HDT-UD: A very large Universal Dependencies treebank for German

Emanuel Borges Völker[*] and **Maximilan Wendt**[*]
Universität Hamburg
emanuel.borges.voelker@studium.uni-hamburg.de
6mwendt@informatik.uni-hamburg.de

Felix Hennig
University of Edinburgh
mail@felixhennig.com

Arne Köhn
Saarland University
koehn@coli.uni-saarland.de

Abstract

We report on the conversion of the Hamburg Dependency Treebank (Foth et al., 2014) to Universal Dependencies. The HDT consists of more than 200.000 sentences annotated with dependency structure, making every attempt at manual conversion or manual post-processing extremely costly. The conversion employs an unranked tree transducer. This formalism allows to express transformation rules in a concise way, guarantees the well-formedness of the output and is predictable to the rule writers. Together with the release of a converted subset of the HDT spanning 3 million tokens, we release an interactive workbench for writing and refining tree transducer rules. Our conversion achieves a very high labeled accuracy with respect to a manually converted gold standard of 97.3%. Up to now, the conversion effort took about 1000 hours of work.

1 Introduction

Despite the availability of several German treebanks (TIGER (Brants et al., 2004), TüBa-D/Z (Telljohann et al., 2004), HDT (Foth et al., 2014)) and a fairly active research community, the only other larger German treebank which has been converted to Universal Dependencies is TüBa-D/Z (Çöltekin et al., 2017), consisting of 95.595 sentences (1.788k tokens). Until now, the largest German treebank distributed by the UD project was German GSD, consisting of 15.590 sentences (292k tokens). As that treebank's original annotation still stems from the pre-UD time, interesting syntactic constructs are often not annotated in accordance to the UDv2 guidelines[1]. Furthermore, the German UD guidelines themselves were often not up to date, sometimes even incomplete. Our work on converting the HDT to Universal Dependencies therefore also consisted in a large part of working out the best way to represent German dependency structures in the UD annotation schema; these decisions are encoded in the resulting treebank and – where applicable – were documented to be added to the general UD documentation.

The Hamburg Dependency Treebank is a native dependency treebank developed mainly for research in parsing; the annotation schema (Foth, 2006) was developed as part of the annotation effort. The texts in the treebank stem from heise.de, a well-known German technical website reporting about new software and hardware, technology-related politics, earnings of tech companies, inter alia. Some texts are short and formulaic, others are long editorials. The HDT has an average sentence length of 18.4 and more than 200.000 sentences are manually annotated. The text of the HDT can be distributed for academic use, the annotations are licensed under a Creative Commons share-alike license. The original treebank and its conversion to Universal Dependencies are available under `https://nats.gitlab.io/hdt/`.

We will give a rough overview of other treebanks which were converted to Universal Dependencies and of the methods which were employed, explain why our approach is different and how it works. After detailing our conversion process, we discuss general issues faced when converting a treebank to a different schema as well as specific problematic structures in our case and how we dealt with them, explain how we converted morphological features and finally evaluate the results of the conversion process. We close

[*] First two authors contributed equally.

[1] For example, names consisting of several tokens are regularly incorrectly annotated as *compound* instead of *flat* (or *flat:name*) in the German GSD treebank.

with a description of the (treebank agnostic) interactive conversion workbench developed as part of the conversion effort which enabled this large scale conversion.

2 Related Work

The Universal Dependencies (UD) project (McDonald et al., 2013) has caused many treebank maintainers to convert their treebank from their schema – often only used by this one treebank – to the UD schema. Examples can be found in Swedish (Nivre, 2014; Ahrenberg, 2015), Finnish (Pyysalo et al., 2015), Danish (Johannsen et al., 2015), Norwegian (Øvrelid and Hohle, 2016), Turkish (Sulubacak et al., 2016), Hindi (Tandon et al., 2016) and North Sámi (Tyers and Sheyanova, 2017). Most conversions rely on ad-hoc scripting of the conversion process, and a lot of manual intervention.

In some cases, a more systematic approach was taken. Pyysalo et al. (2015) based their conversion on dep2dep[2], a treebank conversion tool that allows for the definition of rules which are then converted to prolog code. Tyers and Sheyanova (2017) used XSLT, an XML tree conversion language, to build a pipeline to convert their parser output from the native schema to UD. Çöltekin et al. (2017) converted their constituency treebank with an automatic approach based on traditional head-finding heuristics. Seddah et al. (2018) converted a treebank by training a parser on a different treebank annotated with Universal Dependencies, using the original source annotations as additional features to the parser. This parser is then able to produce high quality UD annotations for other treebanks of the same language and with the same source annotation schema. For German, this approach is unlikely to work in the future as the different treebanks do not share a common source annotation schema.

Hennig and Köhn (2017) developed TrUDucer, a tool to convert dependency treebanks between different schemas based on the top-down tree transducer formalism (Maletti, 2010). This work builds on TrUDucer to convert the HDT and develop an interactive workbench for treebank conversion.

3 Converting between Dependency Schemas using Tree Transducers

We utilise tree transducers to convert dependency tree structures from the HDT schema made for German (and the HDT in particular) to the more general Universal Dependencies. The tree transducer formalism builds on the use of local context for partial conversion of subtrees based on predefined rules which are applied iteratively in a top-down fashion. Figure 1 shows an example conversion of a sentence, using a tree transducer defined by the following rules:

```
n:S() -> n:root();
n:SUBJ() -> n:nsubj();
n:DET() -> n:det();
n1:PP(n2:PN()) -> n2:obl(n1:case());
```

Each rule consists of a left-hand side and a right-hand side, where the left-hand side is a description of a structure found in the source treebank, and the right-hand side the corresponding structure in the target schema. The child nodes are listed in parentheses, each node is referred to by an identifier (`n, n1, n2`) and its dependency relation to the parent node (after a colon). Node structure can be changed arbitrarily, a common use-case is the matching of a function word and a content word and then switching their position in the tree. An example of this is the last rule given above, which is applied in the step from the second tree to the third one in Figure 1.

