Chapter 5

HinMaT Tools

“Talk is Cheap, Show me the Code”
-Linus Torvalds

“Software is a great combination between artistry and engineering.”
-Bill Gates

After having laid the foundation of HinMaT’s architecture and Lexical Organization, now is the time for describing the nuts and bolts of HinMaT in detail. This Chapter is dedicated for explaining various computational tools that were designed and developed by us for HinMaT. This chapter is divided into three sections, first section introduces process of deriving the tools from the architecture/lexical organization and functional classification of these tools. The subsequent sections describe the tools under each class.

5.1 Introduction

Computational tools are basic building blocks of any system/framework. They are derived from the technical design & architecture of the system. After working out the process flow, detailed architecture and lexical requirements of HinMaT, these tools were chalked out. The tools are classified on the basis of their functionality and usage area in the overall design and process flow. The detail classification is presented in following Figure 5.1:
5.2 Lexical Resources Management tools

Since HinMaT is a rule based MT system, it primarily relies on pool of lexical data whose detail organization has been described in the previous chapter (Chapter 4). This data needs to be managed neatly as per the principles of DBMS. The data management involves its effective creation, updation and deletion. The lexical
resources Management tools do this work for each of the lexical resources, which were introduced in the previous Chapter. In simple words the tools described here, are nothing but GUI based sophisticated data entry screen forms. As the lexical resources are to be primarily created and maintained by linguists and occasionally non-linguists, we have designed the screens in such way that they are easy to use and quite intuitive. The screens are designed around the requirements of linguists and principles of UI designing. All tools are user friendly and follow common design philosophy. They also have embedded Intelligence. Their design philosophy is explained in the following section:

5.2.1 Design Philosophy

In our design, UI screen is divided in three parts: actual data entry panel, command panel (navigation and DBMS operations panel) and data grid view panel. This design is common for all Resource Management tools. To ease the task of data entry, proper mix of modern UI controls is used. To avoid data entry errors, most of the controls (typically dropdown list/list) are populated with predefined master data. Not all fields on the forms are editable, few fields especially representing Table IDs are populated using auto increment mechanism and are displayed in read-only mode so that user can’t edit the values in such fields. All controls are preceded with intuitive labels.

The command panel contains navigation/DBMS operations panel in which basic navigation facility (moving forward/backward) to navigate through data records is provided along with DBMS operations for adding new entry, editing entry, deleting entry, searching data. The data grid view panel uses grid control to show the data in the lexical resource under consideration. The grid view also shows the summary statistics for the resource under context in its header part. It is important to note here that all Tools support UNICODE data format and all data operations are handled through stored procedure mechanism of MS-SQL Server® as it makes database operations more secured and faster. The data on the UI screen may go into single or multiple tables as the case may be. All tools prevent duplicate entry of data.

IntelliSense is provided on UI screens to automatically compute values of certain fields, e.g. on Transfer Lexicon screens the divergences are computed automatically, POS ontology sub categorization options are curtailed to reflect only options under
parent category in the hierarchy and word ending are computed automatically in SL/TL lexicon.

The general UI skeleton is given in the following Figure 5.2:

![GUI Controls used for Data Entry](image)

**Figure 5.2 General UI design for Lexical Resources Management Tools**

All tools have following buttons for Adding/Editing/Deleting data to the resource. All buttons can also be accessed using short cut keys (key accelerators). The details are given in following table (Table 5.1)

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>New</strong></td>
<td>This button is clicked, whenever user wants to add a new entry (record) to the resource. All controls on the form are initialized to default values and new blank record is created.</td>
</tr>
<tr>
<td><strong>Save</strong></td>
<td>This button is clicked to Add entry to the database table for the resource, after user fills up all controls on the form under context.</td>
</tr>
<tr>
<td><strong>Update</strong></td>
<td>User can select a row in the data grid for editing by double clicking the row, so that all controls on UI are populated from the values in the columns of selected row, user can then edit the values in the controls and finally click this button to reflect the changes to the database table(s).</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>User first fetches the row in the UI controls’ panel by double clicking the row in Data table and then deletes the row by clicking this button.</td>
</tr>
<tr>
<td><strong>Cancel</strong></td>
<td>After clicking the New button or at any other stage if the user wants to abandon the editing or addition operation, all changes are discarded and controls are initialized to their default values.</td>
</tr>
<tr>
<td><strong>Close</strong></td>
<td>Clicked to close the form.</td>
</tr>
</tbody>
</table>
Button state for each button toggles from ‘enabled’ to ‘disabled’ or vice-versa depending upon the process logic. All forms have facility to navigate through existing data in the resource. The detail functionality of navigation button panel is explained below (Table 5.2):

