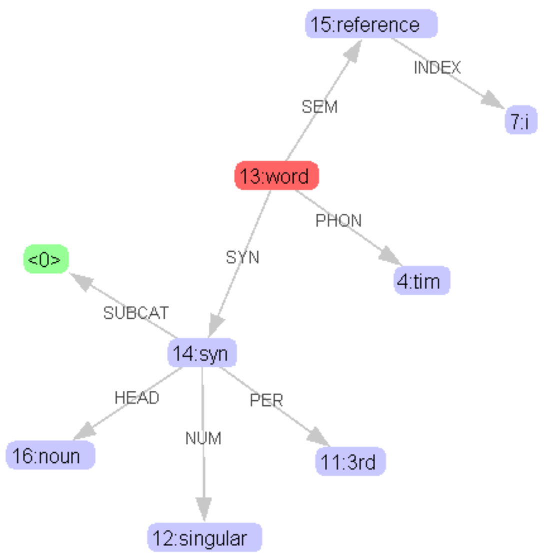


Figure 1.3: AVM for “tim” (left) and its corresponding model (right)

sign over “tim” and “snores”.

$$\begin{bmatrix} \textit{phrase} & \\ \text{PHON} & \langle \textit{tim, snores} \rangle \\ \text{SYN} & \begin{bmatrix} \textit{syn} & \\ \text{HEAD} & \textit{verb} \\ \text{SUBCAT} & \langle\ \rangle \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \textit{predication} & \\ \text{INDEX} & s \\ \text{REL} & \textit{snore\_rel} \\ \text{ARG} & \langle i \rangle \end{bmatrix} \end{bmatrix}$$

Figure 1.4: AVM for “tim snores” (root sign only)

Now, to display a full derivation, we form a tree made of all signs from the lexicon together with the found mother signs. All mother-daughter relationships are also encoded simply by adding the attributes HEAD-DTR (head-daughter) and DTRS (daughters) to each mother sign. The resulting structure is in figure 1.5.

This is the entire derivation in one structure. The squared numbers are so-called tags. Tags with the same numbers indicate structure sharing (also known as re-entrancy), i.e. more than one path (of features) leading to the same sub-structure. Thus, the structure of AVMs is not necessarily tree-like. HPSG structures may even potentially contain cycles. Depicting an AVM as a graph is therefore very useful. In figure 1.5 these re-entrancies are a consequence of unifying the structures for the verb and its subject. The derivation’s graph makes structure sharing even more obvious: different edge paths lead to the same node. The graph’s nodes or

$$\begin{bmatrix} \textit{phrase} & \\ \text{PHON} & \langle \boxed{1}, \boxed{2} \rangle \\ \text{SYN} & \begin{bmatrix} \textit{syn} & \\ \text{HEAD} & \boxed{8} \\ \text{SUBCAT} & \langle\,\rangle \end{bmatrix} \\ \text{SEM} & \boxed{4} \\ \text{HEAD-DTR} & \begin{bmatrix} \textit{word} & \\ \text{PHON} & \langle \boxed{2}\textit{snores} \rangle \\ \text{SYN} & \begin{bmatrix} \textit{syn} & \\ \text{HEAD} & \boxed{8}\textit{verb} \\ \text{NUM} & \boxed{6}\textit{singular} \\ \text{PER} & \boxed{7}\textit{3rd} \\ \text{SUBCAT} & \langle \boxed{3} \rangle \end{bmatrix} \\ \text{SEM} & \boxed{4}\begin{bmatrix} \textit{predication} & \\ \text{INDEX} & \textit{s} \\ \text{REL} & \textit{snore\_rel} \\ \text{ARG} & \langle \boxed{5} \rangle \end{bmatrix} \end{bmatrix} \\ \text{NON-HEAD-DTR} & \boxed{3}\begin{bmatrix} \textit{word} & \\ \text{PHON} & \langle \boxed{1}\textit{tim} \rangle \\ \text{SYN} & \begin{bmatrix} \textit{syn} & \\ \text{HEAD} & \textit{noun} \\ \text{SUBCAT} & \langle\,\rangle \\ \text{NUM} & \boxed{6} \\ \text{PER} & \boxed{7} \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \textit{reference} & \\ \text{INDEX} & \boxed{5}\textit{i} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

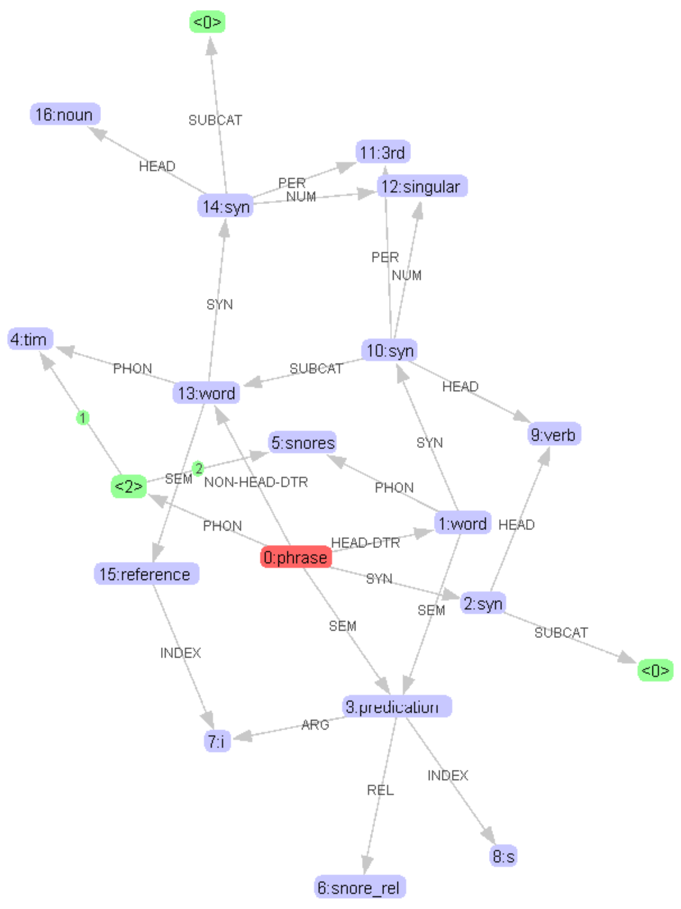


Figure 1.5: AVM for the full derivation of "tim snores" (left) and its corresponding model (right)

states are labelled with types, and its edges are labelled with names of attributes, also called features.

Another typical component of HPSG is the type hierarchy. It defines what types exist, what features they (are allowed to) have and what their inheritance relations are. This may be summarised in a graph such as that in figure 1.6.

This type hierarchy contains only what is necessary for our little example of "Tim snores". *tim* and *snores* are types that have the super-type *phon* in common. *word* and *phrase* are both sub-types of *sign*. We see that *sign* has three appropriate features: PHON, SYN and SEM because they are listed in the same node. Each feature specifies its value type. This can be any type in the hierarchy, it does not have to be a leaf type. Sub-types inherit these features and may even further specify their value types. PHON expects either *phon* or any of its sub-types as its value in all models based on this type hierarchy. *i* and *j* are variables for some referent; *s* and *t* are situation variables for some situation semantics. They, too, can be treated as types. They have the super-type *index* in common. The type that generalises over all other types is the top type *top*. It must have no super-type.

While type hierarchies typically act as an ontology for HPSG models, in our program

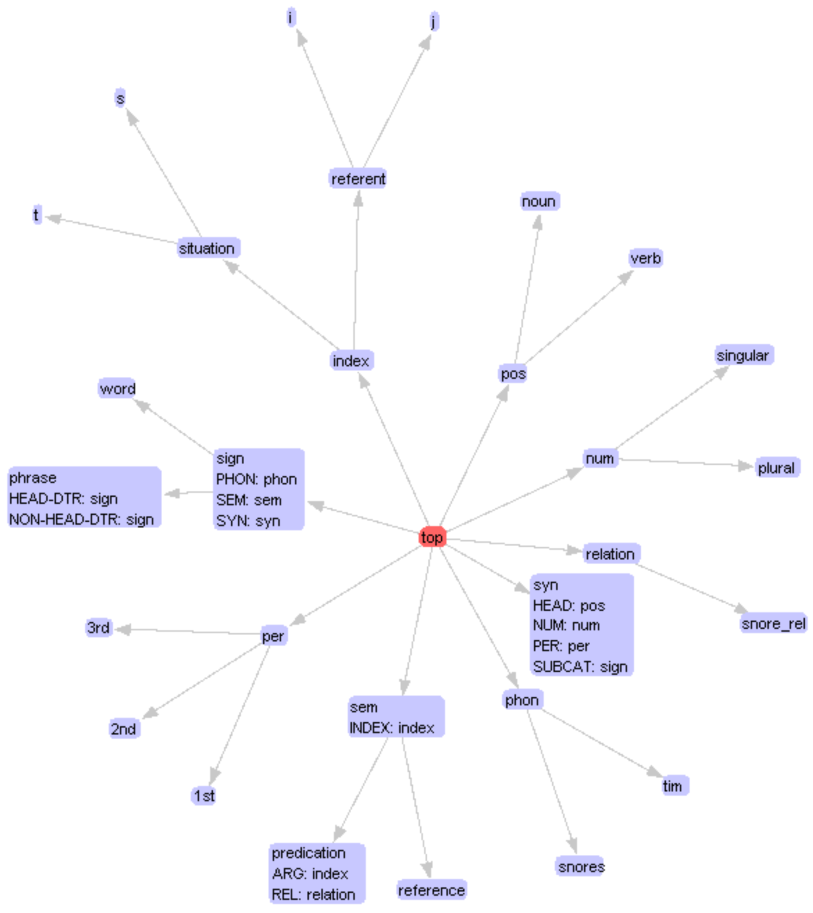


Figure 1.6: A type hierarchy for our examples

the user is free to build models independent of this notion. Since there are many ways of interpreting such an ontology, we leave it for the user to decide the restrictions placed thereby. One example is feature appropriateness: Are type t's features allowed for t, or actually necessary? Another example is this: Must each model state contain a leaf type or are generalised (i.e. non-leaf) types also allowed? These restrictions and many more can be expressed by virtue of PPDL_2 formulas.

1.2 Polyadic Propositional Dynamic Logic

Besides the type hierarchy and models there is one more input: A theory, i.e. a set of formulas which must be satisfied by the models in order for these to be declared well-formed. We employ a version of polyadic propositional dynamic logics (henceforth PPDL), as introduced in Søgaard and Lange [2009]. There is a more expressive and a less expressive version of this. The latter was coined PPDL_2. It cannot describe as many linguistic phenomena as the former, but it nevertheless is an attractive compromise because of its efficiency: the model checking problem is in P. That is, given a type hierarchy, a model and a PPDL_2 formula, the question whether the formula is satisfied by the model based on this type hierarchy can be answered deterministically in polynomial time.

Before we work with the models that appeared in the above figures, let us consider their nature: They are polyadic Kripke structure, i.e. graphs with labelled nodes (states) and labelled edges, including hyper-edges. A hyper-edge has only one label, but it can point to several successor states "at the same time" (see figure 1.7(d)). It may also point to no successor states (see figure 1.7(b)). This is different from a state without the feature (see figure 1.7(a)). A non-list edge, as in figure 1.7(c), is equivalent with a hyper-edge that points to exactly one state.

In our graph visualisations we simulate a hyper-edge via a so-called list node and its outgoing list edges. The list node is labelled with the amount of value states written in angled brackets. The n list edges are labelled with the numbers from 1 to n, thus marking their order in the represented list. A list node may not have another list node as a value state.

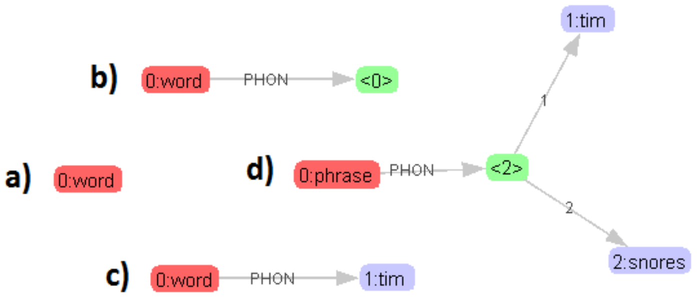


Figure 1.7: Basic Kripke structures. a) A state with no successor, b) A state with an empty list as PHON value, c) A state with on PHON-successor, d) A state having as its PHON-successor a list containing two states

Let's look at some simple $PPDL_2$ formulas that are applicable to these models. The simplest of formulas contains nothing more than a type, e.g. *word*. The formula *word* is satisfied in state 0 by the models (a), (b) and (c) in figure 1.7. For model (d) this is not the case because here state 0 contains the distinct type *phrase*. In fundamental dynamic logics diamond formulas such as $\Diamond\phi$ describe a state to have a successor state which fulfills the sub-formula ϕ. Box formulas like $\Box\phi$ assert that all successor states satisfy ϕ. This is applicable to graphs whose edges are unlabelled and not hyper-edges. Since we are dealing with graphs with those properties, though, $PPDL_2$ allows diamonds and boxes (i.e. modalities) to contain the feature that is to be considered, and after the modality may come an arbitrary amount of sub-formulas. Thus $\langle \text{PHON} \rangle (tim)$ is satisfied in states which have an outgoing edge labelled PHON that points to a state labelled with type *tim*, e.g. state 0 in model (c) but not in the other three models. Instead, state 0 in models (b) and (d) respectively satisfy $\langle \text{PHON} \rangle ()$ and $\langle \text{PHON} \rangle (tim, snores)$. With this intuition, we can now consider the propositional logics among types and the construction of more complex diamonds (also called diamond modalities).

Types are the propositional variables that formulas mention. Features appear in formulas as modal operators. Our formulas do not use the existential or universal quantification found in first order predicate logics. Instead they only ever affect a strictly local part of the model[2]. That is, a state satisfies a formula iff it and the referred-to states in its vicinity fulfill the

[2]With perhaps the exceptions of ϵ and *

expressed requirements. Since the size of a formula is constrained, the number of states that are considered in the formula's evaluation is also constrained.

We make mention of a state's neighbouring state by using the connecting feature in a diamond modality. $(\textit{syn} \rightarrow \langle \text{HEAD} \rangle (\textit{verb} \vee \textit{noun}))$, for example, says that if a state is of type *syn* its HEAD-successor must fulfill the formula $(\textit{verb} \vee \textit{noun})$, i.e. it must be of type *verb* or *noun.* We call HEAD a program, in this case an atomic program because it is only a feature. This is one way to express the need for appropriateness, as given in the type hierarchy, i.e. it is appropriate for *syn* to have *pos* (part of speech) as HEAD feature, and *pos* in turn may be *verb* or *noun.* It is typical for HPSG rules to be implications because they constrain the way in which derivations may behave. These constraints are often based on the (sub-)derivation's root type.

1.3 Complex examples

The following are some examples of more complex formulas and their evaluation. Here we cover all modal operators provided by PPDL$_2$. These explanations are semi-formal. If the reader choses, she may already consult chapter 2 for the precise definitions.

1.3.1 Non-empty Phonology

We demand that every sign have a non-empty phonological representation using $(\textit{sign} \rightarrow \langle \text{elem(PHON)} \rangle (\top))$. This is an implication regarding *sign.* Since *word* and *phrase* are sub-types of *sign*, all states containing either of these types must fulfill the subformula $\langle \text{elem(PHON)} \rangle (\top)$. This formula's diamond contains a non-atomic prorgram.

To explain its denotation step by step, let us consider all appearances of the PHON feature in figure 1.5's model: states 13 and 1 each point directly to a *phon* state (i.e. *tim* and *snores*, respectively). But state 0 points to a list node which in turn has two *phon* value states. To indicate their order, the edges are numbered 1 and 2. Internally the polyadic Kripke structure saves this particular data in the form of a relation. For every feature f it contains a polyadic relation R_f. The tuples therein tell us what state points to what states. In our current model we have $R_{\text{PHON}} = \{(13, 4), (1, 5), (0, 4, 5)\}$. Each tuple's first state (the so-called initial state) is where its edge labelled PHON originates, and all further states in that tuple (the non-initial states) are what this edge points to, either directly or via a list node. If the tuple is binary the logics and the visualisation make no difference of whether this might express a singleton list value or a non-list value. All tuples containing only an initial state or more than one none-initial state are linked up in the model graph via a list node.

Because $\langle \text{elem(PHON)} \rangle (\top)$ is formulated in ignorance of the denoted relation's arity we need to make use of the elem operator. It makes all tuples with more than one non-initial state binary. Singleton tuples are discarded and binary tuples stay the way they are. That is, tuples like (x, y) are unaffected, while tuples such as $(x, y_1, ..., y_n)$ are split up into binary tuples that all share the source state x: $(x, y_1), ..., (x, y_n)$. This relation is considered virtual because it is not necessarily found in the model. These newly formed tuples are only part of the evaluation and not the original derivation. Now that we have computed the program's denotation, we can check every state s whether it contains a tuple that originates in s and points to some state that satisfies the formula $\top$. Since $\top$ is satisfied in every state, we are thus demanding that s have a PHON feature and that it be non-empty. If so, this formula is evaluated as true in s. Because of our original implication $(\textit{sign} \rightarrow \langle \text{elem(PHON)} \rangle (\top))$, the explained sub-formula only needs to be satisfied in all *word*, *phrase* and *sign* states. All other states trivially satisfy the implication.