The structures are matched in the tree. Where to match is defined by the *conversion frontier* (Shown as orange lines in Figure 1). At the start of the tree conversion the conversion frontier is set at the root of the tree. After a rule is matched and applied, the conversion frontier is moved below the nodes that have been converted in this rule. The conversion then continues at the new conversion frontier, where again a rule is applied and the frontier moved. This way the frontier is moved down from the root to the leaves, until all nodes are converted. The rule list can contain multiple matching rules with gradually decreasing specificity, allowing to describe edge cases first and defining more generic rules later.

[2]`https://github.com/TurkuNLP/dep2dep`

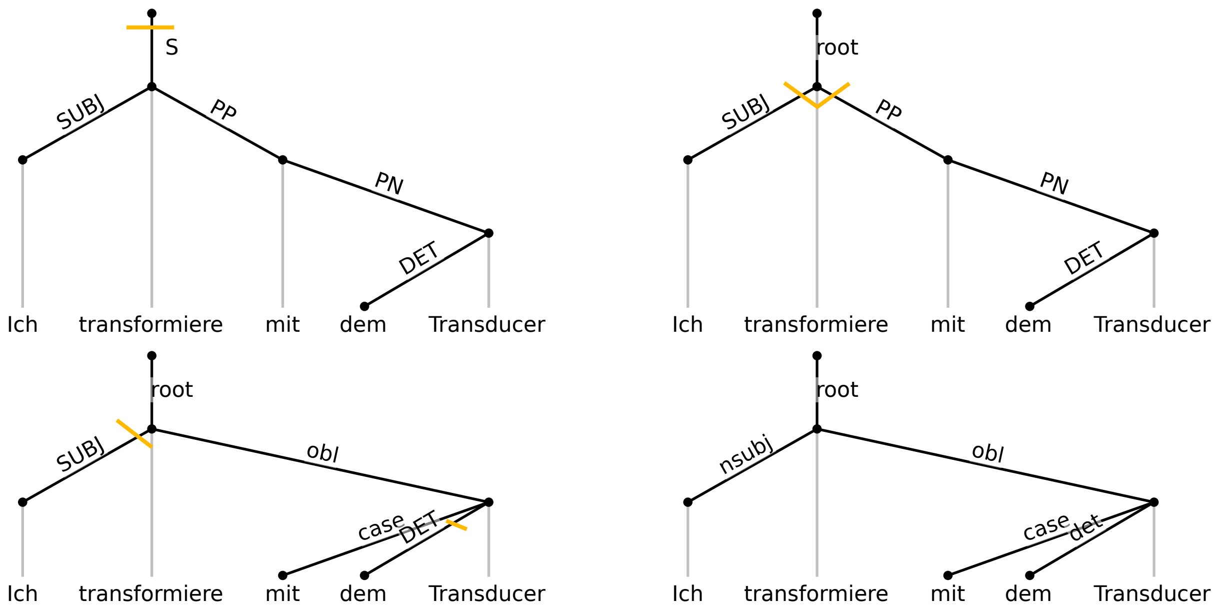


Figure 1: Conversion of "Ich transformiere mit dem Transducer" (*I transform with the transducer*); yellow lines indicate the state nodes, HDT labels are uppercase, UD are lowercase. Taken from Hennig and Köhn (2017)

This approach guarantees that trees are always well formed. As the rules all convert the tree locally, different subtrees can be converted independently of each other and different choices of where a rule should be applied next lead to the same result. This makes the process transparent to rule authors, deterministic and reproducible.

On top of this fundamental mechanism, the TrUDucer implementation provides certain convenience features. Similarly to the dependency relation, the PoS tag of a node to match can be specified directly by the use of a period. To formulate more complex conditionals with other node properties Groovy code[3] can be added to the rule in a *rule body*:

```
p.NN({n.VAFIN:AUX()}) -> p(n:cop()) :- { n.lemma == "sein" };
p:root({n:S()}) -> n:root(p:parataxis()) :- { p.ord > n.ord };
```

Groovy is a fully functional programming language and through the nodes the whole tree can be accessed and modified to accomodate edge cases. Also shown above is the use of curly braces on the left-hand side of the rule, used to match a frontier node and allowing to match more local context above the frontier. The first rule detects copula verbs (see Section 4.1), the second one checks that the node p is to the right of n. More details can be found in Hennig and Köhn (2017).

4 Converting the Hamburg Dependency Treebank to Universal Dependencies

While the HDT and UD have many similarities, most notably the use of dependency relations, they also have a few key differences which are relevant when converting the dependency trees from one schema to another. The HDT was annotated with a schema specifically tailored to the needs and particularities of the German language. In contrast, UD is a framework designed for cross-linguistic comparability. This is exemplified by the way dependency relations are treated. UD relations are headed by content words since this makes it possible to maintain high comparability when across differently structured languages[4]. HDT relations are headed by function words since this allows a more precise representation of the language-specific syntactic structure (Foth et al., 2014). The difference in focus is also noticeable when looking at what kind of information is valued and how it is represented in the respective annotation

[3]An embeddable java-based scripting language: `https://groovy-lang.org/`
[4]`https://universaldependencies.org/u/overview/syntax.html`