<table>
<thead>
<tr>
<th>Navigation Controls</th>
</tr>
</thead>
<tbody>
<tr>
<td>**First (</td>
</tr>
<tr>
<td><strong>Previous(&lt;)</strong></td>
</tr>
<tr>
<td><strong>Next (&gt;)</strong></td>
</tr>
<tr>
<td>**Last (&gt;</td>
</tr>
</tbody>
</table>

5.2.2 POS ontology Manager

This tool allows us to define a POS category in HinMaT POS ontology notation as described in the previous chapter and (Bhavsar & Pawar, 24-25 February 2011). Grammatical category of word is very important from computational point of view. HinMaT has adopted a unique ontology system for these categories, which allows further hierarchical sub-classification of individual categories up to three vertical levels with unrestricted horizontal expansion at each vertical level. User may or may not choose option at second or third level. UI uses grid view to show the entered data, which can be navigated using navigation control. Tool allows normal database operations like Add, Update and Delete. It is important to note here that, whenever category data is updated, the change is reflected in all relevant databases so as to keep the data in consistent state. The data created using this tool defines the grammar ontology for HinMaT and works as master data (basis) for all other tools of HindMaT, wherever grammatical categories are referred. The following screen shot shows the UI of the tool (Pl. see Figure 5.3).
5.2.3 SL/TL Word Lexicon Manager

This tool is used for creating basic lexical resource for HinMaT i.e. SL/TL words. This tool allows user to enter a word (non-verb POS category) and its associated grammatical information which includes root word, grammatical category as defined in the HinMaT POS ontology, grammatical and semantic features specific to the selected input language. User can also specify language as well as the usage domain(s) from the predefined list of domains for the entered word.

Tool allows normal database operations like addition, editing (deletion and updation) of words. The tool also has a feature called ‘Word Like’, which is used to generate the paradigm based feature extraction from entered words for similar new words. This makes it easy for a non-linguist also to enter new words by analogy. UI also supports grid based navigation & searching of the entered words on the basis of words & the POS categories. Grid View title bar (Header) presents the summary statistics of the entered Hindi and Marathi (in the present context of HinMaT) words. The data is stored in optimized format as reported in Lexical organization (pl. see section 4.4.2.1). This data storage methodology would help in building paradigms of word classes over the period of time. The tool computes word ending symbol (कारत) automatically. To
further ease the job of data entry GUI uses one checkbox captioned as ‘freeze controls’, whenever this checkbox is checked, the values in POS category, feature specification etc. are not cleared even after saving of current word and pressing ‘New’ button. So that while entering the next word, control values at that instance can be directly used as their states are not initialized. Tool allows searching of words based on word surface (word instance) or HinMaT POS category. This tool provides navigation facility to navigate through the entered records. This tool also provides a button to invoke Morphology Generator. The screen shot of the tool is given below (pl. see Figure 5.4).

![Figure 5.4 SL/TL Word Lexicon](image)

### 5.2.4 Transfer Lexicon Builder

This tool is used to store the SL to TL word mapping(s) information for chosen language pair. User can define the bilingual mapping between words, which are already entered in the Source/Target Lexicon. UI designing for this tool was bit challenging because three components viz. SL lexicon, TL lexicon and Transfer lexicon itself, were to be represented on single screen. Accordingly, we have designed our UI which represents the SL lexicon and TL lexicon and Transfer lexicon using three panels on main UI window. User can choose the SL word and its equivalent TL
word from the separate grids. The IDs for SL and TL lexicons are reflected in the Transfer Lexicon at real time as the user navigates through these lexicons.