1.3.2 Subcategorisation Principle

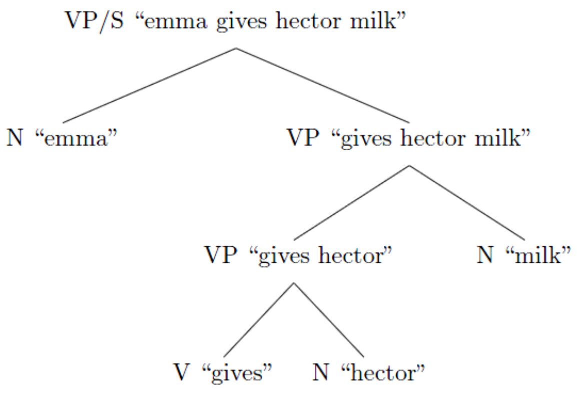


Figure 1.8: HPSG-like syntax tree of "Emma gives Hector milk". Each node abbreviates a sign, mentioning its syntactic category and phonology.

$$
\begin{array}{l}
(\textit{phrase} \rightarrow \\
\quad \langle \text{elem}(\ominus((\text{HEAD-DTR; SYN}), \text{SUBCAT}, \text{NON-HEAD-DTR}) \cap (\text{SYN; SUBCAT}))\rangle(\top) \\
\qquad \vee \\
\quad \langle(\ominus((\text{HEAD-DTR; SYN}), \text{SUBCAT}, \text{NON-HEAD-DTR}) \cap (\text{SYN; SUBCAT}))\rangle() \\
)
\end{array}
$$

Figure 1.9: Subcategorisation Principle

The subcategorisation principle describes the mother-daughter relationship of signs, i.e. verbs might subcategorise nouns, noun phrases, prepositional phrases; nouns might subcategorise determiners, etc. As all the nouns in our current examples are either proper nouns or mass nouns, they each subcategorise nothing. Thus, for every *word* state whose SYN-successor has a HEAD value of *noun*, that state's SYN-successor has an empty list as its SUBCAT value. In *word* states whose SYN-successor has a HEAD value of *verb*, on the other hand, that SYN-successor usually specifies a SUBCAT value list of at least one sign, namely the subject of that sentence.[3]

In our examples we have assumed a binary branching among mother and daughter signs. Each mother has a head-daughter and a non-head-daughter. The former is what gives the mother its character, the latter is an argument which the head-daughter subcategorises. Since the mother now "incorporates" that argument daughter, it inherits the SUBCAT list from its head-daughter with the exception that the non-head-daughter is subtracted from the list.

The complement operator $\ominus$ is used to express this subtraction. It has three arguments: (1) a path leading to a certain state, here (HEAD-DTR; SYN), (2) a feature of that state, here SUBCAT, and (3) a sub-program describing what value is to be subtracted from the

[3]There are exceptions such as sentences with an intransitive verb in the imperative form, e.g. "Leave!".

feature's value. Let us evaluate the formula for the subcategorisation principle step by step in the model that would correspond to the tree in figure 1.8 (not shown). The sign "gives" specifies a SUBCAT list refering to its three arguments "emma", "hector", "milk" in that order. When it combines with "hector", the resulting phrase's SUBCAT list must be "emma", "milk". When that sign combines with "milk", their mother's SUBCAT list now only contains "emma". Finally, after the sign for the subject "emma" is merged, the root sign has an empty SUBCAT list. Each time the list resulting from subtracting a phrase's non-head-daughter from its head-daughter's SUBCAT list must be identical to that phrase's SUBCAT list. This identity of values is expressed, as before, by an intersection.

In the first and second subtraction the list that should appear in the intersection's denotation contains two signs and one sign, respectively. To deal with lists of any length, we apply the elem operator here. Therefore, if there was a list with more than one state they are split up into separate non-list edges as described earlier, and the diamond-subformula is true. If the intersection's denotation is an empty relation, so is that of the elem operator, and the diamond-subformula is false. The latter subtraction results in an empty list. In the model this is represented by a singleton tuple, i.e. there is only a source state and no value states. This would also falsify our principle's first diamond-formula. Hence, we added a disjunct with a very similar diamond-formula. This one has no elem operator as it is only meant for the special case of finding empty SUBCAT lists. The round brackets after the diamond are empty, which is why this diamond-formula is only fulfilled if its program's denotation contains at least one singleton tuple.

1.3.3 Phonology Principle

The phonology principle also deals with lists. Each sign's PHON value tells us how it is spoken. In our example derivation the PHON list of a mother is simply the concatenation of its daughters' PHON lists. Theories of word order deal with the problem of the order in which such lists concatenate. We will not deal with this problem here. Instead, using the formula in figure 1.10, we will just check whether a mother's PHON value is indeed the concatenation of that of its daughters in either order.

$$
\begin{array}{l}
(\textit{phrase} \rightarrow (\\
\quad \langle \text{elem}((\text{PHON} \cap \text{app}(\text{HEAD-DTR, PHON, NON-HEAD-DTR, PHON})))\rangle(\top) \\
\qquad \vee \\
\quad \langle \text{elem}((\text{PHON} \cap \text{app}(\text{NON-HEAD-DTR, PHON, HEAD-DTR, PHON})))\rangle(\top) \\
))
\end{array}
$$

Figure 1.10: Phonology Principle

The app operator appends a list to another list. Its first and third arguments are paths originating in the same state. Its second and fourth arguments are features respectively pointing to the first and second list in question. Since paths are deterministic (see definition 7), the lists to be concatenated are unique if they exist in the model. Say the tuple representing the first list is $(x, y_1, ..., y_n)$, and the tuple representing the second list is $(x, z_1, ..., z_m)$. The second is appended to the first resulting in $(x, y_1, ..., y_n, z_1, ..., z_m)$.

Again we use intersection and elem in a diamond-formula to check whether the two described lists are identical. As we assume no empty PHON lists, there is no need for a sub-formula covering this case. Rather there are two very similar diamond-formulas in a disjunction, allowing for a concatenation in either order. This is necessary for models such as “emma gives hector milk”, because with “gives hector” and “gives hector milk” the non-head-daughter’s PHON list is appended to that of the head-daughter, and with “emma gives hector milk” vice versa.

1.3.4 Acyclicity

Another interesting property of models we can check is whether they contain any cycles. This may be deemed a meta-linguistic property, but we can nonetheless express it using the formula in figure 1.11. Inside this formula the so-called empty formula ϵ appears. It denotes a relation containing a tuple (s, s) for every state s in the model.

$$
\begin{array}{l}
\neg\langle(\\
\quad (((\text{HEAD-DTR} \cup \text{NON-HEAD-DTR})^* \ ; \text{HEAD-DTR}) \\
\qquad \cup \\
\quad ((\text{HEAD-DTR} \cup \text{NON-HEAD-DTR})^* \ ; \text{NON-HEAD-DTR})) \\
\cap\ \epsilon)\rangle(\top)
\end{array}
$$

Figure 1.11: Acyclicity Formula

This formula is fulfilled iff there are no cycles with regards to mother-daughter relationships. The star operator denotes the reflexive-transitive closure of its argument relation. It is limited, though, in that it only operates on binary tuples. Its sub-program may also only be the empty program, an atomic program or the union of atomic programs. PPDL_2 union also only regards binary tuples. Since we assume mother-daughter branchings to be binary, all HEAD-DTR tuples and NON-HEAD-DTR tuples are binary. The sub-program $((\text{HEAD-DTR} \cup \text{NON-HEAD-DTR})^* \ ; \text{HEAD-DTR})$ denotes all paths over HEAD-DTR and NON-HEAD-DTR edges, finishing with a HEAD-DTR edge. The first sub-program composed with this one via $\cup$ denotes all such paths finishing with a NON-HEAD-DTR edge. The intersection then determines whether any of these paths are reflexive (i.e. (s, s) for some state s). There is a cycle iff this is so. The negation in front of the diamond sub-formula then inverts this value.

Any model whose mother-daughter projection is truly tree-like will thus fulfill this acyclicity formula. Figure 1.12 shows a model that does not. Due to the cycle in it, all three states falsify the formula.

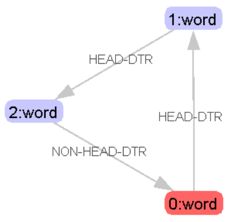


Figure 1.12: A simple model containing a cycle w.r.t. daughter-mother relationships

The only program operator that did not appear in our examples is $\sqcap$. It is called intersection-in-a-point, has two arguments and is written infix. It compares every tuple in the first argument relation with every tuple in the second argument relation. In contrast to the usual intersection, which only maintains tuples of the same length which contain the same states in the same positions, this intersection-in-a-point includes a binary tuple (x, t) for every combination of input tuples $(x, y_1, ..., t, ..., y_n)$ and $(x, z_1, ..., t, ..., z_m)$ it compares. The elem operator is therefore actually a special case of intersection-in-a-point: elem(α) is equivalent to $(\alpha \sqcap \alpha)$.

1.4 Overview of Chapters

Having introduced our approach to linguistic materials and their descriptions, we can now implement a systematic processing tool for them. We use Java for this implementation. The object-oriented paradigm lets us represent types, type hierarchies, states, tuples, relations, models, formulas, programs, etc. as objects that interact and incorporate each other. Chapter 2 makes our formal notion of all involved components precise. Chapter 3 explains our implementation of those components and the model checker itself. The parsers involved in making it possible to import HPSG material from LKB and TRALE, and exporting projects to XML are detailed in chapter 4. The graph visualisations are made possible by the prefuse Java package. Chapter 5 gives an overview of the steps involved. Chapter 6 then serves as a documentation on how to use the Model Checker program, and chapter 7 concludes our dealings with HPSG model checking.

Chapter 2

Formal Definitions

Most of the following definitions have been adopted from Søgaard and Lange [2009], but have been repeated here for convenience. Note that there are slight variations in notation, though.

2.1 HPSG Components

Definition 1 *Signature*

A signature $\Sigma = \langle T, F \rangle$ is a binary tuple where

- T is the set of all types.
- F is the set of all features.

Definition 2 *Polyadic Kripke Structure*

A polyadic Kripke structure (or model) $M = \langle S, \{R_f \,|\, f \in F\}, V \rangle$ over the signature $\Sigma = \langle T, F \rangle$ is a 3-tuple where

- S is the set of states.
- the second component is the set of polyadic relations $R_f \subseteq R_{\text{allowed}}$, one assigned to each feaure f in F. We define $R_{\text{allowed}} = \{(s, t_1, ..., t_n) \mid n \in \{0, ..., |S|\}$ and $s, t_1, ..., t_n \in S$ and $\forall i = 1, ..., n : \forall j = 1, ..., i-1 : t_i \neq t_j\}$ to be the set of all tuples of length at least 1 and at most $|S| + 1$ in which all non-initial states, if there are any, are distinct.

  This restriction is motivated by reasons of tractability. In the literature alternative HPSG structures are allowed to contain sets of states. In our models sets are somewhat simulated by lists containing no duplicates among their non-initial states. There also seems to be no linguistic motivation to give a state a value list with multiple reference to the same state.
- $V : S \to T$ is a valuation function assigning each state its type.

Definition 3 *Type Hierarchy*

A type hierarchy $Th = \langle Ist, Appr \rangle$ over the signature $\Sigma = \langle T, F \rangle$ is a binary tuple where

- *Ist* $: T \to \mathcal{P}(T)$ is a function mapping every type t to the set of all immediate super-types of t, where $\mathcal{P}(T)$ is the power-set of T. By extension, $\mathit{Ist}^* : T \to \mathcal{P}(T)$ is the function mapping each type to the set of all its super-types.
  Formally: $\mathit{Ist}^*(t) = \{t\} \cup \bigcup_{t' \in \mathit{Ist}(t)} \mathit{Ist}^*(t')$.
  The function *Ist* must not allow cycles in the hierarchy, i.e. $t \notin \mathit{Ist}^*(t)$ must hold for all $t \in T$. Furthermore, the type hierarchy must be connected in such a manner that there is exactly one type with no super-type. This is considered the top type.

- *Appr* $: T \times F \to T$ is a partial function, with $\mathit{Appr}(t, f) = t'$ defining that f is an appropriate feature for t and selects t' as its value type.

2.2 Formulas and Programs

Definition 4 *Syntax of Fomulas and Programs*

Let $\Sigma = \langle T, F\rangle$ be a signature. t is some type in T and f is some feature in F. Formulas (ϕ_i) and programs $(\alpha_i, \beta_i, \gamma_i)$ over Σ are constructed as follows:

$$\begin{array}{rcl}
\phi_1, \phi_2, ..., \phi_n & \to & t \mid \neg\phi_1 \mid (\phi_1 \wedge \phi_2) \mid (\phi_1 \vee \phi_2) \mid (\phi_1 \to \phi_2) \mid (\phi_1 \leftrightarrow \phi_2) \\
 & & \mid \langle\alpha_1\rangle(\phi_1, ..., \phi_n) \mid [\alpha_1](\phi_1, ..., \phi_n) \mid \top \mid \bot \\
\alpha_1, \alpha_2 & \to & \epsilon \mid f \mid (\alpha_1 \cap \alpha_2) \mid (\alpha_1 \cup \alpha_2) \mid (\alpha_1; f) \mid \beta_1^* \mid (\alpha_1 \sqcap \alpha_2) \\
 & & \mid \ominus(\gamma_1, f, \alpha_1) \mid \mathrm{app}(\gamma_1, \gamma_2, \gamma_3, \gamma_4) \mid \mathrm{elem}(\alpha_1) \\
\beta_1, \beta_2 & \to & \epsilon \mid f \mid (\beta_1 \cup \beta_2) \\
\gamma_1, \gamma_2, \gamma_3, \gamma_4 & \to & \epsilon \mid f \mid (\gamma_1; f)
\end{array}$$

Following a diamond or box comes a finite sequence of 0 or more sub-formulas in round brackets. We will refer to the set of all well-formed programs and formulas over Σ as Progs_Σ and $\mathit{Formulas}_\Sigma$, respectively.

Definition 5 *Normal Form of Formulas*

Let $\Sigma = \langle T, F\rangle$ be a signature. For every PPDL$_2$ formula ϕ there is an equivalent PPDL$_2$ formula $\phi' = N(\phi)$ in normal form. $N : \mathit{Formulas}_\Sigma \to \mathit{Formulas}_\Sigma$ is defined as follows:

$$N(\phi) = \begin{cases}
\phi & \text{if } \phi \in T \cup \{\top\} \\
\neg N(\psi) & \text{if } \phi = \neg\psi \\
(N(\psi_1) \wedge N(\psi_2)) & \text{if } \phi = (\psi_1 \wedge \psi_2) \\
\neg(\neg N(\psi_1) \wedge \neg N(\psi_2)) & \text{if } \phi = (\psi_1 \vee \psi_2) \\
\neg(N(\psi_1) \wedge \neg N(\psi_2)) & \text{if } \phi = (\psi_1 \to \psi_2) \\
 & \text{if } \phi = (\psi_1 \leftrightarrow \psi_2) \\
\langle\alpha\rangle(N(\psi_1), ..., N(\psi_n)) & \text{if } \alpha \in \mathit{Progs}_\Sigma \text{ and } \phi = \langle\alpha\rangle(\psi_1, ..., \psi_n) \\
\neg\langle\alpha\rangle(\neg N(\psi_1), ..., \neg N(\psi_n)) & \text{if } \alpha \in \mathit{Progs}_\Sigma \text{ and } \phi = [\alpha](\psi_1, ..., \psi_n) \\
\neg\top & \text{if } \phi = \bot
\end{cases}$$

Definition 6 *Semantics of Programs*

Let $M = \langle S, \{R_f \mid f \in F\}, V\rangle$ be a polyadic Kripke structure over the signature $\Sigma = \langle T, F\rangle$. The semantics R_f of features $f \in F$, i.e. atomic programs, are given in the model, whereas the semantics R_α of complex programs $\alpha \in \mathit{Progs}_\Sigma \backslash F$ are obtained recursively in the following manner:

$$\begin{array}{rcl}
R_\epsilon & = & \{(s,s) \mid s \in S\} \\
R_{(\alpha_1 \cap \alpha_2)} & = & R_{\alpha_1} \cap R_{\alpha_2} \\
R_{(\alpha_1 \cup \alpha_2)} & = & \{(s,t) \in S^2 \mid (s,t) \in R_{\alpha_1} \text{ or } (s,t) \in R_{\alpha_2}\} \\
R_{(\alpha_1 \,;\, \alpha_2)} & = & \{(s,t_1,...,t_n) \in R_{\text{allowed}} \mid \exists (s,s') \in R_{\alpha_1} \text{ and } (s',t_1,...,t_n) \in R_{\alpha_2}\} \\
R_{\alpha^*} & = & \{(s,t) \in S^2 \mid \exists s_0,...,s_n : \forall i = 1,...,n : (s_{i-1}, s_i) \in R_\alpha \\
 & & \text{and } s = s_0, t = s_n\} \\
R_{(\alpha_1 \sqcap \alpha_2)} & = & \{(s,s') \mid \exists (s,t_1,...,t_n) \in R_{\alpha_1} \\
 & & \text{and } \exists (s,u_1,...,u_m) \in R_{\alpha_2}, \exists i,j : s' = t_i = u_j\} \\
R_{\ominus(\alpha_1,\alpha_2,\alpha_3)} & = & \{(s,t_1,...,t_{i-1},t_{i+1},...,t_n) \\
 & & \mid \exists (s,s') \in R_{\alpha_1}, \exists (s',t_1,...,t_n) \in R_{\alpha_2}, \exists (s,t_i) \in R_{\alpha_3}\} \\
R_{\text{app}(\alpha_1,\alpha_2,\alpha_3,\alpha_4)} & = & \{(s,t_1,...,t_n,u_1,...,u_m) \in R_{\text{allowed}} \\
 & & \mid (s,s') \in R_{\alpha_1}, (s',t_1,...,t_n) \in R_{\alpha_2}, \\
 & & (s,s'') \in R_{\alpha_3}, (s'',u_1,...,u_m) \in R_{\alpha_4}\} \\
R_{\text{elem}(\alpha)} & = & R_{(\alpha \sqcap \alpha)}
\end{array}$$

Definition 7 *Deterministic Relations*

Let $S = \{s_1, ..., s_n\}$ be a set of states, and let R be a polyadic relation over S so that all tuples in R contain at least one state. R is considered deterministic iff for every state $s \in S$ there is at most one tuple in R whose first state is s.