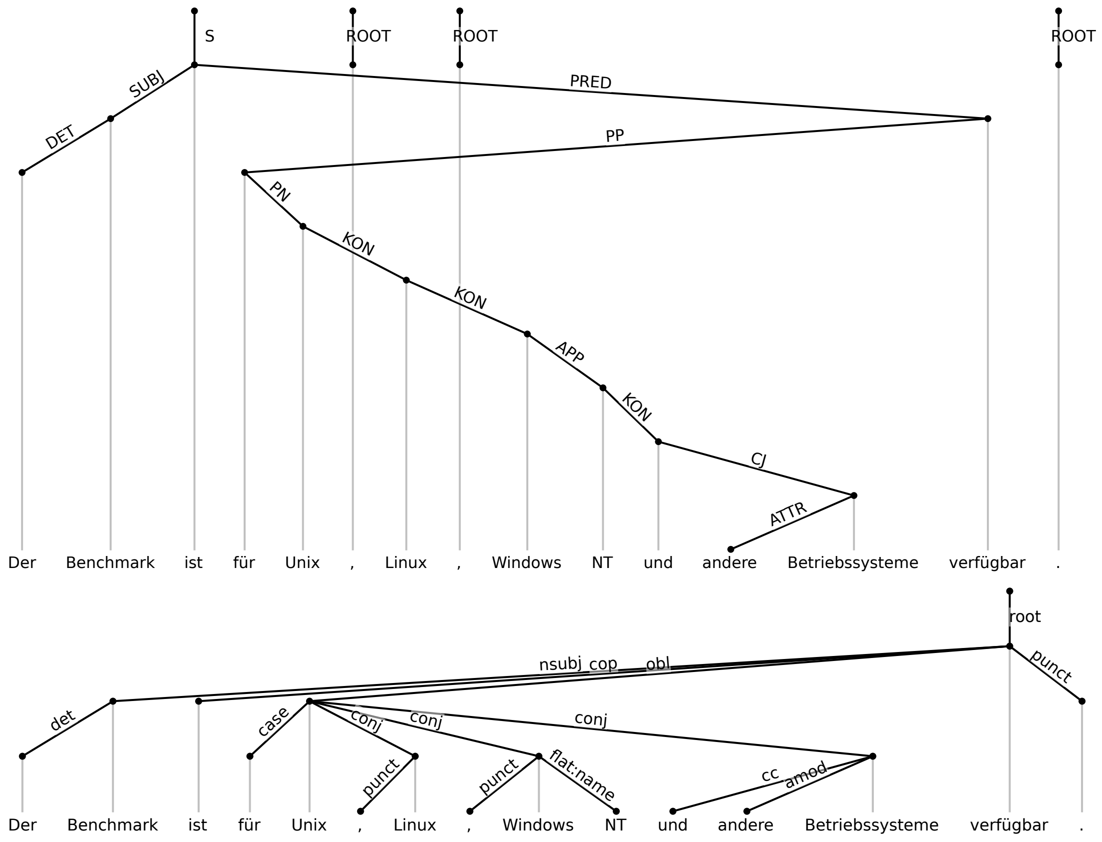


Figure 2: A dependency structure in the HDT schema (top) and the corresponding UD tree (bottom).

schemas. In the next sections, we will go into more detail about the difficulties caused by this and how we solved them.

Since Hennig and Köhn (2017) already covered the most common structures, a significant part of the recent conversion effort consisted in creating rules for edge cases and exceptions to the preexisting general rules as well as augmenting the preexisting rules using Groovy code in cases where they needed more complex predicates. We implemented a total of 99 conversion rules for dependency conversion. 34 rules are simple one-to-one mappings converting a single node considering only the previous relation and PoS tag. In about one quarter of the rules we make use of Groovy code to condition matching in ways not expressible otherwise. Usually this was only necessary to add trivial predicates, applying the tree transducer formalism as usual. The only exception to this are coordinating conjunctions, which we will go over in more detail after commenting on some of the simpler extended rules.

4.1 Considering information difference

We mentioned earlier that HDT and UD value different kinds of information. Even though in some cases, like with the 34 simple one-to-one rules, the dependency relations are equivalent and can be transduced easily, the rules often do have to make significant adaptations. The difference in focus between the two schemas means that, when converting dependency trees from the HDT to UD, we face three possible structural complications: incongruence when equivalent structures are represented differently across the two annotation schemas, information deficit when information not contained in the HDT is required by UD and information loss when information contained in the HDT is not kept in UD.

Incongruence can usually be solved by standard TrUDucer rules since the information contained in both representations is the same. The dependency trees simply have to be adapted as described in the Section 3; by changing labels, attaching relations in different places, inverting them and so on.

Information deficit often requires rules with the additional predicates Groovy code can provide because the information needed for the UD relations is not contained in the corresponding HDT relations. Still, the information needed can be extracted from the relations' immediate context most of the time. In one case this was not enough, so we used Wiktionary with minimal manual corrections to supplement our data and determine the correct conversion.

Information loss does not affect the conversion process since it does not impact the well-formedness of the resulting UD relations. It is also only information loss in the sense that the information is not contained it the dependency relations anymore. Much of it is still retained as part of the universal features.

Objects Objects are a good example for both information loss and one-to-one matching. The HDT uses seven different relation labels involving objects, covering phenomena like case (genitive, dative or accusative) and constructions with several objects. UD only differentiates between (direct) objects and – if there are several objects in the same phrase – indirect objects. Genitive, dative and accusative objects are converted to *obj* (or to *iobj* if applicable) and case information is mostly retained in the UD features. The other structures involving objects receive other labels, often depending on context: object clauses are converted to *ccomp*, object infinitives (an infinitive as complement of another verb) to *xcomp* etc.