This tool intelligently computes and records the GNPC as well as semantic feature divergences for the selected SL/TL word pair. The detail structure of Transfer Lexicon data table is described in the previous chapter. This tool allows many to many mappings between given SL word and TL word pairs. On source side a button named ‘Search TL’ is provided to find already the mapped word entries for this word in TL lexicon. The same is reflected in Transfer Lexicon grid panel also. Searching of word in SL or TL as well as Transfer Lexicon is also supported by this tool. For searching a word in the lexicon, one has to enter the particular word in the Text Box provided on SL or TL side and simply press the ‘enter’ key, if the word is present in the Lexicon, all details of the ‘search word’ are populated in other controls on that panel. Link to SL/TL lexicon is also provided in case user feels to update a word entry. Tool supports common database operations like add, update and delete on Transfer Lexicon panel. Navigation controls are provided on all three panels. It is important to note here that divergence flags are recomputed on every launching of this tool as well as updation of words through SL/TL Word Lexicon Manager tool. Summary statistics of each lexicon are shown in their respective data grid view.
Due to rich morphological nature of Indian Languages, considerable number of word forms get generated for each root word on SL as well as TL side, hence defining transfer mapping for each word is a laborious task. This tool has been developed to ease the Transfer Lexicon building process. It operates on simple idea that, if root words X, Y from SL and TL respectively, happen to be the translations of each other then following holds in general:

\[ x_{i_{ft}} = y_{i_{ft}} \forall i.f_{i}(i = 1, ..., n) \]

Where,

\( f_{i} \): feature structure associated with \( i^{th} \) word form consisting of < 

\( Gender, Number, Person, Case > \) feature tuple

\( x_{i_{ft}} \): denote the \( i^{th} \) inflected word form of root word X \( X \in SL \) having feature

---

30 Inflectional morphology occurs frequently with common/abstract nouns and ‘आ’ (Aa) ending adjectives and all verbs.
structure $f_i$

$y_{i/f_i}$ denote the $i^{th}$ inflected word form of root word $Y \in SL$ having feature structure $f_i$

This tool does the same thing i.e. it recommends the mappings between SL & TL word forms based on mapping of their root words. The tool only recommends the mappings, saving of mappings to Transfer Lexicon is left to the user. There are interesting observations regarding the mappings. They are explained in following section.

5.2.5.1 Mapping scenarios

Following cases are observed in the context of mappings.

a. Perfect match: Two feature structures $f_1$ and $f_2$ have same values for their features i.e.

$$f_{1,g} = f_{2,g}$$
$$f_{1,n} = f_{2,n}$$
$$f_{1,p} = f_{2,p}$$
$$f_{1,c} = f_{2,c}$$

b. Generic Match: The feature value either of GNPC features in $f_1$ and/or $f_2$ is specified using wild card symbol ‘*’ to mean generic value i.e.

$$f_{1,g \in \{*,m,f,n,ms,fn,mm\}} = f_{2,g \in \{*,m,f,n,ms,fn,mm\}}$$

$$f_{1,n \in \{s,p\}} = f_{2,n \in \{s,p\}}$$

$$f_{1,p} = f_{2,p}$$

$$f_{1,c \in \{*,\text{direct,oblique}\}} = f_{2,c \in \{*,\text{direct,oblique}\}}$$

It is worth mentioning here that, the generic feature matching can be misleading sometimes especially, in the context of Hindi and Marathi pair because the feature values are so much overloaded, that they can be resolved only after considering them in the group of two or three E.g. Hindi feature tuple <masculine,*,3rd,*> for common nouns in most of the cases represent two tuples <masculine, sing, 3rd, obl> and
<masculine, pl, 3rd, direct>, this is not true in case of Marathi, but in simple wild card matching (*,*), may recommend wrong tuples mapping like <masculine, pl, 3rd, obl> and <masculine, sing, 3rd, direct>. Same is true for Marathi tuple <feminine, *, 3rd,*>, which actually represent <feminine, sing, 3rd, direct>, <feminine, pl, 3rd, direct> and <feminine, sing, 3rd, obl>, this may also recommend wrong tuple <feminine, pl, 3rd, obl> to <singular, direct-case> values in Hindi, such cases should be avoided. More details can be found regarding these kinds of overloaded representations in Chapter 3.

c. Divergence: Divergence is common scenario in natural languages, and Hindi-Marathi pair is no exception to this. Main source of divergence in Hindi-Marathi pair at the lexical level comes from Gender feature because Marathi has three genders (masculine, feminine, neutral), while Hindi has only two (masculine, feminine). Hence, we have a big chunk of words from Hindi maps to ‘neutral’ gender words in Marathi. E.g. Hindi word ‘पेड़’ <m, *, 3rd, *> (tree) maps to Marathi word ‘झाड’ <neutral, sing, 3rd, direct>, which means, we can’t have an equivalent word in Marathi with ‘masculine’ gender. We have to consider such situations also while recommending words. Another scenario occurs; when Marathi inflected forms may contain ‘neutral’ gender word form besides ‘masculine’ and ‘feminine’ word forms. E.g. Marathi word form ‘मुले’ <neutral, pl, 3rd, direct> (boys), which is equivalent of Hindi word ‘बचे’ <masculine, *, 3rd, *>, there is a gender divergence in this case that to only for this word form. This is partial divergence case.