Definition 8 *Semantics of Formulas*

Let $Th = \langle \mathit{Ist}, \mathit{Appr}\rangle$ be a type hierarchy and $M = \langle S, \{R_f \mid f \in F\}, V\rangle$ a polyadic Kripke structure over the signature $\Sigma = \langle T, F\rangle$. Before a formula ϕ is evaluated it is normalised to ϕ' which is constructed only of $\top$, the types t in T, negation, conjunction and diamond modalities. Thus, for some $s \in S$:

$$\begin{array}{rcl}
M, s \vDash \top & & \\
M, s \vDash t & \Leftrightarrow & t \in \mathit{Ist}^*(V(s)) \\
M, s \vDash (\phi_1 \wedge \phi_2) & \Leftrightarrow & M, s \vDash \phi_1 \text{ and } M, s \vDash \phi_2 \\
M, s \vDash \neg\phi & \Leftrightarrow & M, s \nvDash \phi \\
M, s \vDash \langle\alpha\rangle(\phi_1, ..., \phi_n) & \Leftrightarrow & \exists s_1, ..., s_n \in S : (s, s_1, ..., s_n) \in R_\alpha \text{ and } \forall i = 1, ..., n : M, s_i \vDash \phi_i
\end{array}$$

If $M, s \vDash \phi$ we say that ϕ is satisfied by M in state s. In addition, if $\forall s \in S : M, s \vDash \phi$ we say that ϕ is satisfied by M, or $M \vDash \phi$. The question $M \overset{?}{\vDash} \phi$ is called the **model checking problem** or **model checking question**.

Furthermore, if a theory $\Phi = \{\phi_1, ..., \phi_n\}$ is satisfied by a model M we write $M \models \Phi$. This is the case iff $\forall \phi \in \Phi : M \models \phi$. Upon extending this matter to a collection of models $\mathcal{M} = \{M_1, ..., M_m\}$ we can assert that such a collection satisfies a theory Φ, that is $\mathcal{M} \models \Phi$, iff $\forall M \in \mathcal{M} : M \models \Phi$.

2.3 Model Checking Algorithm

Definition 9 *Model Checking Algorithm*

The algorithm presented on the following pages solves the model checking problem. It consists of several procedures, some of which are recursive. The *check* procedure is the main procedure which makes use of all further procedures to answer the model checking question.

The input consists of a type hierarchy $Th = \langle \mathit{Ist}, \mathit{Appr} \rangle$ and a polyadic Kripke structure $M = \langle S, \{R_f \mid f \in F\}, V \rangle$, both over the signature $\Sigma = \langle T, F \rangle$, and a PPDL$_2$ formula ϕ. M satisfies ϕ iff calling the *check*-procedure returns **true**. The following commentary gives an informal overview of the algorithm's steps, after which the algorithm itself is explicated.

For every state $s \in S$ we maintain a set of formulas labels_s as a global variable. That is, labels_s contains all formulas that s is labelled with. In the end this is exactly the set of all sub-formulas of ϕ' which are satisfied by M in s, where ϕ' is the normalised version of ϕ.

The label sets start out empty. Initally they are filled with all atomic formulas that their respective state satisfies via the *addAtomicLabels* procedure, i.e. the state's type and all its super-types as specified by Th. Of course, every state also satisfies the formula $\top$. Next, the recursive *addComplexLabels* procedure revursively traverses ϕ' bottom-up. As it does so, all encountered sub-formulas satisfied by any state s are added to labels_s. Dealing with negations and conjunctions is straightforward. But if a diamond formula is encountered, its sub-program's denotation is calculated explicitely by the *computeRelation* procedure. This procedure is the explicite deterministic version of what happens in definition 6. The computed relation is then inspected to see which tuples contain states in which the diamond formula's respective sub-formulas are satisfied. The idea of labelling states with sub-formulas in this manor is based on a labelling algorithm described in Blackburn and van Benthem [1988].

After the traversal and labelling have been completed all label sets are inspected to see if they contain ϕ'. Iff this is the case, M satisfies ϕ'. Since ϕ is equivalent to ϕ', iff all label sets contain ϕ', M satisfies ϕ.

```
**procedure** $check()$:
  $\phi' := normaliseFormula(\phi)$
  $addAtomicLabels()$
  $addComplexLabels(\phi')$
  $isSatisfied = \mathbf{true}$
  **for all** $s \in S$:
    **if** $\phi' \notin labels_s$ then
      $isSatisfied = \mathbf{false}$
      **exit for**
  **return** $isSatisfied$


**procedure** $normaliseFormula(\phi)$:
  **return** $N(\phi)$, where $N$ is the normalisation function from definition 5


**procedure** $addAtomicLabels()$:
  **for all** $s \in S$:
    $t := V(s)$
    $labels_s := labels_s \cup \{t, \top\} \cup getIstSet(t)$


**procedure** $getIstSet(t)$:
  **for all** $t' \in Ist(t)$:
    $istSet := istSet \cup getIstSet(t')$
  **return** $istSet$


**procedure** $addComplexLabels(\phi)$:

  **if** $\phi = \neg\psi$ **then**
    $addComplexLabels(\psi)$
    **for all** $s \in S$:
      **if** $\psi \notin labels_s$ **then**
        $labels_s := labels_s \cup \{\phi\}$

  **if** $\phi = (\psi_1 \wedge \psi_2)$ **then**
    $addComplexLabels(\psi_1)$
    $addComplexLabels(\psi_2)$
    **for all** $s \in S$:
      **if** $\psi_1 \in labels_s$ **and** $\psi_2 \in labels_s$ **then**
        $labels_s := labels_s \cup \{\phi\}$
  if $\phi = \langle\alpha\rangle(\psi_1, ..., \psi_n)$ then
    for $i = 1$ to $n$:
      $addComplexLabels(\psi_i)$
    $R = computeRelation(\alpha)$
    for all $(s_1, ..., s_m) \in R$:
      if $m = n + 1$ then
        $allLabelsExist$ = true
        for $i = 1$ to $n$:
          if $\psi_i \notin labels_{s_{i+1}}$ then
            $allLabelsExist$ = false
            exit for
        if $allLabelsExist$ = true then
          $labels_{s_1} := labels_{s_1} \cup \{\phi\}$


procedure $computeRelation(\alpha)$:

  if $\alpha \in F$ then:
    return $R_\alpha$

  if $\alpha = \epsilon$ then
    $R := \emptyset$
    for all $s \in S$:
      $R := R \cup (s, s)$
    return $R$

  if $\alpha = (\delta_1 \cap \delta_2)$ then
    $R_1 := computeRelation(\delta_1)$
    $R_2 := computeRelation(\delta_2)$
    $R := \emptyset$
    for all $t_1 \in R_1$:
      for all $t_2 \in R_2$:
        if $t_1 = t_2$ then
          $R := R \cup \{t_1\}$
    return $R$

  if $\alpha = (\delta_1 \cup \delta_2)$ then
    $R_1 := computeRelation(\delta_1)$
    $R_2 := computeRelation(\delta_2)$
    $R := \emptyset$
    for all $(s_1, ..., s_n) \in R_1$:
      if $n = 2$ then
        $R := R \cup \{(s_1, ..., s_n)\}$
    for all $(s_1, ..., s_n) \in R_2$:
      if $n = 2$ then
        $R := R \cup \{(s_1, ..., s_n)\}$
    return $R$
```

```
if $\alpha = (\delta_1;\delta_2)$ then
  $R_1 :=$ computeRelation$(\delta_1)$
  $R_2 :=$ computeRelation$(\delta_2)$
  $R := \emptyset$
  for all $(s_1, s_2) \in R_1$:
    for all $(t_1, ..., t_n) \in R_2$:
      if $s_2 = t_1$ then
        $R := R \cup \{(s_1, t_2, ..., t_n)\}$
  return $R$

if $\alpha = \delta^*$ then
  $R' :=$ computeRelation$(\delta)$
  $R := \emptyset$
  for all $s \in S$:
    $R := R \cup \{(s, s)\}$
  for all $(s_1, ..., s_n) \in R'$:
    if $n = 2$ then
      $R := R \cup (s_1, s_2)$
  for all $s \in S$:
    for all $t \in S$:
      for all $u \in S$:
        if $(t, s) \in R$ and $(s, u) \in R$ then
          $R := R \cup \{(t, u)\}$
  return $R$

if $\alpha = (\delta_1 \sqcap \delta_2)$ then
  $R_1 :=$ computeRelation$(\delta_1)$
  $R_2 :=$ computeRelation$(\delta_2)$
  $R := \emptyset$
  for all $(s_1, ..., s_n) \in R_1$
    for all $(t_1, ..., t_m) \in R_2$
      if $s_1 = t_1$ then
        for $i = 2$ to $n$:
          for $j = 2$ to $m$:
            if $s_i = t_j$ then
              $R := R \cup \{(s_1, s_i)\}$
  return $R$

if $\alpha = \ominus(\delta_1, \delta_2, \delta_3)$ then:
  $R_1 :=$ computeRelation$(\delta_1)$
  $R_2 :=$ computeRelation$(\delta_2)$
  $R_3 :=$ computeRelation$(\delta_3)$
  $R := \emptyset$
  for all $(s_1, s_2) \in R_1$
    for all $(t_1, ..., t_n) \in R_2$
      for all $(u_1, u_2) \in R_3$
        if $s_1 = u_1$ and $s_2 = t_1$ then:
          for $i = 2$ to $n$:
            if $t_i = u_2$ then:
              $R := R \cup \{(s_1, t_2, ..., t_{i-1}, t_{i+1}, ..., t_n)\}$
  return $R$
```

```
**if** $\alpha = \mathrm{app}(\delta_1, \delta_2, \delta_3, \delta_4)$ **then**
  $R_1 := \mathit{computeRelation}(\delta_1)$
  $R_2 := \mathit{computeRelation}(\delta_2)$
  $R_3 := \mathit{computeRelation}(\delta_3)$
  $R_4 := \mathit{computeRelation}(\delta_4)$
  $R := \emptyset$
  **for all** $(s_1, s_2) \in R_1$
    **for all** $(t_1, ..., t_n) \in R_2$
      **for all** $(u_1, u_2) \in R_3$
        **for all** $(v_1, ..., v_m) \in R_4$
          **if** $s_1 = u_1$ **and** $s_2 = t_1$ **and** $u_2 = v_1$ **then**
            $R := R \cup \{(s_1, t_2, ..., t_n, v_2, ..., v_m)\}$
  **return** $R$

**if** $\alpha = \mathrm{elem}(\delta)$ **then**
  **return** $\mathit{computeRelation}(\delta \sqcap \delta)$
```

Chapter 3

Implementation of Main Components

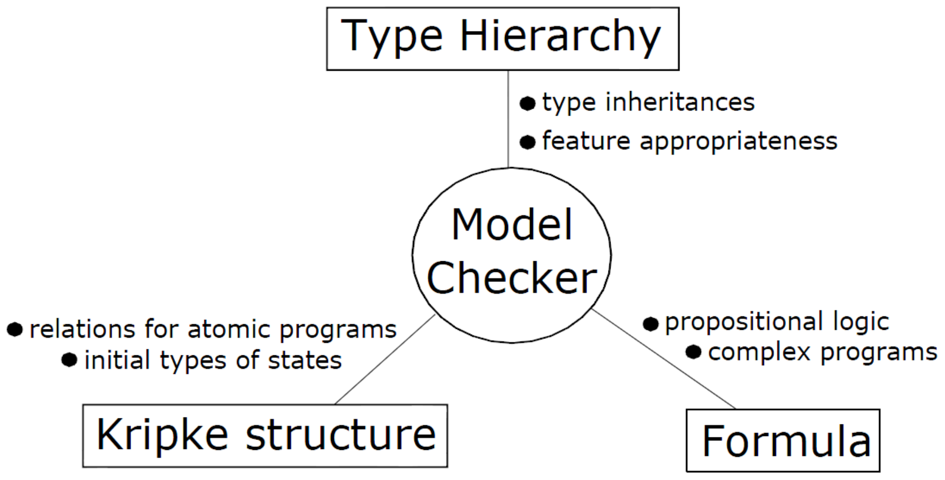


Figure 3.1: The main components and their interaction

This chapter details our Java implementation of the main components: Type hierarchy, polyadic Kripke structure, formula and model checker. Figure 3.1 outlines their relationship. The notion of a signature is not implemented as such. The components mentioned here are not restricted to a given collection of types and features. Rather they may refer to types not covered by the type hierarchy, or to features which have no relation assigned to them by the Kripke structure.

In our implementation no main component maintains any pointers to another. They are separate objects making mention of common types and features, but only by naming them using `String`s rather than by reference. Keeping the main components isolated like this adds to their robustness and interchangeability. During the model checking procedure they come together. If, at this point, a model's state has a type that is not in the type hierarchy, this does not hinder the procedure. It just means that that state can satisfy no other formulas except its own type and $\top$. Similarly, if a formula mentions a feature without any assigned relation in the model, that feature's semantics will be $\emptyset$.

Internally in the main components there are many data structures which employ hash tables, like `HashMap<T1, T2>` and `HashSet<T>`. These are generic data types, where `T1`, `T2` and `T` can be replaced by any Java type. A type hierarchy, for example, polymorphically instantiates its attribute `Map<String, Type>` by a `HashMap<String, Type>`, so that `String`s function as keys, each of them pointing to a `Type`. This data structure may thus be considered a dictionary or an associative array.

Attention must be paid to the `hashCode()` method for objects that are used as keys in a `HashMap` or `HashSet`. When two such objects have the same content, they are still separate instances of their respective class, and thus by convention produce different hash codes. The program can be forced to consider them as equal by overriding the `hashCode()`-method, though. For `Formula`s and `Program`s the new method requires a compositional calculation of that hash code, i.e. each formula's or program's hash code is the product of a fixed factor combined with the hash codes of all sub-formulas and sub-programs.

3.1 Types and Type Hierarchies

To distinguish between a type in a type hierarchy and a type in a Java program, we will call the latter Java type and print Java types and Java code, in a different font, e.g. `String`. To represent types we created the class `Type` with the attributes

```
private String typeName;
private Set<Type> istSet;
private Map<String, Type> featureMap;
```

In our formal definition of type hierarchies, we let the type hierarchy $Th = \langle \mathit{Ist}, \mathit{Appr} \rangle$ govern the inheritance relations among the types and their appropriate features. In the implementation this is handled differently: We save references to a type's immediate super types and its appropriate features in the object itself. Thus, all the structure usually provided in a type hierarchy is already contained in the instances of the class `Type`. Since more than one immediate super type is generally allowed for any type t, we employ a `Set<Type>` to keep references to t's immediate super-types $\mathit{Ist}(t)$. The function Appr is implemented as a `Map<String, Type>`. Because Appr may be a partial function, it suffices to save only defined features in its domain, i.e. every combination $\mathit{type}, \mathit{valueType} \in T, \mathit{feature} \in F$ with $\mathit{Appr}(\mathit{type}, \mathit{feature}) = \mathit{valueType}$ results in the execution of `istSet.put(feature, valueType)` in the `Type`-object `type`.

What exactly does a `TypeHierarchy`-object do then? It still has a governing responsibility over all types and features. In contrast to types, features are not implemented as a class of their own. Rather they appear only in the form of `String`s pointing to types or relations. Each `TypeHierarchy` keeps the following data structures as attributes:

```
private Map<String, Type> typeMap;
private TreeSet<String> featureSet;
private Type topType;
```

Since it is one of our main components, the other components will communicate with the type hierarchy rather than with any `Type`-objects. Obviously, models and formulas mention types, but there only the types' names appear as `String`s rather than `Type`s. Therefore, a `TypeHierarchy` provides an interface with other components by virtue of methods such as `addType(String typeName)`, `getType(String tyeName)`, `addIst(String typeName, String` `istName)` and `addFeature(String typeName, String featureName, String valueTypeName)`. In order to quickly find the `Type`-object corresponding to each `String` typeName, a `Map<String, Type>` is used by the aforementioned methods. Even though feature appropriateness is encoded into each `Type`, the `TypeHierarchy` maintains its own `TreeSet<String>` of feature names. This will prove useful later on. A reference to the top type of the type hierarchy is saved for the model checking procedure. When having created a type hierarchy from an input, instead of demanding a certain type to be declared the top type (i.e. the hierarchy's root type) explicitely, it suffices to call the method `updateTopType()`. This will begin at an arbitrary type in the hierarchy and recursively follow one arbitrary reference in the current type's set of immediate super types, until a type with no immediate super type is encountered. This then must, by definition, be the top type.