Copula verbs An example of incongruence and of those simple extended predicates are copula verbs, which are not annotated in the HDT but required by UD - as shown in Figure 2. However, UD only accepts a maximum of one copula for most languages, usually a form of to be (called "pure copula") which is annotated as *cop*.[5] For disambiguation of copula and non-copula verbs, a check of the verb's lemma is required, which is a predicate not included in the TrUDucer rule syntax but easily added by implementing it in the Groovy code.

APP relations The *APP* relation is a notable beneficiary of the expanded predicates that Groovy code offers – converting these relations turned out to be particularly difficult since some PoS tags are unreliable in the HDT. This lack of reliability makes it harder to correctly disambiguate between the target UD relations in these cases since in the HDT, all consecutive constituents of noun phrases (as long as they are not determiners or attributes) are subordinated to their precursor as *APP* (Foth, 2006, p. 13f.), which leads to highly similar structures with the main distinguishing feature being those unreliable PoS tags.

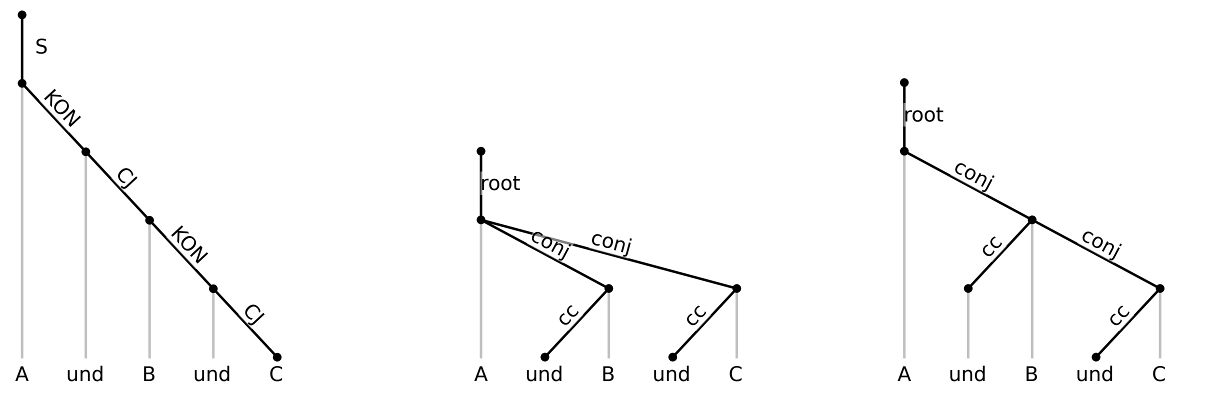
The standard case of the *APP* relation is an apposition, which is annotated as *appos* in UD. However, as can be seen in Figure 2, the *APP* relation is also used for the constituents of a noun phrase in general, for example in names or dates which consist of several tokens and whose UD relation is *flat*. We mainly recognise *flat* relations using certain PoS tags – they will be further discussed in Section 4.4. For *appos*, we use different predicates.

appos is either used when an *APP* relation is interrupted by punctuation, or as a general fallback rule when neither this nor the PoS tags resulting in *flat* apply. Since punctuation is not part of the HDT dependency relations, we used Groovy code to incorporate punctuation into the rules and help with recognising appositions.

Inherently reflexive Verbs Another interesting benefit of embedding Groovy code to extend the predicates is the availability of complex data structures for rule matching. One language feature required in the UD annotation schema but not encoded in the HDT (information deficit) is the special annotation of inherently reflexive verbs – their objects are annotated as *expl:pv* instead of *obj*. The HDT does not annotate reflexivity at all. While finding a reflexive use of a verb is trivial, differentiating between inherently reflexive and non-inherently reflexive verbs requires knowledge about the verb and its use context. This is impossible to achieve by using simple predicates in the treebank conversion because no such knowledge about the verb is given, but we can look through a list of verb usage generated by parsing the German Wiktionary[6] entries for verbs instead. The collaborative nature of Wiktionary means that the data is semi-structured, making information more difficult to extract. Also, verbs often are only inherently reflexive in specific contexts which are not easy to disambiguate automatically. Sometimes, entries in the German Wiktionary are simply incomplete, missing features like reflexivity completely. We manually corrected parts of the list of verb usage to prevent erroneous conversions.

[5]`https://universaldependencies.org/u/dep/cop.html`

[6]`https://de.wiktionary.org/`



(a) HDT conjunction: (A and B and C), ((A and B) and C) or (A and (B and C))

(b) Standard UD conjunction: either (A and B and C) or ((A and B) and C)

(c) Explicitly nested UD conjunction: this is unambiguous and only means (A and (B and C))

Figure 3: Conjunction ambiguities in HDT and UD annotations

4.2 Converting complex structures – coordinating conjunctions

One of the more interesting cases that require complex rules as well as scripting concerns coordinating conjunctions. As seen in Figure 2, the HDT treats coordinating conjunctions as long chains, with each following conjunct being subordinated to the prior. Most other dependency relations tend to branch off, leaving this chain intact, but the aforementioned *APP* relation is also annotated as a single chain of dependency relations in the HDT and regularly interrupts conjunction chains.

In contrast, UD also treats the first conjunct of a conjunction as the head, but requires that "all the other conjuncts depend on it via the conj relation"[7], creating a structure with several parallel branches instead of one long branch (respectively chain).

Issues with identifying the structures To convert these conjunctions to UD, the individual conjuncts need to be pulled up. This is straightforward enough for pure conjunctions, but there are two major issues.