3.2 Kripke Structures: States, Tuples and Relations

Polyadic Kripke structures $M = \langle S, \{R_f \,|\, f \in F\}, V\rangle$ are implemented as instances of the class `KripkeStructure` whose attributes are

```
private Map<Integer, State> stateMap;
private Map<String, Relation> relationMap;
private int stateCounter;
```

The first attribute corresponds to S, except that it assigns each state a number in order to differenciate between states with the same type. The integer `stateCounter` is incremented each time a newly added state saves it as one of its attributes, starting at 0. The set $\{R_f \,|\, f \in F\}$ becomes a `Map<String, Relation>`-object where each feature `String` f is mapped to a `Relation`-object. As in the conventional definition of polyadic relations, each `Relation` contains a set of `Tuples`, each of which in turn is an ordered list of an arbitrary number ($\geq$ 1) of `States`. Each state contains information about its number and (the name of its) type, i.e. the type specified by the valuation V. Again, although `States`, `Tuples` and `Relations` have their internal data structures, other main components will only manipulate them by interfacing with the `KripkeStructre`-object through methods such as `addState(String initialTypeName)`, `getState(int number)`, `addTuple(String relationName, Tuple tuple)`, `addRelation(String relationName)`.

3.3 Formulas

Java's object orientation will serve us well in the construction of formulas. We can implement the tree structure of formulas easily by supplying a class for each connective. We can further define the kind and amount of arguments for each connective according to the syntax of formulas. All formula classes generalise to class `Formula`. The relevent class information on atomic formulas $t \in T$ and complex formulas is summarised in figure 3.2.

As mentioned in section 3.1 `AtomicFormulas` only mentions the name of their type, rather than pointing to the actual `Type`. The Java type `ArrayList<Formula>` is an ordered and possibly empty list of `Formula`-objects. For more information on `IProgram` see section 3.4.

Formula	Class	Class Attributes
t	`AtomicFormula`	`String typeName`
$\neg\phi$	`NegationFormula`	`Formula subFormula`
$(\phi_1 \wedge \phi_2)$	`ConjunctionFormula`	`Formula subFormula1, subFormula2`
$(\phi_1 \vee \phi_2)$	`DisjunctionFormula`	`Formula subFormula1, subFormula2`
$(\phi_1 \rightarrow \phi_2)$	`ImplicationFormula`	`Formula subFormula1, subFormula2`
$(\phi_1 \leftrightarrow \phi_2)$	`BiimplicationFormula`	`Formula subFormula1, subFormula2`
$\langle\alpha\rangle(\phi_1, ..., \phi_n)$	`DiamondFormula`	`IProgram subProgram,` `ArrayList<Formula> subFormulaList`
$[\alpha](\phi_1, ..., \phi_n)$	`BoxFormula`	`IProgram subProgram,` `ArrayList<Formula> subFormulaList`
$\top$	`TopFormula`	
$\bot$	`BottomFormula`	

Figure 3.2: Implementation of formulas

3.4 Programs

Similarly to formulas, programs are objects put together as a tree. This time the leaves or atomic programs are features $f \in F$. The result of our implementation of all program operators according to the syntax of programs is summarised in figure 3.3.

Noticeably, some of the prescribed attributes are of a Java type not listed in the column of classes. These are Java interfaces rather than Java classes. This is indicated by their prefix '`I`'. Java interfaces are employed for the implementation of programs to make multiple inheritance possible, the most general being `IProgram`. In a simplified class diagramm, figure 3.4 displays all inheritances among interfaces (with prefix '`I`') and classes (without prefix '`I`') associated with programs.

When a class inherits from a class, or when an interface inherits from an interface, the declaration of that class or interface in the Java code is succeeded by `extends`, e.g. `public interface IAtomicUnionProgram extends IProgram`. Whenever a class inherits from an interface, `implements` is used instead. For our purposes there is no need to differentiate between `extends` and `implements`, though. Only the possibility of multiple inheritance among Java types is relevant here.

The need for this multiple inheritance originates from the different classifications of programs in the syntax for programs: α_i, β_i and γ_i from definition 4 may be called normal programs, atomic union programs and atomic composition programs, respectively. The latter two are restricted for reasons of model checking complexity. Nevertheless, we end up with two different kinds of composition and union. To avoid having to implement them as separate classes, multiple inheritance through Java interfaces is employed.

Now it becomes obvious why a `UnionProgram` takes two `IProgram` arguments while an `AtomicUnionProgram` takes two `IEmptyOrAtomicOrAtomicUnionProgram` arguments. Analogously, a `CompositionProgram` takes firstly an `IProgram` argument, but an `AtomicCompositionProgram` must take an `IEmptyOrAtomicOrAtomicCompositionProgram` as its first argument. The `IterationProgram` is also confined to arguments of Java interface `IEmptyOrAtomicOrAtomicUnionProgram`.

The advantage in this differenciation can be seen in the syntax of the $\ominus$ and app operators.

Operator	Class	Class Attributes
ϵ	`EmptyProgram`	
f	`AtomicProgram`	`String featureName`
$(\alpha_1 \cap \alpha_2)$	`IntersectionProgram`	`IProgram subProgram1, subProgram2`
$(\alpha_1 \cup \alpha_2)$	`UnionProgram`	`IProgram subProgram1, subProgram2`
$(\alpha_1 \cup \alpha_2)$	`AtomicUnionProgram`	`IEmptyOrAtomicOrAtomicUnionProgram subProgram1, subProgram2`
$(\alpha_1; \alpha_2)$	`CompositionProgram`	`IProgram subProgram1, AtomicProgram subProgram2`
$(\alpha_1; \alpha_2)$	`AtomicCompositionProgram`	`IEmptyOrAtomicOrAtomicCompositionProgram subProgram1, AtomicProgram subProgram2`
α^*	`IterationProgram`	`IEmptyOrAtomicOrAtomicUnionProgram subProgram`
$(\alpha_1 \sqcap \alpha_2)$	`IntersectionInAPointProgram`	`IProgram subProgram1, subProgram2`
$\ominus(\alpha_1, \alpha_2, \alpha_3)$	`ComplementProgram`	`IEmptyOrAtomicOrAtomicCompositionProgram subProgram1, AtomicProgram subProgram2, IProgram subProgram3`
$\mathrm{app}(\alpha_1, \alpha_2, \alpha_3, \alpha_4)$	`AppendProgram`	`IEmptyOrAtomicOrAtomicCompositionProgram subProgram1, subProgram2, subProgram3, subProgram4`
$\mathrm{elem}(\alpha)$	`ElemProgram`	`IProgram subProgram`

Figure 3.3: Implementation of programs

Wherever an argument must be an `IEmptyOrAtomicOrAtomicCompositionProgram`, it functions as a path syntactically, and as a pointer to certain value tuples in the model semantically. By forcing these compositions to be atomic, the resulting relations remain deterministic.

3.5 Model Checker

Finally, the main component, that brings all other main components together, is the model checker. Even though model checking is a procedure, we treat it as a component, as it is encapsulated in a Java class when implemented. Its purpose is to answer the question $\mathcal{M} \overset{?}{\models} \Phi$ for some collection of models $\mathcal{M}$ and a theory Φ.

All involved components need to be handed over to the `ModelChecker`-object for checking. For this purpose, they are compiled into a `Project`, i.e. an object with the following class attributes.

```
private TypeHierarchy th;
private ArrayList<ModelGroup> modelGroupList;
private Theory theory;
```

`TypeHierarchy` has already been introduced. The other attributes are straightforward: A `ModelGroup` is just a class containing an `ArrayList<KripkeStructure>` called `modelList`.

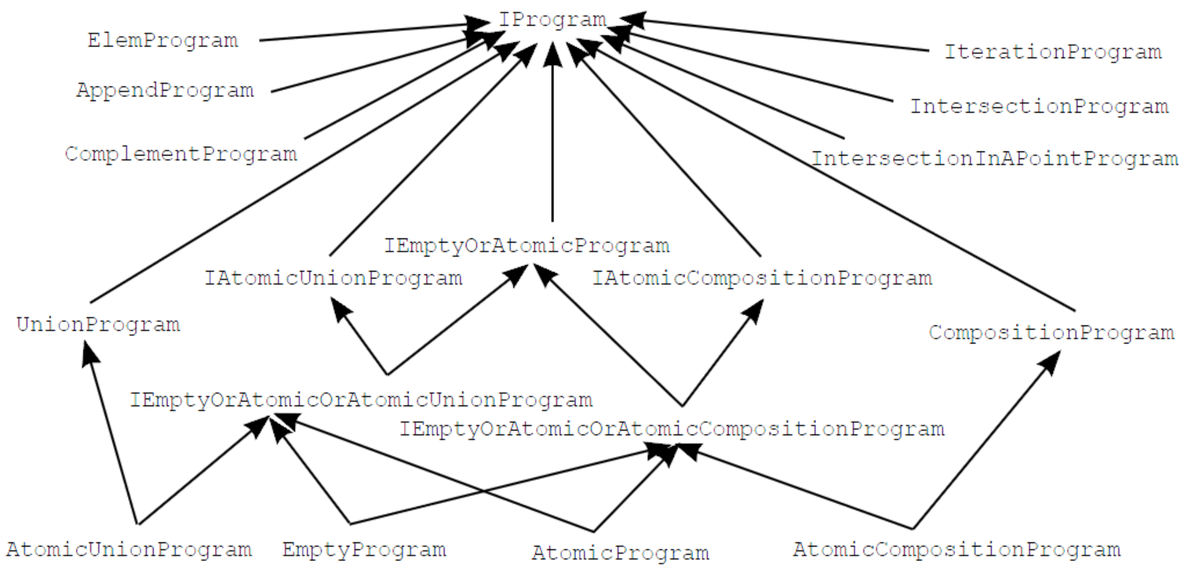


Figure 3.4: Inheritance among program interfaces and classes

Several `ModelGroup`s are then contained in an `ArrayList<ModelGroup>` in the project, for organising a possibly vast amount of models.

From the given project the `ModelChecker` checks each combination of model $M = \langle S, \{R_f \mid f \in F\} V \rangle$ and formula ϕ. For this we employ an implementation of the algorithm in definition 9. For this purpose our class `ModelChecker` supplies the data structures

```
private Map<IProgram, Relation> extendedRelationMap;
private Map<State, Set<Formula>> stateLabelMap;
```

Their usage is similar to that of `KripkeStructure`'s class attributes. All information from M is contained in them, but in an extended form. Instead of mapping `String`s to `Relation`s, we now use the actual `IProgram`s as map keys. This `Map<IProgram, Relation>` is named `extendedRelationMap` because it will not only contain the relations R_f for all $f \in F$, but also all relations denoted by complex programs that appear in ϕ'. The `Map<State, Set<Formula>>` is what records the state labels. Each state in M is mapped to a set of `Formula`s, i.e. the set of labels.

With these data structures in place, we can now proceed with checking ϕ' against M. The following methods implement the mentioned formal algorithm.

- `adoptRelationsFromModel(KripkeStructure model)`

  This method iterates through the model's `Map<String, Relation> relationMap`, saving the same relations in the `extendedeRelationMap`, except that this time `AtomicPrograms` that are based on the respective feature `String` are used as map keys.

- `createAtomicLabels(TypeHierarchy th, KripkeStructure model)`

  Now each state collects labels in the form of atomic formulas. The two atomic labels that every state s will maintain is $\top$ and $V(s)$, i.e.

`addLabelToState(currentState, new TopFormula());` and
`addLabelToState(currentState, new AtomicFormula(`
`currentState.getInitialTypeName().toLowerCase()));` are called. Beyond these, s is labelled with all super-types of $V(s)$. Each `Type` has a built-in recursive method `getAllSuperTypes()` that returns a `Set<Type>`-object, i.e. the set of all super-types. It performs a calculation that is analogous to that of Ist^* in definition 3.

  If the type $V(s)$ is not in the project's type hierarchy the latter labelling obviously is skipped.

- `normalizeFormula(Formula formula, KripkeStructure model)`

  For our labelling algorithm each formula ϕ must be in normal form ϕ', i.e. it may only be constructed from $\top$, types $t \in T, \wedge$ and $\langle\alpha\rangle$ where $\alpha \in \mathit{Progs}$. Therefore, a normalisation method was implemented which works in the same way as the function N in definition 5. Additionally, though, whenever this method encounters a `DiamondFormula` or `BoxFormula` it extracts that formula's `subProgram` and includes its denoted relation in the `extendedeRelationMap` by calling
  `extendedRelationMap.put(subProgram, program.getSemantics(model))`. If this program should refer to any feature for which the model has defined no relation, the program is included in the `extendedeRelationMap` anyway, but it is mapped to an empty relation, i.e. $\emptyset$.

- `createComplexLabels(Formula formula, KripkeStructure model)`

  Now that our formula is normalised to ϕ' and all atomic labels are assigned, we can go about evaluating ϕ' in a bottom-up fashion. The workings of the `createComplexLabels` method are described by the *addComplexLabels* procedure from defnition 9.

- `isFormulaSatisfiedByModel(Formula formula, KripkeStructure model)`

  After having labelled every state with all sub-formulas of ϕ' which it satisfies, the only step remaining is performed by going through all states and seeing if they are labelled with ϕ' itself. If there is a state without this label, it will be pointed out by the method, so that the user can be informed which state falsifies which formula. If no such state is identified the formula is declared satisfied by the model.

Chapter 4

Input and Output

This chapter is concerned with the implementation of several different interfaces with the Model Checker program: (1) Importing models and type hierarchies from other programs, (2) converting formula strings that were entered by hand to Java objects, and (3) converting the main input components to XML files and back.

4.1 Input from LKB and TRALE

Importing input material is not an integral part of the program. But it gives the user a chance to see models and type hierarchies produced by two other parsers instead of having to create her own from scratch. These programs are LKB [Copestake, 2002] and TRALE [Carpenter and Penn, 1999]. We tested them using "LKB for windows"[1] and the TRALE parser pre-installed on Stefan Müller's bootable Linux CD called Grammix [Müller, 2007a].

The following gives a glimpse of our effort to parse files that contain type hierarchies and feature structures to `TypeHierarchy` and `KripkeStructure` objects, respectively. The principle remains the same in all four parsers: Recursively descending top-down LL parsing without back-tracking. They are each adapted to the specific kind of structure at hand in an ad hoc fashion.

4.1.1 LKB

Figure 4.1 shows an excerpt from a `types.tdl` file from the LKB grammar `g8gap` described in Copestake [2002].

```
unary-rule := phrase &
[ ORTH #orth,
  SEM #cont,
  ARGS < [ ORTH #orth, SEM #cont ] > ].
```

Figure 4.1: An entry from an LKB grammar's `types.tdl`

[1] Downloadble at `http://wiki.delph-in.net/moin/LkbInstallation`. The bootable Linux CD available at `http://depts.washington.edu/uwcl/twiki/bin/view.cgi/Main/KnoppixLKB` is an alternative, but we did not test this.

An LKB grammar constitutes several files, but we will focus on this one, as it is useful for creating a type hierarchy. This example encodes the definition of type *unary-rule*. In `types.tdl` there is such an entry for each basic type. Left of "`:=`" the defined type is mentioned. To the right of "`:=`" come several expressions, seperated by "`&`", that define this type explicitely and implicitely. The type *phrase* is an immediate super-type of *unary-rule*; but what follows the "`&`" is an attribute-value matrix describing a property of all states with the type *unary-rule*. Words starting with a "`#`" are variables for reentrancy. "`<`" and "`>`" indicate lists. Upper-case words are features. Thus, this entry tells us that each state with the type *unary-rule* must be such that its ORTH successor and its SEM successor have a value which is respectively compatible with the ORTH and the SEM successor of the first state in the list of its ARGS successor. In other terms, this rule may be expressed as the feature structure description in figure 4.2.

$$\textit{unary-rule} \rightarrow \textit{phrase} \wedge \begin{bmatrix} \text{ORTH} & \boxed{1} \\ \text{SEM} & \boxed{2} \\ \text{ARGS} & \left\langle \begin{bmatrix} \text{ORTH} & \boxed{1} \\ \text{SEM} & \boxed{2} \end{bmatrix} \right\rangle \end{bmatrix}$$

Figure 4.2: LKB entry as feature structure description

Our LKB type hierarchy parser turns all data in a `types.tdl` file in to a `TypeHierarchy` in three steps:

1. **Tokenising**
   This makes use of regular expressions. After all Prolog comments, line breaks, etc. are removed and the LKB entries are neatly separated, each entry's tokens are split and saved in an array of token strings. From here they are each converted to that token object which corresponds with the string's content.

2. **Creating a `Construct` tree**
   The parser now linearly goes through this sequence of token objects and builds a tree of `Construct` objects. Each represents a part of the input, e.g. `CType`, `CAvm`, `CList`, `CReference`, etc. The leaves of this tree are all token objects such as `TFeature`, `TType` and `TReference`.

3. **Creating a type hierarchy from the `Construct` tree**
   This tree is the product of the actual recursive descent. Once it is established we traverse it bottom-up in order to simultaneously build a `TypeHierarchy` with the same structure. `Type`s are instantiated and augmented with features via the content of the `TType` and `TFeature` objects. Multiple inheritance is easily realised because in a hierarchy each type's name is unique.

Next, we take a look at "LKB for windows" to demonstrate how to come by model data. That data text is then parsed into a `KripkeStructure` in much the same way as above.

The "LKB Top" window is shown in figure 4.3. From here a grammar is loaded by selecting "Load → Complete Grammar" from the menu, and then a file `script` from an LKB grammar folder. This file internally loads all files involved in that folder's grammar. If we load the `g8gap` grammar we are presented with its type hierarchy in graph form (see figure

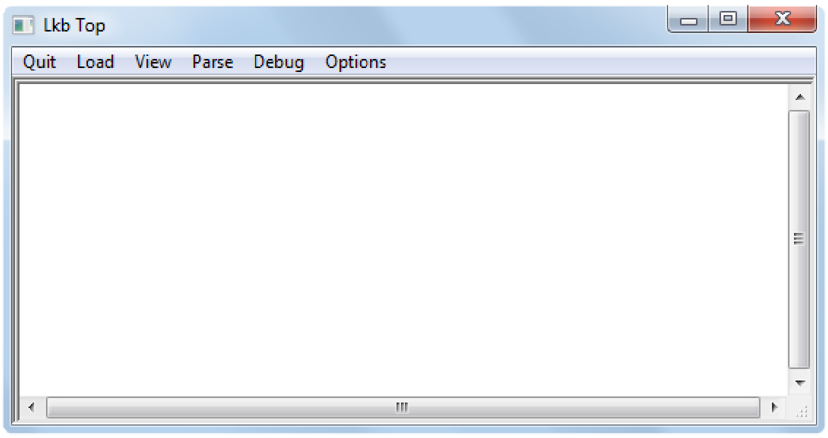


Figure 4.3: The “LKB Top” window

4.4). Types whose names starts with “glbtype” are quasi-types inserted into the hierarchy in order to satisfy the greatest lower bounds condition [Copestake, 2002].

If we now select “Parse → Parse input” from the menu, we are prompted to input an expression which consists of the words provided by the grammar, in our case “the”, “this”, “that”, “these”, “those”, “aardvark”, “dog”, “cat”, “bark”, “chase”, “to”, “near”, “give”. If successful, the parse is shown in a new window containing at least one small parse tree. By clicking on it and selecting “Show enlarged tree” a window with the same tree opens. This parse tree is larger, though, and allows the user to inspect its individual nodes. The parse tree only shows the syntactic backbone of the selected HPSG derivation. In order to see all information involved in the derivation, a context menu appears when clicking on a node. From this “Feature structure” may be selected. If that node is the root of the parse tree, the user may view the feature structure of the entire derivation this way (e.g. “The dog barks”, see figure 4.5).

The display of the feature structure is not quite in the conventional way, but looks quite similar. There is, however, an option provided for exporting the given information to a $\TeX$-file (e.g. figure 4.6), thus rendering it to the well-known style of an AVM (figure 4.7). We are interested in this option because it is a way of saving the derivation into a format that can then be translated to a model in the form of a Kripke structure.

The text in figure 4.6 can now be parsed using the same kind of steps mentioned above. Of course our `LkbModelParser` has its own token and construct objects for tokenising and parsinig. This time step 3 creates a `KripkeStructure` object from the `Construct` tree. Interestingly, there is no need for polyadic relations in the resulting `KripkeStructure` because LKB represents lists in an implicite way in its AVMs. Figure 4.7’s ORTH sub-structure, for example, represents the string “the dog barks”, which would explicitely be [ORTH $\langle$*the*, *dog*, *barks*$\rangle$]. The implicite way nests AVMs of some list type, dividing each sub-list into its head element (FIRST) and tail list (REST).

4.1.2 TRALE

Figure 4.8 shows the beginning of the contents of a TRALE grammar’s `signature` file. It contains all types and features for that grammar. `bot` is the top type. The tree structure of the hierarchy is realised by indentation, i.e. in figure 4.8 `bot` is the immediate super-type

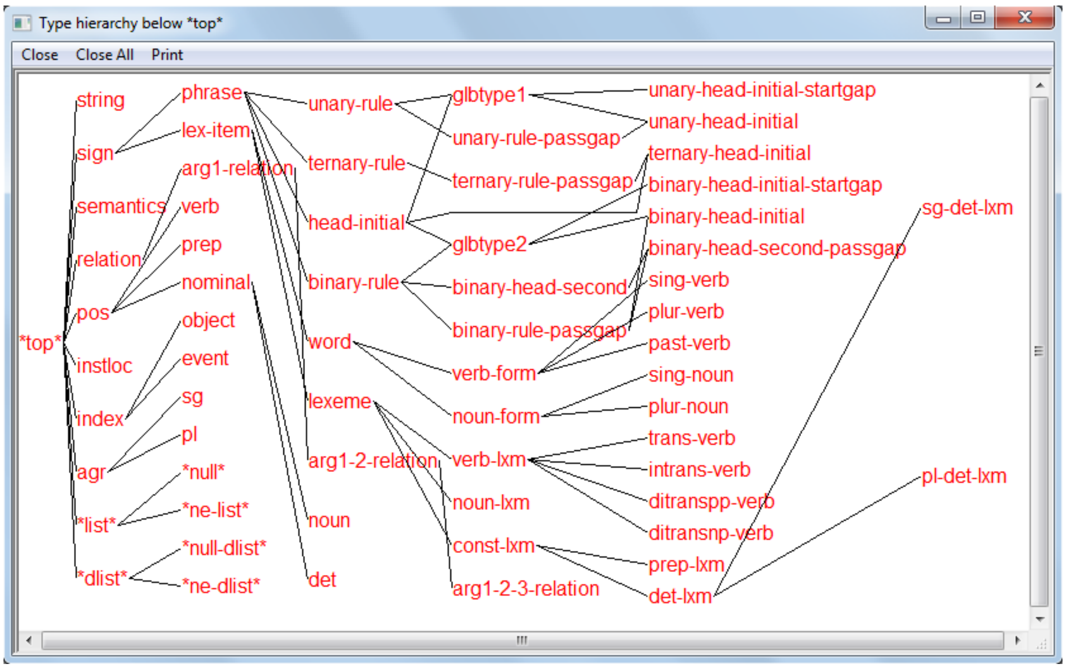
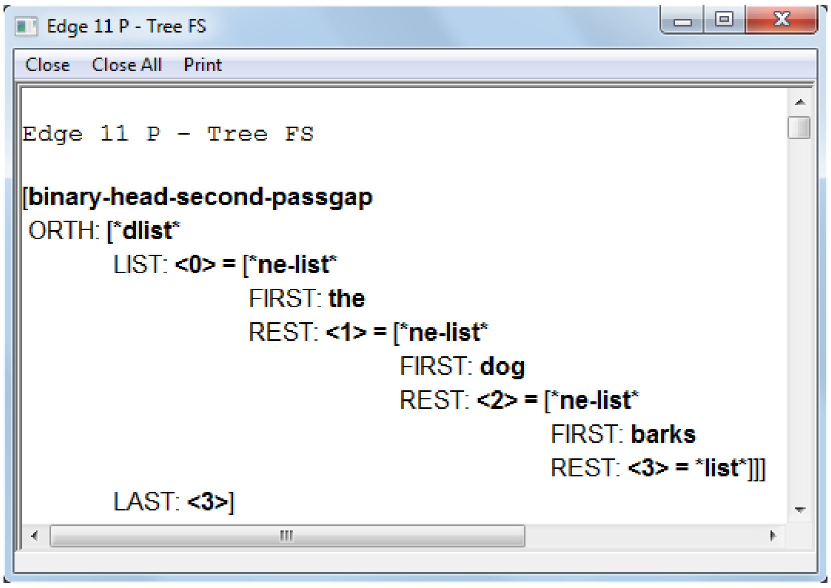


Figure 4.4: The LKB type hierarchy for the g8gap grammar [Copestake, 2002]



```
Edge 11 P - Tree FS

[binary-head-second-passgap
 ORTH: [*dlist*
        LIST: <0> = [*ne-list*
                     FIRST: the
                     REST: <1> = [*ne-list*
                                  FIRST: dog
                                  REST: <2> = [*ne-list*
                                               FIRST: barks
                                               REST: <3> = *list*]]]
        LAST: <3>]
```

Figure 4.5: The beginning of an LKB feature structure for “The dog barks”

```
$ \avmplus{\att{binary-head-second-passgap}\\
        \attval{ORTH}{\avmplus{\att{*dlist*}\\
                \attval{LIST}{\ind{0}} \avmplus{\att{*ne-list*}\\
                                   \attvaltyp{FIRST}{the}\\
                                   \attval{REST}{\ind{1}} \avmplus{\att{*ne-list*}\\
                                                      \attvaltyp{FIRST}{dog}\\
                                                      \attval{REST}{\ind{2}}
  \avmplus{\att{*ne-list*}\\
          \attvaltyp{FIRST}{BARKS}\\
          \attval{REST}{\ind{3}}\ \ \myvaluebold{*LIST*}}}}\\
                \attval{LAST}{\ind{3}}}}\\
  ...
```

Figure 4.6: An excerpt of LKB's TEX output for the parse of "The dog barks"

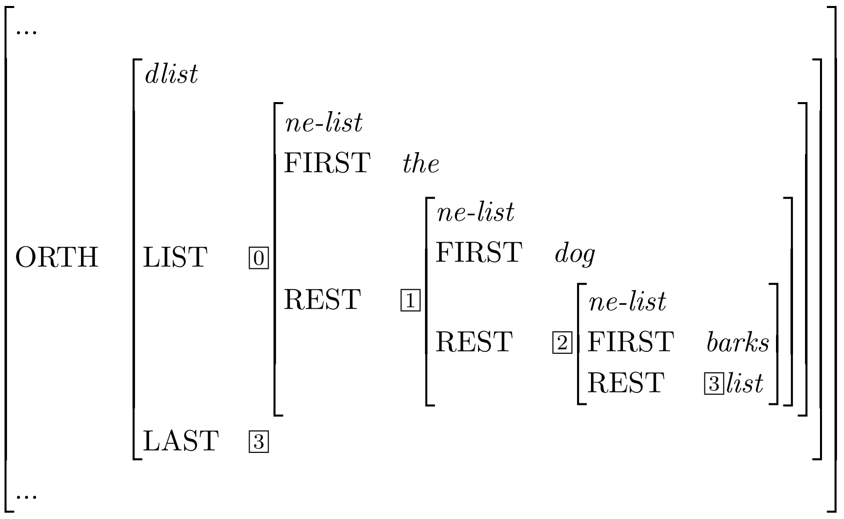


Figure 4.7: A part of an AVM produced by LKB, the rendering of figure 4.6's TEX code. Nested AVMs of type *dlist* or *ne-list* represent a list.

```
type_hierarchy
bot
  nonloc rel:list slash:list
  none_or_sign
    sign phon:list loc:local nonloc:nonloc v2:bool trace:minus_or_extraction_or_vm lex:bool max_:bool
      word
        spr_saturated_word
          glbtype14
            complementizer_word
            preposition_word
              mod_preposition
                noun_mod_preposition
                  location_noun_mod_prep
              comp_preposition
            verb_word
              trans_verb
                strict_trans_verb
                ditrans_verb
              subjlos_verb
              intrans_verb
                strict_intrans_verb
                np_pp_verb
  per
    second_or_third
      third
      second
    first_or_third
      &third
      first
...
```

Figure 4.8: An excerpt of a TRALE type hierarchy

of `nonloc`, `none_or_sign` and `per`, `none_or_sign` is the immediate super-type of `sign`, etc. A type's appropriate features are listed in the same line. Here `nonloc` has two appropriate, REL and SLASH, which both expect a value of type `list`.

The parsing of such data is then straightforward. At every time our TRALE type hierarchy parser saves the type name last seen for each indentation and declares it to be the immediate super-type of all types found with one more indentation. We may also encounter the same type more than once. Every non-initial occurence of a type is marked with a & indicates that the same type has more than one immediate super-type. In figure 4.8, for example, `third` and `&third` are parsed to the same `Type`, inheriting both from `second_or_third` and `first_or_third`.

To come by model data from TRALE, we go through a similar process as in section 4.1.1. We open the TRALE parser which loads a grammar. In our following example this is the grammar that goes with chapter 10 of Müller [2007b]. Upon entering a well-formed sentence into the prompt we get to see two windows: the chart parse (figure 4.9) and the resultung AVM (figure 4.10).

The latter window allows us to save the resulting data to a file (see figure 4.11). This now serves as the input for our `TraleModelParser`. It again tokenises the text into its own token objects by virtue of regular expressions. Then a construct tree is created for every

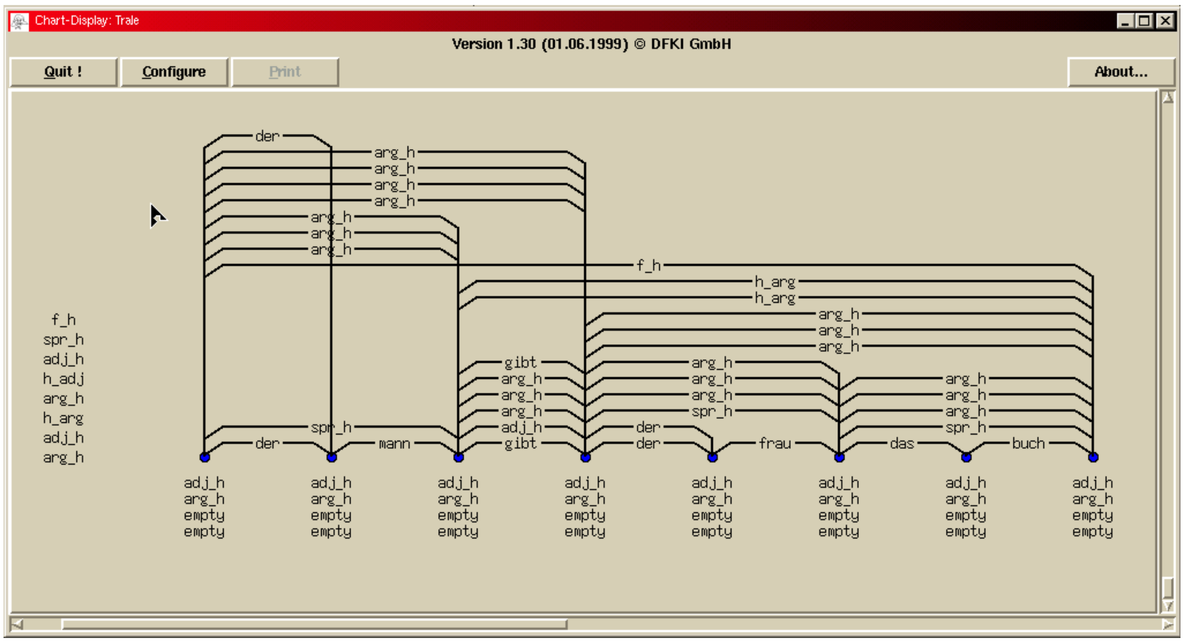
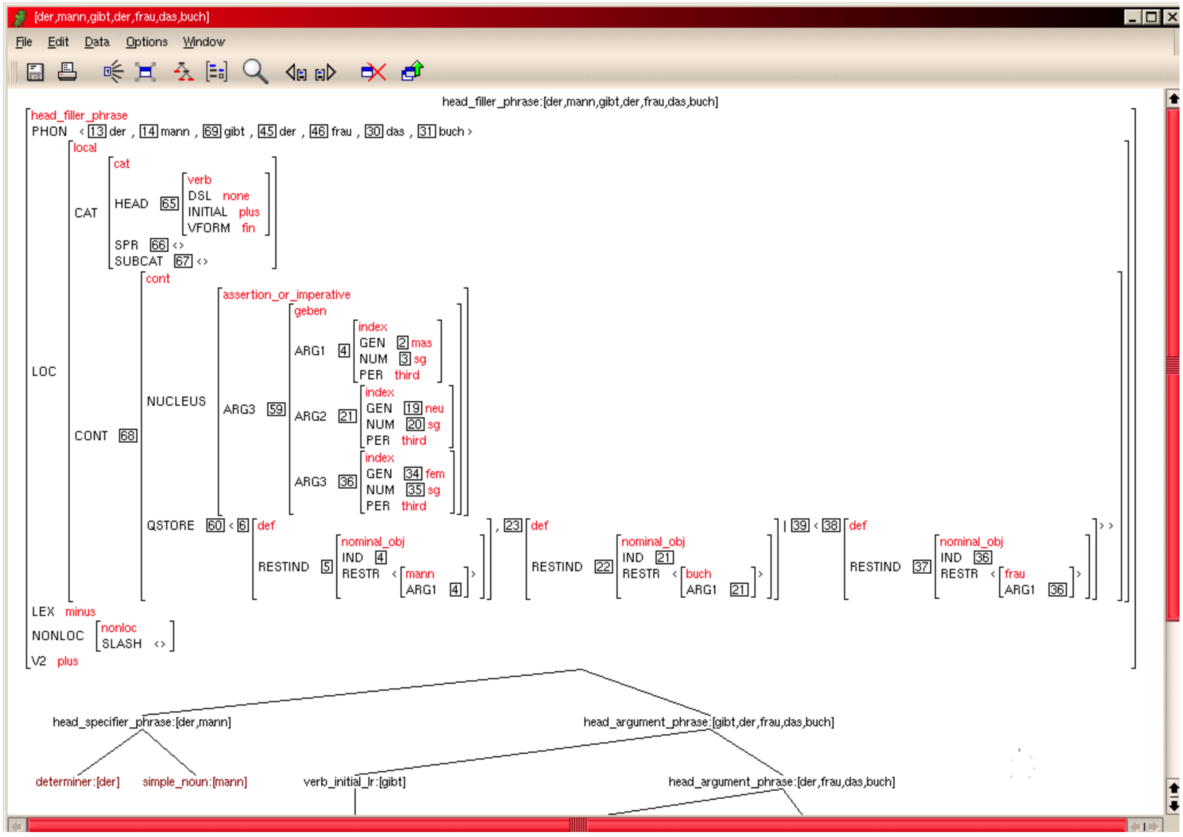