The first issue is the interruption of conjunction chains by the APP relation (Figure 2). It is difficult and awkward to match and convert these structures using only tree transducers since the interrupting APP chains have no fixed length. Unlike with the simpler cases, we don't only use Groovy code here to expand rule predicates, but also for the rule itself. Groovy code allows us to check whether any child nodes of a conjunction interrupted by any number of APP relations are also conjunctions and, if they are, recognise them as part of the original conjunction and pull them up. Sadly, this is not a perfect solution because of the possibility of nested coordination, which leads us to our second issue: ambiguity.

When there are multiple conjuncts linked by conjunctions (as opposed to being linked by punctuation, as is commonly the case for longer conjunctions), there tends to be some ambiguity concerning their exact hierarchy. Figure 3 shows how this ambiguity is represented in HDT and UD with the general example phrase "A and B and C" and its three possible meanings: meaning 1 (A and B and C), meaning 2 ((A and B) and C) and meaning 3 (A and (B and C)).

Figure 3a shows how the example phrase would be annotated in the HDT. It would be thinkable to distinguish meaning 1 form meanings 2 and 3 by, for example, linking "B" to its previous conjunction via the *KON* relation instead, allowing the example phrase to have the same dependency structure as the phrase "A, B and C" if deemed more appropriate by the annotator. But since the HDT guidelines require a conjunct following a conjunction to always be attached to the conjunction via the *CJ* relation, the HDT is completely unable to distinguish between the phrase's three possible meanings (Figure 3a).

The UD representation is slightly less ambiguous. Meanings 1 and 2 are still annotated the same way (Figure 3b), but meaning 3 is represented by a different structure than the other two (Figure 3c).

Conversion details When converting such structures, we face the problem that the given dependency relations contain less information than the target dependency relations require. It is not impossible to

[7]`https://universaldependencies.org/u/dep/conj.html`

disambiguate such cases, but this generally requires access to, and understanding of, semantic information and context. In practice it would be necessary to check every conjunction manually to always correctly disambiguate them. For now, we settled on an automatic conversion without user input that converts as many conjunctions correctly as possible. Meaning 1 is by far the most common one in the HDT. Conjunction words generally are replaced by punctuation in longer conjunctions and in this case they are correctly converted to the corresponding UD structure shown in Figure 3b. The cases where we have meaning 1 coincide with the dependency structure of Figure 3a, and which would lead to a wrong conversion, are quite rare to nonexistent. This leaves meanings 2 and 3. Meaning 2 should be converted to structure 3b, but will by default be converted to structure 3c. Meaning 3 should and will be converted to the structure shown in Figure 3c.

In conclusion, the interruptions of *KON* relations by *APP* relations make their conversion more tricky, but pose a solvable problem. The ambiguity inherent to nested coordination remains an issue.

4.3 Decisions based on user input

For some rules we can not decide whether they should be applied even by looking at the global sentence context; they need user input. We re-use the ability to evaluate Groovy predicates by having functions that yield to the user and use their response. The answer is stored in a database so that the user does not need to be asked again on subsequent conversions. For example, the rules need to yield to the user when converting inherently reflexive verbs which are not in the dictionary and therefore cannot be told apart automatically. We are also considerung using rules that yield to user input for prepositional phrases, where in some cases the PoS tag is not sufficient to disambiguate the dependants below nominals and the dependants below predicates. Rules with interactive user input requests are implemented for both mentioned cases but are not used in the current conversion. The amount of manual labour needed even for these edge cases would still be substantial because of the size of the Hamburg Dependency Treebank. Right now the edge cases will fall back to default conversions until we either find a way to automatically distinguish these cases or we manually decide the relation for each occurrence once.

Fragmented sentences Due to the nature of fragments in the HDT, a default fallback rule is not an option in those cases. The HDT uses the label S for the root of a tree as well as the root of a fragment (e.g. a parenthesis, or more general a construction that cannot be integrated into the tree structure without breaking hard constraints imposed by the HDT annotation schema). In the HDT, it is acceptable for sentences containing these structures to have multiple roots, but UD requires sentences to have a single root. Our current conversion process is able to recognise these structures and convert the individual subtrees correctly, but is not yet able to automatically attach the fragments correctly due to the sheer number of possibilities.

Punctuation Another use case of the Groovy scripting language embedded in the rule file is the annotation of punctuation in the sentence. Punctuation is not annotated in the original HDT and therefore can not be converted by the TrUDucer rules as it is not part of the well-formed tree required by the TrUDucer rules. However, annotation of punctuation in universal dependencies is defined by four precise rules[8]. With heuristics we were able to apply those rules to the otherwise converted treebank. In some German sentences punctuation can be ambiguous. Whenever the heuristics discover such ambiguities, they will not annotate them. Punctuation for which no heuristics apply will also not be annotated. Example of that are unbalanced quotation marks or parentheses. Punctuation in these sentences will have to be manually annotated; until then, they are excluded from the release of the converted treebank.

4.4 Features

While the main focus of the TrUDucer software is converting dependency trees, which PoS tag and morphology conversion is not a part of, the software can also apply given lookup tables for conversion of said features. They are applied in an additional step after the dependency conversion. The annotation of morphology in HDT and UD is quite similar, therefore we can convert many features with another

[8]`https://universaldependencies.org/u/dep/punct.html`

one-to-one conversion schema. For now, features not directly encoded in the HDT are not annotated in the UD conversion. The tree transducer rules often rely on morphological information, and while morphology was not the focus of the HDT's original annotation, it still contains a lot of information which is lost or more difficult to access after a complete conversion due to the UD annotation being more coarse-grained than the HDT one. Thus we convert the morphology (in particular the PoS tags) after the dependency relations.

Part of Speech tags For conversion of Part of Speech tags, we use a simple mapping. As the HDT uses the Stuttgart-Tübingen-Tagset (Schiller et al., 1999) for PoS tags, we adapted the lookup table used in Çöltekin et al. (2017), who converted the TÜBA-D/Z, which uses the same tagset. An exception for the simple lookup is the PoS tag *PIDAT* (attributive indefinite pronoun with a determiner)[9] which, depending on the dependency relation, should either be converted to *DET* or to *ADV*.