Figure 4.9: TRALE Chart Parse of "Der Mann gibt der Frau das Buch"


[der,mann,gibt,der,frau,das,buch]
File Edit Data Options Window

```
head_filler_phrase:[der,mann,gibt,der,frau,das,buch]
head_filler_phrase
PHON  < 13 der , 14 mann , 69 gibt , 45 der , 46 frau , 30 das , 31 buch >
LOC   local
      CAT   cat
            HEAD 65 verb
                    DSL     none
                    INITIAL plus
                    VFORM   fin
            SPR 66 <>
            SUBCAT 67 <>
      CONT 68 cont
            NUCLEUS assertion_or_imperative
                    ARG3 59 geben
                            ARG1 4  index
                                    GEN 2 mas
                                    NUM 3 sg
                                    PER third
                            ARG2 21 index
                                    GEN 19 neu
                                    NUM 20 sg
                                    PER third
                            ARG3 36 index
                                    GEN 34 fem
                                    NUM 35 sg
                                    PER third
            QSTORE 60 < 6 def
                          RESTIND 5 nominal_obj
                                    IND 4
                                    RESTR < mann
                                            ARG1 4 >
                        , 23 def
                          RESTIND 22 nominal_obj
                                     IND 21
                                     RESTR < buch
                                             ARG1 21 >
                        | 39 < 38 def
                               RESTIND 37 nominal_obj
                                          IND 36
                                          RESTR < frau
                                                  ARG1 36 > > >
LEX   minus
NONLOC nonloc
       SLASH <>
V2    plus

head_specifier_phrase:[der,mann]    head_argument_phrase:[gibt,der,frau,das,buch]
determiner:[der]   simple_noun:[mann]   verb_initial_lr:[gibt]   head_argument_phrase:[der,frau,das,buch]
```

Figure 4.10: TRALE parse of "Der Mann gibt der Frau das Buch" with the root note expanded

```
!newdata "[der,mann,gibt,der,frau,das,buch]" (S0(1"head_filler_phrase")(V2"phon"(L3(#4 13)
(#6 14)(#8 69)(#10 45)(#12 46)(#14 30)(#16 31)))(V17"loc"(S18(19"local")(V20"cat"(S21
(22"cat")(V23"head"(#24 65))(V25"spr"(#26 66))(V27"subcat"(#28 67))))(V29"cont"(#30 68))))
(V+31"dtrs"(L32(#33 71)(#35 70)))(V+36"head_dtr"(#37 70))(V38"lex"(S39(40"minus")))
(V+41"max_"(S+42(+43"bool")))(V+44"non_head_dtrs"(L45(#46 71)))(V47"nonloc"(S48(49"nonloc")
(V+50"rel"(S+51(+52"list")))(V53"slash"(L54))))(V+55"trace"(S56 (57"minus")))(V58"v2"(S59
(60"plus"))))(R61 69(A62"gibt"))(R63 68(S64(65"cont")(V66"nucleus"(S67(68"assertion_or_imperative")
(V69"arg3"(#70 59))))(V71"qstore"(#72 60))))(R73 67(L74))(R75 66(L76))...
```

Figure 4.11: An excerpt of a text file containing the TRALE parse of "Der Mann gibt der Frau das Buch"

entry in the file. Every entry represents an HPSG reference (tag) and its tree-formed sub-structure of the entire AVM, down to all subsequent tags in that sub-structure. When creating a `KripkeStructure` from these construct trees, all sub-structures are merged into a single structure according to their references.

Since TRALE allows cycles in its models, the recursive descent into a model avoids getting caught in loops by remembering which nodes have already been visited. When a loop is found the edge responsible for closing the cycle gets saved in a global array for later and that path is not pursued further. Afterwards those critical edges are then inserted into the grand structure. This can appear inside a nominal phrase, for example, when a determiner contains a reference to its noun and vice versa.

Even though the resulting `KripkeStructure` is a translation of TRALE's output, it is not necessarily a HPSG model in the strict sence. That is, there might be states containing types which are not maximally specific, i.e. leaf types. This can happen because TRALE's search of a grammar-fulfilling construct only specifies types as far as is necessary.[2]

4.2 Formula Parser

$PPDL_2$ formulas are entered by hand in a textfield provided by the GUI (see figure 6.1). It allows UNI-code characters, but as it would be inconvenient for the user to enter UNI-code characters such as $\epsilon, \cap$ etc., buttons are provided for this purpose. In fact, all characters (and strings) constituting operators and junctors are covered by these buttons. A `CaretListener` keeps track of the cursor's position in the formula textfield so that the symbol associated with the clicked button is inserted there.

In addition to these expressions, the user may include commentaries in the formula textfield. Whenever '//' appears in the text, the rest of that line is considered a commentary by the formula parser. Any tab-space, normal space or line break is also ignored by the formula parser. Nevertheless the raw text, i.e. as it was entered, is saved separately in the formula object. This is for reasons of convenience. A user entering a fairly complex formla will appreciate the possibility to divide it into 'blocks' so as to visually stay in control (e.g. our presentation of formulas in chapter 1). Once the formula is parsed and again rendered to a linear and compressed string (no spaces or line breaks), it can be very hard to read and make sence of. Thus, each formula has an attribute `humanText` for saving the originally entered text, which is again shown in the formula textfield whenever this formula is selected in the GUI.

[2]Frank Richter made us aware of this point.

When the user finishes entering a formula, she hands it over to the parser, i.e. a `FormulaParser`-object receives the entered `String` and does the usual three steps: tokenise, create a `Construct` tree, create a `Formula` from the `Construct` tree.

This parser also only needs a linear sweep of the token sequence because it is predictive and in a sence. The operators that appear in formulas come in three different categories: prefix (e.g. app, elem, $\ominus$), infix (e.g. $\wedge, \cap, \sqcap$), and postfix (e.g. *). Each creates a new construct in the tree being built. Each construct saves a pointer to its parent construct and retains a list of children contructs. The "character" of a construct is fixed by internally saving the determining token as `characteristicToken`. With infix and postfix operators at least one child argument will be encountered before the characteristic operator token comes. The parser does not mind this. It just takes the children and the operator as they come.

The reader will have noticed that the syntax of formulas and programs is strict as pertaining to round brackets. It demands them, even where they are normally omitted because there is no human fear of ambiguity (e.g. elem$((\mathrm{F}_1 \cap F_2))$ must have both pairs of round brackets, the inner one for the intersection and the outer one for the elem program). In the present program this convenience is sacrificed for determinism in parsing. Thus, it suffices to linearly go through the sequence of tokens once. The * operator, since it is a postfix operator, needs special treatment. When a construct receives such a token, it removes its last child from the children list, attaches a new child construct in its place with the star token as its `characteristicToken`, and reattaches the remooved child to the new construct's children list.

Diamond and box formulas are another interesting case. Since they can have an arbitrary amount of sub-formulas, a problem arises when a modality construct has 0 children. Take $\langle\alpha\rangle()$, for example. After the program α has been fully received and added to the pertinent child list, a closed angled bracket comes next and becomes this construct's `characteristicToken`. After this an open round bracket is received. Since a child is expected to come after the open round bracket, an unspecified construct is created, as usual, and appended to the children list. Next, the tokens constituting that child should come, but instead a closed round bracket is encountered. So, for this special occasion, an if-clause takes care of the 'wrong expectation'. The modality construct is relieved of its last child and gets closed.

Formulas such as $(a \rightarrow b \leftrightarrow c)$ are ambiguous and should either be $((a \rightarrow b) \leftrightarrow c)$ or $(a \rightarrow (b \leftrightarrow c))$. This ambiguity is caught by the parser when it attempts to save the received token as the current construct's `characteristicToken`. It first checks if this construct already has a characteristic token. This case is only allowed if both tokens come from the same operator and if that operator is associative. That is, formulas like $(a \cap b \cap c)$, $(a \vee b \vee c)$, $(a; b; c)$ etc. are allowed. In any other case, e.g. $(a \wedge b \vee c)$ or $(a \rightarrow b \rightarrow c)$, an `Exception` is thrown that explains which construct contains which operator ambiguity.

After all information has been stored up in a tree of `Construct` objects that tree is transformed into a `Formula` bottom-up. Tokens that constitute strings which are not operators must either be types or features. Which is which is infered from context. This context depends upon each construct's character and is respectively provided in a list of information about the kind of expected children. If a construct's children are of the wrong type or amount an `Exception` informs the used about this.

Another thing that needs to be inferred is whether a composition is atomic or not (i.e. $(\gamma; f)$ or $(\alpha; f)$ from definition 4). The same goes for union programs: Are they atomic or not (i.e. $(\beta_1 \cup \beta_2)$ or $(\alpha_1 \cup \alpha_2)$ from definition 4)? Since we are building these programs and

```
public void write(Object o, String filename) throws Exception{
  if(!filename.endsWith(".xml")){
    filename += ".xml";
  }
  XMLEncoder encoder =
     new XMLEncoder(
        new BufferedOutputStream(
          new FileOutputStream(filename)));
  encoder.writeObject(o);
  encoder.close();
}
```

Figure 4.12: Writing a Java object `o` to an XML file `fileName`

```
public Object read(String filename) throws Exception {
    XMLDecoder decoder =
        new XMLDecoder(new BufferedInputStream(
            new FileInputStream(filename)));
    Object o = decoder.readObject();
    decoder.close();
    return o;
}
```

Figure 4.13: Creating an object `o` from an XML file `fileName`

formulas bottom-up we determine what arguments are contained in compsition and union programs before this question of atomicity is addressed. Once the children are known, the parser choses the non-atomic versions if it must, and it choses the atomic versions in any other case.

When constructs which are made from an associative operator are transformed into their respective programs and formulas the amount of children they collected is important. If they have two children their transformation is straightforward. If, however, they have more than two children their list of children is recursively "left-folded", i.e. the first two children are transformed to a program or formula, which is then merged with the next child and so forth.

4.3 XML Serialisation of Java objects

Our standard file format for saving all main components is XML. This is very convenient because Java has a native serialiser for writing objects to XML and reading them again. The only code necessary for writing (or xml-encoding) is found in figure 4.12.

Any Java object `o` can be handed to this `write`-method, along with the path `fileName` of the XML file that it is to be written to. In order to reinstate the XML data from a file to a Java object, we use the `read`-method found in figure 4.13.

Here the path of the XML file to be read is handed to the method. It xml-decodes the data therein and creates the object `o` from it. Event hough the type of the originally written object may be more specific than `Object`, the `XMLDecoder` only creates an object of type `Object`. Whatever method called the `read`-method can then cast the returned object to

```
  private Map<String, Type> typeMap;
  private TreeSet<String> featureSet;
  private Type topType;


  public TypeHierarchy(){

  }


  public Map<String, Type> getTypeMap() {
    return typeMap;
  }
  public void setTypeMap(Map<String, Type> typeMap) {
    this.typeMap = typeMap;
  }
  public TreeSet<String> getFeatureSet() {
    return featureSet;
  }
  public void setFeatureSet(TreeSet<String> featureSet) {
    this.featureSet = featureSet;
  }
  public Type getTopType() {
    return topType;
  }
  public void setTopType(Type topType) {
    this.topType = topType;
  }
...
```

Figure 4.14: An excerpt from the code in class `TypeHierarchy` demonstrating the Java convention for getters and setters

whatever type it needs. Thus, if the file of a serialised object of an unexpected type is read, a `ClassCastException` will occur.

An essential prerequisite for Java's XML serialiser is adherence to the Java convention of providing every involved class with getters and setters. For us, the involved classes are `KripkeStructure`, `TypeHierarchy` and `Theory`, but by extension all objects that they hold internally must also keep to the convention, i.e. the classes `Relation`, `Type`, `Formula`, etc. Getters and setters are methods with a prescribed format. Figure 4.14 shows an excerpt of the code in class `TypeHierarchy`.

All class attributes must be declared `private`. For each attribute `T x` there must be a getter method `getX` returning `x`, and a setter method `setX` taking an object of type `T` with a void return. Another requirement for Java's XML serialiser is that there be an empty constructor, as seen here. Another constructor not shown here is one that we use when instantiating a `TypeHierarchy`-object by virtue of a parser, for example. It is responsible for initialising all class attributes. The `XMLDecoder`, on the other hand, takes care of initialising all class attributes, and therefore is content with an empty constructor.

Figure 4.15 shows an excerpt from an XML file created by an `XMLDecoder` by serialising a `TypeHierarchy`-object. Refer to section 3.1 for a reminder of the class attributes of `Type` and `TypeHierarchy`. The `XMLDecoder` then learns from the row `<void property="topType">`

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_24" class="java.beans.XMLDecoder">
 <object class="hpsg.TypeHierarchy">
  ...
  <void property="featureSet">
   ...
  </void>
  <void property="topType">
   <object id="Type0" class="hpsg.Type">
    <void property="featureMap">
     <object class="java.util.HashMap"/>
    </void>
    <void property="istSet">
     <object class="java.util.HashSet"/>
    </void>
    <void property="typeName">
     <string>top</string>
    </void>
   </object>
  </void>
  <void property="typeMap">
   ...
   <void method="put">
    <string>top</string>
    <object idref="Type0"/>
   </void>
  </void>
 </object>
</java>
```

Figure 4.15: An excerpt of an XML file containing a serialised `TypeHierarchy`-object

that the `TypeHierarchy`-object has a property `topType` and instantiates a new `Type` for this via the data in the subsequent 11 rows: The topType gets an empty `HashMap` as `featureMap`, an empty `HashSet` as `istSet`, and the `String` "top" as `typeName`.

An important feature of xml-decoding can be seen in this example. When, in a lower row, the top type is put into the type hierarchy's `typeMap`, the `String` "top" is used as its key. The object that this key points to should of course be the `Type` instantiated earlier. In order not to create another object with the same attributes at this point, the value object to be put into the `typeMap` is found by reference. That is, when the top type occurs for the first time in `<object id="Type0" class="hpsg.Type">` it receives an `id` made from its Java type and a counter number. When this object appears again, the `XMLDecoder` makes sure that it uses the already instantiated object by looking it up with the same `id` in row `<object idref="Type0"/>`.

Chapter 5

Graph Visualisation

This chapter concerns the considerations that led to our choice and implementation of graph visualisation. Our goal was to show all aspects of a model's structure in the form of a network. These include

- **states** indicating their types
- directed **edges** indicating their feature
- ordered **lists** of states, including empty lists

When, for example, the sentence "Tim snores" and a few rudimentary features thereof are considered, we obtain an AVM such as that in the left half of figure 1.5. Visualising such a feature structure as a network led to the right half of the same figure.

5.1 Prefuse Graph Visualisation

After searching through the vast options for graph visualisation, we found prefuse, an INFORMATION VISUALIZATION TOOLKIT[1], to be a viable option. It comes with many comprehensive features. Among many others, customly labelling and coloring nodes is very useful. Directed edges with arrow heads are also possible. While that release did not allow the same so-called decoration of edges as it did with nodes, the code for such an extension was attainable from a forum entry[2].

Prefuse graphs represent Kripke structures. Each blue node contains a label made up of the unique number and type of the represented state. For every binary tuple in the visualised Kripke structure the visualisation draws a directed edge from its first to its second state. The green nodes visualise lists. In figure 1.5 the PHON feature of state 0 has a list value containing states 4 and 5, in that order. Instead of somehow forcing these nodes to be positioned close to each other, for each list we add a further node which points to its members. That node's label is `<x>`, where `x` is the length of the represented list. In order to identify the order of the list members, the outgoing list edges are numbered starting with 1. Empty lists are represented by nodes marked `<0>` which point to no other node.

The type of graph we use to visualise Kripke structures employs a force-based algorithm to determine the position of its nodes. Loosely, this can be thought of as an algorithm that

[1]We imported the prefuse-beta-20071021 package, which can be found at `http://prefuse.org/download/`
[2]`http://goosebumps4all.net/34all/bb/showthread.php?tid=19` retrieved on January 31, 2011

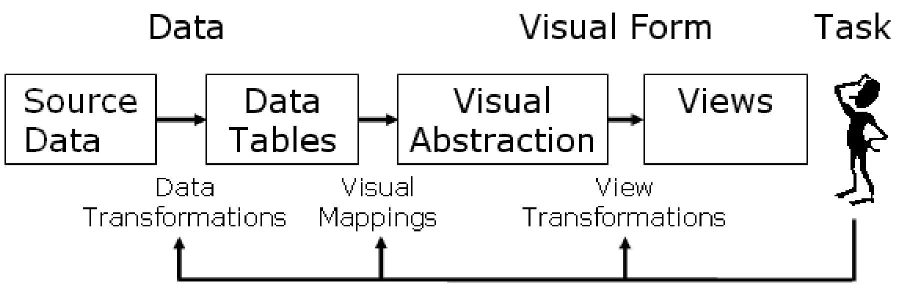


Figure 5.1: Information visualisation reference model

lets each node repell all other nodes, but at the same time being bound to nodes with a shared edge. The result is a decently viewable distribution of all nodes. This panel is provided by prefuse's libraries and will be a part of the Modelchecker's GUI.

5.2 Visualising Kripke Structures

In the prefuse documentation[3] we find a diagram that summarises the fundamental steps of information visualisation (see figure 5.1). This diagram depicts the information visualisation reference model, as defined in Card et al. [1999]. We will now take a look at how the steps in the visualisation pipeline are implemented in the context of our program.

5.2.1 Data Transformations

The data in its intial form is, of course, all the data contained in a Kripke structure $\langle S, \{R_f \mid f \in F, V\rangle$. This data gets transformed into prefuse tables. These tables are similar to relational data base tables and prefuse has its own query language to operate on them. Java class `KripkeStructure` is equipped with a method `toPrefuseGraph()` for creating a `Graph`-object from it. The following Java code shows how in a `Graph` each table's name and its elements' data types are defined.

```
Graph g = new Graph(true); //true -> directed graph

g.addColumn("state", State.class);
g.addColumn("edgeLabel", String.class);
g.addColumn("nodeLabel", String.class);
g.addColumn("isListNodeOrEdge", Boolean.class);
g.addColumn("isVisible", Boolean.class);
```

Obviously this table contains complementary columns, e.g. nodes have no use of the attribute `edgeLabel`, but there is no conflict in this, and all elements to be visualised can be filled into the same table. The second argument of the `addColumn`-method indicates the data type of the data in this column. Even though the labels of states are already covered in the `nodeLabel`-column, it is practical to save the actual `State`-object in the table for later use. The usefulness of the `isVisible`-attribute will become apparent in section 6.5.

[3]`http://prefuse.org/doc/`

List nodes make no use of this column. They and their outgoing edges have the `isListNodeOrEdge`-attribute set to `true` which marks them for their coloring. Note that nowhere in the `KripkeStructure` or its `TextHierarchy` is there a saved value indicating wether a feature points to a single state or a list of states. Therefore no difference is made between a value being a state or a singleton list. We see this in figure 1.5, where the SUBCAT edges starting in state 2 and state 14 each point to a list node, but the one starting in state 10 points directly to state 13, rather than to a list node that in turn points to that state.

5.2.2 Visual Mappings

A `Visualization`-object is responsible for governing all graphical data structures. In our setting it saves the `Graph` that was just outlined. It also saves a `RendererFactory` which is in charge of creating a `VisualItem` for each graph element. We attach the following two `LabelRenderer`s for labelling nodes and edges, respectively.

```
LabelRenderer nodeRenderer = new LabelRenderer("nodeLabel");
nodeRenderer.setRoundedCorner(8, 8);

LabelRenderer edgeRenderer = new LabelRenderer("edgeLabel");
edgeRenderer.setRoundedCorner(8, 8);
```

These two groups of elements are brought in connection with `ColorActions`. They are objects that specify various graphical options such as the color, thickness, shape, visibility and transparency of edges, edge label text, edge label background, arrows, nodes, node label text, node label background and potentially many more graphic items.

Instead of just specifying which group of `VisualItems` has which properties, the applicability of `ColorActions` can also be dependent of the elements' values. Here prefuse's query language comes into play. An example of this is the one node that is considered focused. Its internal `_highlight`-option is active, and its background is rendered red, due to (the last `add`-call of) the following `ColorAction`.

```
ColorAction nodeFill = new ColorAction(modelNodes,
          VisualItem.FILLCOLOR, ColorLib.rgb(200,200,255));
nodeFill.add("isListNodeOrEdge=true", ColorLib.rgb(150, 255, 150));
nodeFill.add("isVisible=FALSE", ColorLib.alpha(0));
nodeFill.add("_highlight AND isVisible=TRUE", ColorLib.rgb(255,200,125));
```

5.2.3 View Transformations

Once the graphic items are displayed they are by no means static. Many kinds of interaction are possible through mouse and keyboard input. They enable the user to drag the entire graph or single nodes (left mouse button: drag), to zoom (right mouse button: click or drag; mousewheel), focus on a node (left mouse button: click) and highlight a node and all adjacent nodes by color (mouse over).

Further interactions are provided by the other panels of the GUI (see section 6.4). That is, the `Visualization`-object can still be edited after it has been rendered. Its components have Java listeners attached to them. This is very useful, since we desire a way to change the underlying Kripke structure by interacting with its visualisation.

5.3 Visualising the Type Hierarchy

Not to forget, a project's type hierarchy is also a graph that ought to be visualised. Figure 1.6 shows the prefuse visualisation of a type hierarchy that comes with the models in section 1.2. Each type is represented by a node labelled with that type's name and all its features and all their value types. Instead of showing the hierarchy with its top type at the top it has a radial shape here. This is merely the result of the force-based positioning of nodes and has no further meaning.

Chapter 6

GUI

All of the functionality described in previous chapters is made applicable in our Java program using a graphical user interface (GUI).[1] We will now take a look at the panels and menu points involved.

6.1 Project Tree

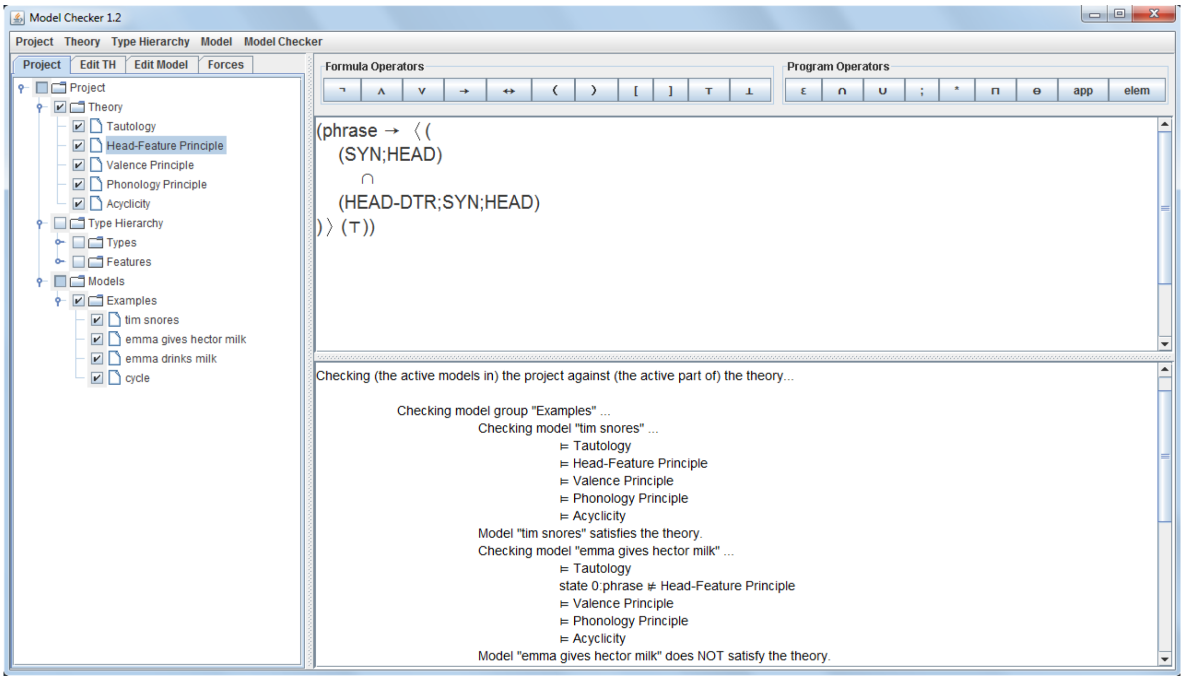
The project tree is to help the user navigate through all input material, i.e. all formulas, the type hierarchy, and all models (see figure 6.1. The root node “Project” and its immediate children are always present. They are initially empty, that is, there are no formulas under “Theory”, the type hierarchy has no types or features, and no model groups or models are found under the node entitled “Models”. Any material that is then imported or created by hand is to be found under one of these three top-level nodes. Almost all nodes may carry titles. These are often entered upon creation of the object contained thereby. They may also be edited at a later time via “Project → Rename Selected Item”. When a formula, model or model group is no longer needed, it may be deleted from the project and removed from the project tree via “Project → Delete Selected Item”.

The project tree makes use of the Checkboxtree package for Java[2] Bigagli and Boldrini [2007]. In addition to a title, every node in the tree has a checkbox next to it. A checked node is considered **active** in our program. When checking formulas against models, only active ones are considered. When displaying a model’s graph the user has the option of hiding certain states and edges. This is done by unchecking, i.e. deactivating, types and features which those states and edges are respectively labelled with.

The menu is sensitive to what kind of node is selected in the project tree. For example, a model group must be selected in order for the menupoint “Model → Add new Model” to be clickable. A new model will thus be added to that model group. Similarly, when a model is to be moved it must first be selected before “Model → Move Model” is clickable. Similarly, the menupoints “Model → Save Snapshot” and “Type Hierarchy → Save Snapshot” are only clickable whenever the intended object is selected in the project tree.

[1]The program is compiled to a `jar` file and should be able to run platform-independently. We only tested it under Microsoft Windows 7, though. Potentially other platforms may not agree with some UNI-code characters or the “Ctrl + left mouse button” combination mentioned in section 6.4.

[2]We used version 3.11 which can be downloaded at `http://ulisse.pin.unifi.it:8081/nexus/index.html#nexus-search;classname~checkboxtree`



```
(phrase → ⟨(
   (SYN;HEAD)
      ∩
   (HEAD-DTR;SYN;HEAD)
)⟩ (⊤))
```

```
Checking (the active models in) the project against (the active part of) the theory...