Unfortunately, there are some problems with the source annotation. PoS tags and morphology were not a focus of the original annotation effort and therefore have reduced quality. Particularly unreliable are the annotations of foreign noun phrases. The three main PoS tags for nouns in the HDT are NN (normal noun), NE (proper name) and FM (foreign material). The HDT documentation states that the distinction between them is often difficult - when in doubt, the tokenizer was supposed to classify a noun phrase's tokens the way it would have if they were isolated (Foth, 2006, p. 43f.), meaning that their PoS tags differ on a case by case basis. On top of making it harder to distinguish between *appos* and *flat* when converting the *APP* relation, as mentioned in section 4.1, it also complicates the correct use of the *flat* relation's subtypes: *flat:name* and *flat:foreign*. The particular lack of reliability of PoS tags involving foreign words makes it impossible to systematically recognize foreign phrases as such and select the correct sub-type for them. Thus, we have not yet implemented *flat:foreign*. *flat:name* is somewhat less error prone.

5 Statistics and Evaluation

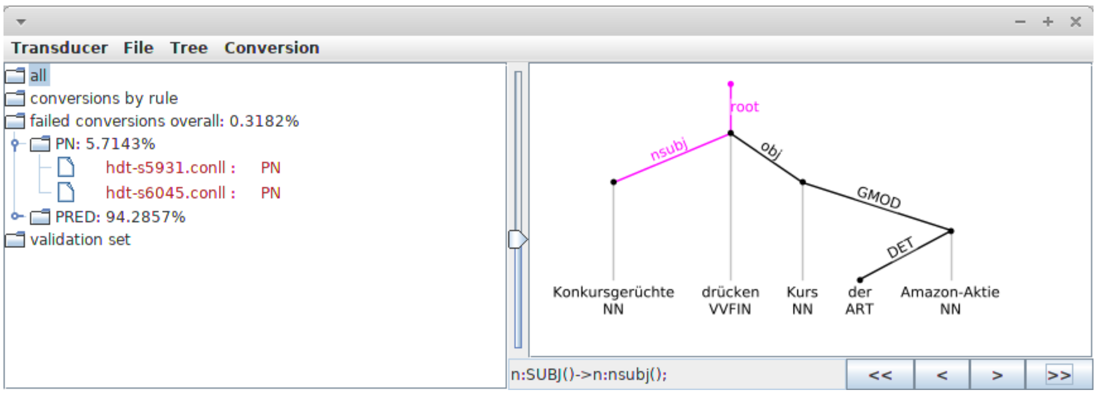
We manually annotated a set of 50 sentences held out from the rule generation process to evaluate the quality of the converter. The sentences were chosen by randomly sampling the part B of the HDT and also used as validation in Hennig and Köhn (2017). Our manual annotations differ slightly because of changes to the current UD standards. Punctuation marks (84 out of the 782 tokens) were ignored during this comparison. Looking at the 698 remaining converted words, 558 matched the hand-annotated dependency relations, leaving 127 words not matching the gold standard. Further analysis showed that the dependency label was actually correct in 88 of these cases but the transducer gave the dependency relation an additional (correct) subtype which was not given in the set of hand-annotated sentences. Some of the hand-annotated relations turned out to be wrong when comparing them to the result from the transducer. In the end, neglecting the punctuation marks, for 679 out of 698 words the automatically converted annotations matched the corrected hand-annotations, yielding a labeled accuracy of 97.3%. We further evaluated the conversion as performed by Seddah et al. (2018): by critically looking through 100 randomly selected sentences and checking for annotation and conversion errors. The resulting accuracy confirms the previous evaluation. Overall, 71% of the evaluated sentences where converted without any errors and 1506 of the 1548 non-punctuation dependencies where converted correctly, again yielding an accuracy of 97.3%.

This accuracy is significantly higher than other reported conversion accuracies; Seddah et al. (2018) e. g. report a labeled conversion accuracy of 94.75% and 93.27% on their held-out sets, which is twice the amount of labeled errors.

6 Interactive Workbench

In the context of dependency trees, the aforementioned transducer rules are applied to convert dependency relations and change the dependency heads. While the TrUDucer software itself is treebank-agnostic, the conversion rules are conversion specific. The development of those treebank-specific rules takes a big proportion of the effort put into converting the treebank. Therefore it is a huge advantage to have the

[9] `https://www.ims.uni-stuttgart.de/forschung/ressourcen/lexika/TagSets/stts-table.html`



```
n:SUBJ()->n:nsubj();
```

Figure 4: The TrUDucer GUI. Left: a selection of different overviews. All sentences of the treebank, Conversion steps sorted by rules applied, trees which could not be completely converted (sorted by the label of the first edge that could not be converted), all sentences of the validation set (comparison against a gold standard). Right: Interactive visualization of a selected conversion. Edges converted in this step are highlighted, unconverted edges with all-caps dependency labels, the rule applied in the current step is shown at the bottom.

process of developing conversion rules be as effortless as possible, which we tried to achieve with the graphical interface, the interactive workbench, for treebank conversion analysis.

One of the benefits of the interactive workbench is the support for treebank search queries. During the development of conversion rules, tests and validations are of huge importance to assess whether all rules work as intended and which structures are not yet converted correctly. Therefore, we added a graphical interface to TrUDucer that allows to search through the converted and unconverted treebank, highlight conversion problems and give additional insights on each individual conversion rule. For each conversion, it can visualize the rule applications and for each rule it shows exactly where it was applied. The latter turned out to be the most important information to assess the effect of each new rule and its interaction with already existing rules with minimal effort.

To further facilitate introspection, TrUDucer implements a filter over all the sentences of the treebank for the applied rules, either by searching for a specific phrase or by searching for subtrees in the dependency structure. The latter is able to search through both the converted and unconverted treebank. This was an important aid in checking the correctness for specific and possibly rare grammatical structures which were otherwise hard to find.

In addition to manually checking new rules by filtering the treebank for structures and applied rules, a regression check using sentences with manually created gold standard is used. Whenever a conversion is considered correct by an annotator, they can add that sentence to the gold standard. The regression check performs the conversion of the source annotation again and checks it against the gold standard. This process is usually run after the rule set was changed to check whether the modifications introduce unwanted side effects. This check is part of the GUI and allows to visually compare the gold standard annotation with the automatically converted annotation, highlight differences and modify and extend the gold standard within that interface, if needed.

The same graphical comparison of two dependency trees turned out to also be usable to visually represent a conversion rule, as the matching pattern is also given in form of a tree and can be compared to the replacement pattern as shown in Figure 5. By visualizing the tree-structure of both sides of a rule, we were able to find mistakes in newly written rules as it helps to get a more natural understanding of each rule.

TrUDucer can be used with the GUI for interactive rule development and in batch-mode to convert a whole treebank. As the treebank-specific rules are decoupled from the software itself, we hope to have created a software which is usable for flexible conversion of different dependency treebanks instead of just the HDT.

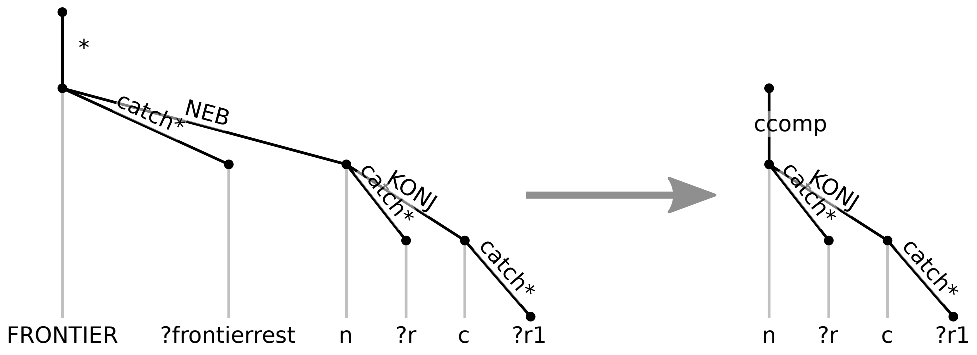


Figure 5: The transducer rule "n:NEB(c:KONJ()) -> n:ccomp(c())" visualised. Noticeably it also shows the frontier node in the matching tree and the catchall nodes (edges labeled with catch*) that are implicitly defined in the rule to allow better understanding of the interactions of the rule. Catchall nodes are special nodes in the rule tree in that they can match with multiple conversion tree nodes simultaneously.

7 Conclusion and Outlook

While we can convert 99.7% of the sentences during dependency conversion, only 90,7% of the treebank have been released at the time of publication. The large difference between dependency conversion coverage and overall conversion coverage is due to our focus being mainly towards dependency conversion, with morphological features and PoS tags only treated afterwards (as mentioned in Section 4.4). We only included sentences passing the UD validator, which in most cases of rejection complained about unattached fragments (in about 2% of all converted sentences), punctuation (estimated 3%) as well as incongruities between PoS and dependency relations.

For a complete conversion of the HDT to UD, there are still a number of things that need to be done. We need to find a way to (at least semi-) automatically resolve dependence ambiguity. This includes attaching fragments. The necessary infrastructure for manual attachment is already in place. However, an at least semi-automatic conversion would be preferable – if possible – due to the large number of fragments and their diversity. 0.9% of the sentences in the treebank contain at least one fragment, not considering fragmented punctuation marks. Similarly, we need to resolve the remaining issues with the conversion of *KON* and *APP* relations: we need to assess where the resulting relations are attached and make sure that the fallback rules minimise wrong attachments. We also need to improve PoS tag quality concerning noun types. We currently plan on doing this by training a tagger on more accurate data and using it to correct the noun-related PoS tags in the HDT, using the active learning approach proposed by Rehbein and Ruppenhofer (2017).

The rules for exceptions and difficult fringe cases concerning prepositional phrases need to be completed. This concerns cases where it is unclear whether the prepositional phrase should receive the *nmod* or *obl* label. Currently, we use *obl* for prepositional phrases attached to a predicate and *nmod* as a general fallback rule since the regent tends to be a nominal in the remaining cases. We plan to add specific rules for regents with other PoS tags as well to increase accuracy. These rules will most likely ask for user input since these cases are often difficult even for a human annotator due to their high ambiguity.

Composite pronouns need to be split into syntactic words[10], and fixed multi-word expressions need to be implemented. They are represented by the *fixed* relation in UD, but not annotated in the HDT. In an ongoing conversion of the TIGER corpus (also using TrUDucer), they are implemented using a list (Watter, 2018). We will probably use a similar solution.

We hope that our conversion of the HDT further establishes German as a UD language and plan to expand the German UD documentation by contributing our findings.

Acknowledgements We would like to thank the three anonymous reviewers for their helpful remarks.

[10]`https://universaldependencies.org/u/overview/tokenization.html`

References

Lars Ahrenberg. 2015. Converting an English-Swedish Parallel Treebank to Universal Dependencies. In *Proceedings of the Third International Conference on Dependency Linguistics (Depling 2015)*, pages 10–19, Uppsala, Sweden, August. Uppsala University, Uppsala, Sweden.