          Checking model group "Examples" ...
                    Checking model "tim snores" ...
                              ⊨ Tautology
                              ⊨ Head-Feature Principle
                              ⊨ Valence Principle
                              ⊨ Phonology Principle
                              ⊨ Acyclicity
                    Model "tim snores" satisfies the theory.
                    Checking model "emma gives hector milk" ...
                              ⊨ Tautology
                              state 0:phrase ⊭ Head-Feature Principle
                              ⊨ Valence Principle
                              ⊨ Phonology Principle
                              ⊨ Acyclicity
                    Model "emma gives hector milk" does NOT satisfy the theory.
```

Figure 6.1: The project tree (left) and the theory panel (right)

6.2 Import and Export

The user may import models (several at one time) and a type hierarchy from LKB or TRALE via the respective menu points under “Model” and “Type Hierarchy”. If she has finished viewing or editing them, she may export created or imported objects to an XML file in like manner. Instead of doing this for every object in the open project, the user may instead chose to save all of them as one project XML file via “Project → Save to XML”. Any XML file created in this way may at a later time be loaded via “Project → Load from XML”. Caution is advised whenever a project or type hierarchy is loaded, because they will take the place of the former ones.

6.3 Theory

The theory panel (see figure 6.1) is shown whenever the “Theory” node or any formula is selected. In the middle the user may enter a formula or edit a selected one. This textfield displays unicode characters, as several of the operators are. To make it easier to enter these operators, buttons are provided above. Each will insert into the textfield at the cursor’s position the text which it bears as caption. Event hough some of these symbols might be found on the user’s keyboard, they are included as a reminder and for completeness. To write a diamond formula’s brackets either the buttons may be used or the symbols for “less than” and “greater than” may be typed. One can include comments anywhere in the formula textfield by typing “//”. All symbols in the same row to the right of “//” will thus be ignored by the formula parser. Also, line breaks, tab spaces and normal spaces are allowed everywhere.

The lower textfield is read-only and is used as output for the model checker's checking and analysing process. Every such evoked process starts with a line of stars, so that they may be distinguished more easily. Instead of having to scroll through the myriad of output text, the user might prefer to have the output textfield cleared before every check or analysis. This is ensured by checking the menupoint checkbox "Model Checker → Always clear Output Textfield before Checking". Alternatively, the output textfield can be cleared anytime by chosing "Model Checker → Clear Output Textfield" from the menu.

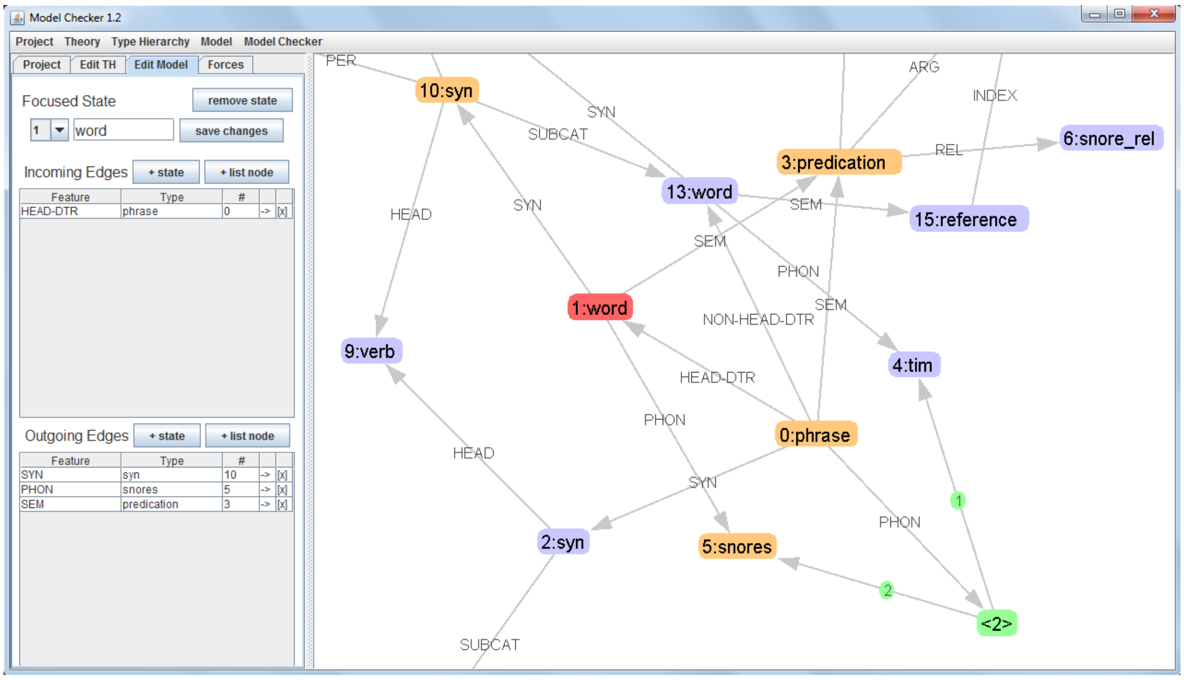
Having written a formula, the user must parse and save it before that information is lost due to selecting some other formula from the project tree. If a formula is currently selected and "Theory → Parse Formula and Save" is clicked, that formula will be overwritten by the parsed formula. By clicking "Theory → Parse Formula and Save As..." the newly parsed formula is instead appended to the theory with a new name that is entered via a message box. In any case, the text in the textfield must first be successfully parsed. If this is not the case, the parser attempts to help the user via a message box containing a rudimentary feedback on the first error found in the parse process.

6.4 Models

All models are organised into model groups using the first three menupoints under "Model". When a model is selected in the project tree its graph representation is shown in the main panel to the right. This display is newly generated for every newly selected model, that is all states originate in the panels's upper left corner and jiggle around until the force-driven algorithm has found a somewhat decent position for all states. The prefuse package provides a panel for manipulating several properties involved in this algorithm. The value sliders linked to those properties both for the currently selected model and for the type hierarchy are found under the tab "Forces". The most useful property is probably DefaultSpringLength, i.e. the length of all existing edges. Other viewing properties may be changed applying the mouse buttons and the mouse wheel to the graph panel.

An important interaction with the graph is focusing a state (thus marking it red). This is done by clicking on it with the left mouse button or by navigating to it using the panel under the tab "Edit Model" (see figure 6.2). Besides navigation, this panel also provides ways to edit all parts of the current model. The focused state may be removed, thus also removing all incident edges. Its containing type may be changed in the top textfield, subsequently clicking "`save changes`". Underneath there are two tables listing all incoming and outgoing edges, respectively. They contain all information about the focused state's incident edges: the feature of each edge and the type and number of the state on the other end of it. The former two may be edited directly in the table by selecting the prudent cell and changing its content. Clicking on "`->`" will focus the state adjacent via that row's edge. When "`[x]`" is clicked, that row's edge is removed from the graph. A new incoming or outgoing state or list node may be added to the focused node using the respective button. For linking two existing nodes in the graph, another methos is employed: the source node must be focused, and the target node must be clicked with the left mouse button while holding the Ctrl-button. This works for any pair of existing nodes, i.e. state nodes and list nodes.

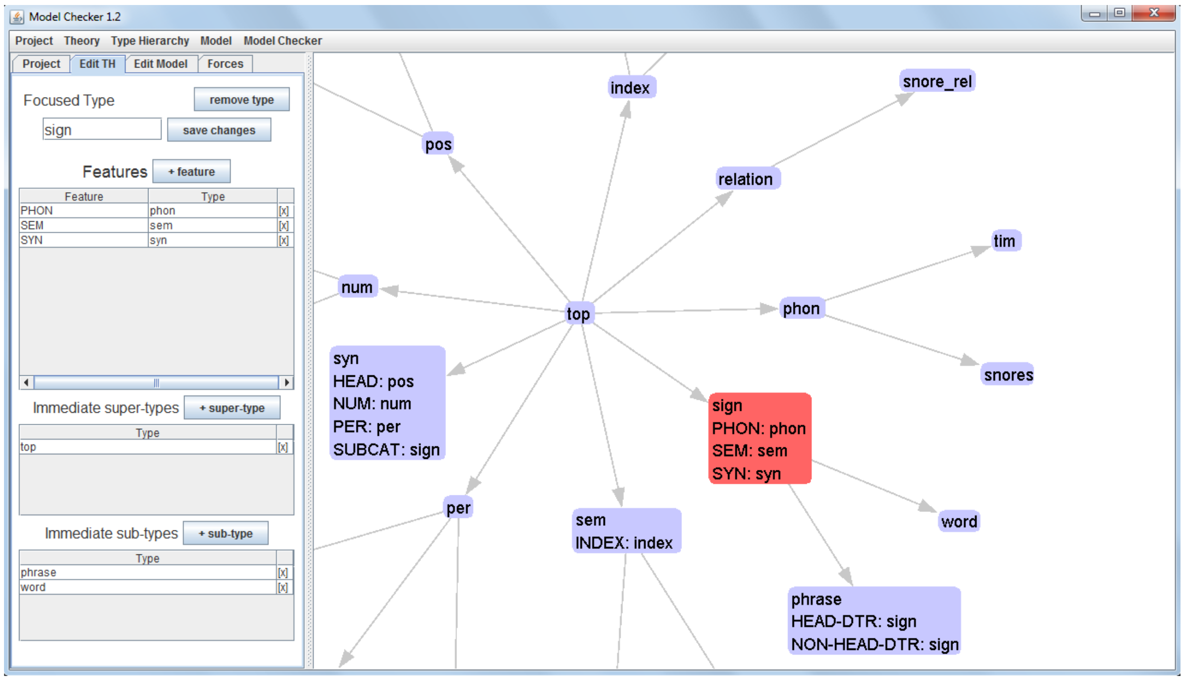
After finishing any manipulation of the current model graph, it may be saved by clicking the menupoint "Model → Check Model and Save (As...)" (same behaviour as with "Theory → Parse Formula and Save (As...)"). The program then checks whether the model is well-



Feature	Type	#		
HEAD-DTR	phrase	0	->	[x]

Feature	Type	#		
SYN	syn	10	->	[x]
PHON	snores	5	->	[x]
SEM	predication	3	->	[x]

Figure 6.2: The panel for editing models (left) and the currently visualised model being edited (right). Green nodes are list nodes captioned with length of the represented list, all other nodes are states. The red node is currently focused. All neighbouring states are highlighted orange for convenience.



Focused Type: sign

Feature	Type	
PHON	phon	[x]
SEM	sem	[x]
SYN	syn	[x]

Immediate super-types

Type	
top	[x]

Immediate sub-types

Type	
phrase	[x]
word	[x]

Figure 6.3: The panel for editing the type hierarchy (left) and the visualised type hierarchy (right). The red type is the focused type.

formed. That is, no two list nodes may be adjacent, all n list edges coming from the same list node must be numbered from 1 to n without duplication or gaps. If so, it is saved in its internal format.

6.5 Type Hierarchy

When the "Type Hierarchy" node is selected from the project tree, the graph representing the project's type hierarchy is shown in the main panel to the right (see figure 6.3). Each node contains a type (lower case), and underneath a list of all features (upper case) declared to be appropriate for that type. Each feature's value type is to the right of the feature. If the user wishes, shes may hide all information about features by unchecking the menupoint check-box "Type Hierarchy → Feature Visibility". The edges represent inheritance relationships, pointing from the more general to the more specific type.

Editing the type hierarchy happens in a very similar fashion as wth models. On the panel under the tab "Edit TH" the focused type can be removed or edited. Its features are listed in a table where they, too, can be removed or edited. A new feature is added to a type via the button "`+ feature`". The user must then select an already existing type from the ensuing message box as its value type. All features must be distinct. In two further tables the focused type's immediate super-types and sub-types are listed. Each row mentions an incoming or outgoing edge, respectively. Selecting the cell containing the adjacent type navigates to that type. Clicking "`[x]`" in the second column deletes the intended edge. New edges may be added using the buttons "`+ super-type`" and "`+ sub-type`" respectively. They each introduce a new type node to the graph. If instead two existing type nodes are to be linked by a new edge, the source node must be focused and the target must be clicked with the left mouse button while holding the Ctrl-button.

Here, too, the type hierarchy must be saved so as to internalise any changes and let them participate in the model checking process. This is achieved by selecting "Type Hierarchy $\rightarrow$ Check Type Hierarchy and Save" from the menu, if the graph obeys the following restrictions: There are no cycles, the graph has exactly one component (i.e. it is not empty and all nodes are connected), and there is exactly one type with no immediate super-type.

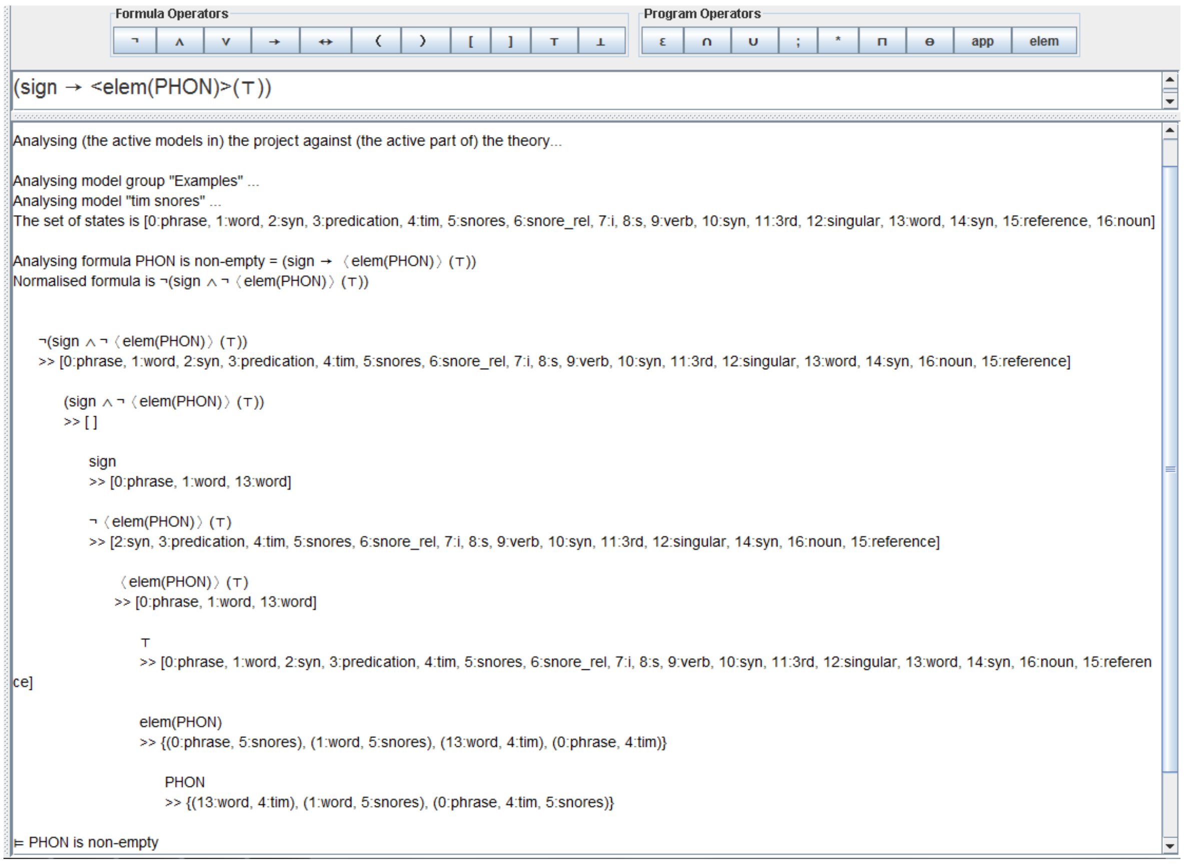
6.6 Model Checker

After all input materials have been prepared the user checks all formulas and models that are to be checked. She then selects "Model Checker $\rightarrow$ Check" from the menu. The checking algorithm's results are summarised in the output textfield (see figure []). All checked models are listed according to their model groups. For each there is one line per active formula ϕ indicating whether this model satisfies ϕ (output: "$\models \phi$") or not. In the latter case, the first state that does not fulfill ϕ is identified (e.g. output: "state 0:phrase $\not\models \phi$"). In figure 6.1 we see that the models for "tim snores" and "emma gives hector milk" (see section 1.1) satisfy all formulas mentioned in chapter 1, except in the case of "emma gives hector milk" and the head-feature principle. This is because we did not include more syntactic information (like its HEAD values) than was necessary in section 1.2 to demonstrate the phonology and subcategorisation principle.

If the user wishes to obtain more detailed information about the interaction of models and formulas, she may instead invoke the analysing process under "Model Checker $\rightarrow$ Analyse". Here, the same textfield is used and all combinations of active models and formulas are checked. But this time, we get to see the step-by-step evaluation of each single check. Figure 6.4 shows a small example: the formula $(\mathit{sign} \rightarrow \langle \text{elem(PHON)} \rangle (\top))$ describing each sign in the model to have a non-empty list of PHON values.

After the formula is normalised to $\neg(\mathit{sign} \wedge \neg\langle \text{elem(PHON)} \rangle (\top))$ it is analysed as a tree bottom-up. That is, when we read the evaluation from bottom to top, we start with the atomic programs and formulas which are indented the most, and successively rise all the way to the root, i.e. the entire formula, seeing more and more complexity and less and less indentation. Each sub-program or sub-formula is displayed with its semantics right underneath, indicated by ">>". Sub-formulas come with a relation (enclosed in curly brackets), i.e. a list of tuples (enclosed in round brackets). The semantics of sub-formulas is given by a list of all states which fulfill it (enclosed in square brackets).

Before the analysis, all states in the model are listed. If the formula should not be satisfied, all non-fulfilling states will be listed afterwards, so that the user can decide for herself where and how far this formula is off from being satisfied. Since a lot of text is usually created in this process, the user is advised to activate only one model and one formula for the detailed analysis.



```
(sign → <elem(PHON)>(⊤))
```

```
Analysing (the active models in) the project against (the active part of) the theory...

Analysing model group "Examples" ...
Analysing model "tim snores" ...
The set of states is [0:phrase, 1:word, 2:syn, 3:predication, 4:tim, 5:snores, 6:snore_rel, 7:i, 8:s, 9:verb, 10:syn, 11:3rd, 12:singular, 13:word, 14:syn, 15:reference, 16:noun]

Analysing formula PHON is non-empty = (sign → ⟨elem(PHON)⟩ (⊤))
Normalised formula is ¬(sign ∧ ¬ ⟨elem(PHON)⟩ (⊤))


   ¬(sign ∧ ¬ ⟨elem(PHON)⟩ (⊤))
   >> [0:phrase, 1:word, 2:syn, 3:predication, 4:tim, 5:snores, 6:snore_rel, 7:i, 8:s, 9:verb, 10:syn, 11:3rd, 12:singular, 13:word, 14:syn, 16:noun, 15:reference]

      (sign ∧ ¬ ⟨elem(PHON)⟩ (⊤))
      >> [ ]

         sign
         >> [0:phrase, 1:word, 13:word]

         ¬ ⟨elem(PHON)⟩ (⊤)
         >> [2:syn, 3:predication, 4:tim, 5:snores, 6:snore_rel, 7:i, 8:s, 9:verb, 10:syn, 11:3rd, 12:singular, 14:syn, 16:noun, 15:reference]

            ⟨elem(PHON)⟩ (⊤)
            >> [0:phrase, 1:word, 13:word]

               ⊤
               >> [0:phrase, 1:word, 2:syn, 3:predication, 4:tim, 5:snores, 6:snore_rel, 7:i, 8:s, 9:verb, 10:syn, 11:3rd, 12:singular, 13:word, 14:syn, 16:noun, 15:referen
ce]

               elem(PHON)
               >> {(0:phrase, 5:snores), (1:word, 5:snores), (13:word, 4:tim), (0:phrase, 4:tim)}

                  PHON
                  >> {(13:word, 4:tim), (1:word, 5:snores), (0:phrase, 4:tim, 5:snores)}

⊨ PHON is non-empty
```

Figure 6.4: The output of analysing the formula $(\mathit{sign} \rightarrow \langle \text{elem(PHON)} \rangle (\top))$ against our model for "tim snores" from section 1.5

Chapter 7

Conclusions

The work presented here is not the first implementation of an HPSG model checker. Richter et al. [2002] have already presented a graphical tool, called Morph Moulder (MoMo), which is used in conjunction with TRALE for inputting TRALE's equivalents of models, type hierarchies and formulas that are then checked for satisfaction. The main difference is the employed logic: Relational speciate re-entrant logic (RSRL, Richter [2004]). RSRL is a very powerful but costly[1] logic, assumed to subsume PPDL_2 in expressivity.

Søgaard and Lange [2009] explain that PPDL_2 is not expressive enough to cover all grammar principles of the fragment of the English language found in Pollard and Sag [1994]. They also provide proofs for its main attractive feature, its effiency: Model checking is in P; to be more precise, it has an upper bound of $\mathcal{O}(|\phi^2| \times |S|^4)$, where ϕ is a PPDL_2 formula and S is the set of states in the model being checked. This makes the checking of derivation structures (from various sources) a tractable postprocessing step. Our Java implementation mirrors the algorithm described in Søgaard and Lange [2009] in principle, modulo some programming adaptions that is.

When testing the correctness of an HPSG parser, one may find a model checker useful to uncover errors in the parsing process. If, for example, one of the parser's principles causes a contradiction and ought to make any derivation impossible (perhaps unknown to the developer), but the parser nonetheless outputs supposedly fulfilling derivation structures, checking those structures against a theory that includes this principle will identify the parser to be faulty. Being able to see which formula is unsatisfied in which states helps the developer to focus in on the problem.[2]

Interaction with our model checker is made highly transparent due to visualisation of the material and a detailed step-by-step analysis of the model-theoretic evaluation of formulas. Being able to navigate large and complex models close up gives the user a good insight and understanding of the material's (local) structure. Visualising arbitrary lists is always difficult, but we hope that we found a reasonable solution to this problem.

We deliberately put very few restrictions on forming type hierarchies and Kripke structures. If the user wishes to enforce various appropriateness constraints on the models, she is free to do so via PPDL_2 formulas. Examples for this can also be found in Søgaard and Lange [2009]. A high level of compatibility was in our interest. This interest was also followed when we implemented an interface with LKB and TRALE. Even though our tool lets you visualise

[1]Kepser et al. [2001] show the general model checking problem to be undecidable for RSRL.

[2]We thank Frank Richter for pointing out this idea.

their outputs, the interpretation thereof is always up to its creator. Dependency grammar is another area that can be explored using our model checker. Providing interfaces with dependency treebanks and HPSG treebanks could be part of a viable future development.[3]

Acknowledgements
I would like to thank Manfred Schmidt-Schauß, Gert Webelhuth, Anders Søgaard and Frank Richter for their helpful input and discussions.

Declaration of Authorship
I hereby declare that the work presented in this thesis is my own. Where I have consulted the work of others, this is always clearly indicated.

[3]Anders Soegaard is to be thanked for stressing this connection.

Bibliography

L. Bigagli and E. Boldrini. Swing-based tree layouts with checkboxtree, 2007. URL `http://www.javaworld.com/javaworld/jw-09-2007/jw-09-checkboxtree.html`.

P. Blackburn and J. van Benthem. Modal Logic: A Semantic Perspective. *ETHICS*, 98: 501–517, 1988.

S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in information visualization: using vision to think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-533-9.

B. Carpenter and G. Penn. ALE 3.2 User's Guide. Technical Memo CMU-LTI-99-MEMO, Carnegie Mellon Language Technologies Institute, 1999.

A. Copestake. *Implementing Typed Feature Structure Grammars*. CSLI Publications, Stanford, CA, 2002.

S. Kepser, M. Sailer, and G. Penn. On the Complexity of RSRL. In *Proceedings FG-MOL 2001, volume 53 of ENTCS*. Kluwer, 2001.

S. Müller. The Grammix CD Rom. A Software Collection for Developing Typed Feature Structure Grammars. In T. H. King and E. M. Bender, editors, *Grammar Engineering across Frameworks 2007*, Studies in Computational Linguistics ONLINE, pages 259–266. CSLI Publications, Stanford, 2007a. URL `http://hpsg.fu-berlin.de/~stefan/Pub/grammix.html`.

S. Müller. *Head-Driven Phrase Structure Grammar: Eine Einführung*. Number 17 in Stauffenburg Einführungen. Stauffenburg Verlag, Tübingen, 1 edition, 2007b. URL `http://hpsg.fu-berlin.de/~stefan/Pub/hpsg-lehrbuch.html`.

C. Pollard and I. A. Sag. *Head-Driven Phrase Structure Grammar*. The University of Chicago Press, Chicago, 1994.

F. Richter. *A Mathematical Formalism for Linguistic Theories with an Application in Head-Driven Phrase Structure Grammar*. Phil. dissertation (2000), Eberhard-Karls-Universität Tübingen, 2004.

F. Richter, E. Ovchinnikova, B. Trawiński, and W. D. Meurers. Interactive Graphical Software for Teaching the Formal Foundations of Head-Driven Phrase Structure Grammar. In G. Jäger, P. Monachesi, G. Penn, and S. Wintner, editors, *Proceedings of Formal Grammar 2002*, pages 137–148, 2002.

A. Søgaard and M. Lange. Polyadic Dynamic Logics for HPSG Parsing. *Journal of Logic, Language and Information*, 18(2):159–198, 2009.