Sabine Brants, Stefanie Dipper, Peter Eisenberg, Silvia Hansen-Schirra, Esther König, Wolfgang Lezius, Christian Rohrer, George Smith, and Hans Uszkoreit. 2004. TIGER: Linguistic Interpretation of a German Corpus. *Research on Language and Computation*, 2(4):597–620.

Çağrı Çöltekin, Ben Campbell, Erhard Hinrichs, and Heike Telljohann. 2017. Converting the TüBa-D/Z Treebank of German to Universal Dependencies. In *Proceedings of the NoDaLiDa 2017 Workshop on Universal Dependencies (UDW 2017)*, pages 27–37, Gothenburg, Sweden, May. Association for Computational Linguistics.

Kilian A. Foth, Arne Köhn, Niels Beuck, and Wolfgang Menzel. 2014. Because Size Does Matter: The Hamburg Dependency Treebank. In Nicoletta Calzolari (Conference Chair), Khalid Choukri, Thierry Declerck, Hrafn Loftsson, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Language Resources and Evaluation Conference 2014*, pages 2326–2333, Reykjavik, Iceland, may. LREC, European Language Resources Association (ELRA).

Kilian A. Foth, 2006. *Eine umfassende Constraint-Dependenz-Grammatik des Deutschen*.

Felix Hennig and Arne Köhn. 2017. Dependency Tree Transformation with Tree Transducers. In *Proceedings of the NoDaLiDa 2017 Workshop on Universal Dependencies (UDW 2017)*, pages 58–66, Gothenburg, Sweden, May. Association for Computational Linguistics.

Anders Johannsen, Héctor Martínez Alonso, and Barbara Plank. 2015. Universal dependencies for danish. In Markus Dickinson, Erhard Hinrichs, Agnieszka Patejuk, and Adam Przepiórkowski, editors, *Proceedings of the Fourteenth International Workshop on Treebanks and Linguistic Theories (TLT14)*, pages 157–167, Warsaw, Poland, December.

Andreas Maletti. 2010. Survey: Tree transducers in machine translation. In Henning Bordihn, Rudolf Freund, Thomas Hinze, Markus Holzer, Martin Kutrib, and Friedrich Otto, editors, *Proc. 2nd Int. Workshop Non-Classical Models of Automata and Applications*, volume 263 of `books@ocg.at`, pages 11–32. Österreichische Computer Gesellschaft.

Ryan McDonald, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Täckström, Claudia Bedini, Núria Bertomeu Castelló, and Jungmee Lee. 2013. Universal Dependency Annotation for Multilingual Parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 92–97, Sofia, Bulgaria, August. Association for Computational Linguistics.

Joakim Nivre. 2014. Universal dependencies for swedish. In *Proceedings of the Swedish Language Technology Conference (SLTC)*, Uppsala, Sweden, November. Uppsala University, Uppsala, Sweden.

Lilja Øvrelid and Petter Hohle. 2016. Universal Dependencies for Norwegian. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Sara Goggi, Marko Grobelnik, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asunción Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Tenth International Conference on Language Resources and Evaluation LREC 2016, Portorož, Slovenia, May 23-28, 2016*. European Language Resources Association (ELRA).

Sampo Pyysalo, Jenna Kanerva, Anna Missilä, Veronika Laippala, and Filip Ginter. 2015. Universal Dependencies for Finnish. In *Proceedings of the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*, pages 163–172, Vilnius, Lithuania, May. Linköping University Electronic Press, Sweden.

Ines Rehbein and Josef Ruppenhofer. 2017. Detecting annotation noise in automatically labelled data. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1160–1170, Vancouver, Canada, July. Association for Computational Linguistics.

Anne Schiller, Simone Teufel, Christine Stöckert, and Christine Thielen. 1999. Guidelines für das Tagging deutscher Textcorpora mit STTS. Technical report, Universität Stuttgart / Universität Tübingen.

Djamé Seddah, Eric De La Clergerie, Benoît Sagot, Héctor Martínez Alonso, and Marie Candito. 2018. Cheating a Parser to Death: Data-driven Cross-Treebank Annotation Transfer. In *Proceedings of the 11th Language Resources and Evaluation Conference*, Miyazaki, Japan, May. European Language Resource Association.

Umut Sulubacak, Memduh Gokirmak, Francis Tyers, Çağrı Çöltekin, Joakim Nivre, and Gülşen Eryiğit. 2016. Universal Dependencies for Turkish. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 3444–3454, Osaka, Japan, December. The COLING 2016 Organizing Committee.

Juhi Tandon, Himani Chaudhry, Riyaz Ahmad Bhat, and Dipti Misra Sharma. 2016. Conversion from Paninian Karakas to Universal Dependencies for Hindi Dependency Treebank. In Katrin Tomanek and Annemarie Friedrich, editors, *Proceedings of the 10th Linguistic Annotation Workshop held in conjunction with ACL 2016, LAW@ACL 2016, August 11, 2016, Berlin, Germany*. The Association for Computer Linguistics.

Heike Telljohann, Erhard Hinrichs, and Sandra Kübler. 2004. The Tüba-D/Z treebank: Annotating German with a context-free backbone. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC 2004)*, pages 2229–2235.

Francis M. Tyers and Mariya Sheyanova. 2017. Annotation schemes in North Sámi dependency parsing. In *Proceedings of the Third Workshop on Computational Linguistics for Uralic Languages*, pages 66–75, St. Petersburg, Russia, January. Association for Computational Linguistics.

Camille Watter. 2018. Converting the german constituency TIGER treebank to the universal dependencies format. Bachelor's thesis, Universität Zürich.