5.2.5.2 Algorithm

<table>
<thead>
<tr>
<th>Input</th>
<th>SL Root Word, TL Root Word.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Mapped word forms of the Root Words’ inflections.</td>
</tr>
</tbody>
</table>

**Formal Specification of the Algorithm:**

1. While there are no unmapped word forms
2. Find Matchings:

2.a First Pass:

For Each word \( w_{SL} \) in SL data row,
For Each word $w_{TL}$ in TL data row,
Inspect GNPC features of $w_{SL}$ & $w_{TL}$, if they match in their
GNPC Values (perfect or generic match, considering all
Scenarios as Discussed above)

i> Add them to the recommended mapping’s data grid view
and mark them as mapped.

ii> Not matched word pairs are added to the list of
unmapped words.

2.b Subsequent passes:
There are two scenarios here, first is divergence where none
of the words match in first pass, and second is few word forms
match, while others don’t have matchings.

i) Divergence case: Most likely there is gender divergence,
as person divergence is scientifically not possible and
Number divergence in our observation is almost impossible
because such words would be assigned ‘*’ value during
entry/generation process. In case of gender divergence,
match the Number/Person features and recommend mappings.
‘Masculine to Neutral’ gender divergence is more common
in Hindi-Marathi pair.

E.g. पेड़ (masculine) $\rightarrow$ झाड (neutral), कमीज़ (feminine) $\rightarrow$ सदरा

If number divergence is observed match Gender and
Person values to recommend mapping based on matching
result.

ii) Partial divergence: witnessed occasionally, in the
context of Hindi-Marathi pair, this refers to the
case where, generated Form has ‘neutral’ gender form
besides ‘masculine’ and ‘feminine’ word forms, in
such cases ‘masculine’ gender forms matching in ‘number’ and ‘person’ features are recommended for mappings.

3. Storing recommended mappings: The data grid shows Recommended Mappings, along with GNPC features and divergence information. User can inspect the mappings then choose the mappings for their final storage to the Transfer Lexicon.

4. End.

5.2.5.3 Tool UI

The UI of the tool is attractive and intuitive. It allows user to select a root word for SL as well as its mapped TL root word. The inflected forms for SL and TL root words are populated in the data grid views by clicking the ‘Search Word’ button for each side. After clicking the ‘Recommend’ button, it executes above algorithm and shows recommended mappings in separate data grid, where columns in the grid also show GNPC features in aligned manner using different colors along with divergence in features(if any). The UI is given in following Figure 5.6.

Figure 5.6 Word Recommender
5.2.6 Morphology Rule Manager

Morphology Rule Manager is responsible for overall management of Morphology Rules. The structure of the morphology rule is explained in the previous chapter. It primarily contains information about rule qualification criterion, general rule information like morphology type, affixes, Input language, and reversibility of rule etc. and output specification which specifies way to arrive to word forms by trimming/concatenation operations along with the feature specification for generated word form. This tool also has standard features like Addition/Updation and Deletion operation for new/existing rules as well as facility to navigate through the morphology rule database. The screen shot for this tool is given using Figure 5.7 below:

![Figure 5.7 Morphology Rule Manager](image)

5.2.7 Morphology Generator

This tool is integrated with Source/Target Lexicon Manager tool. It can automatically generate inflectional as well as derivational morphology (possible word forms of a root word) from the root word by recursive application of morphology rules. In inflectional morphology, POS category of word forms does not change and word
forms of stem word are generated by applying the permutations of GNP/Semantic features.

Tool also generates complete feature specification for the generated word forms which are embedded in morphology rule specification so that user can save the generated forms to SL/TL Lexicon. Tool also provides facility to change a feature value by manual intervention. Looking at word and its’ features, tool applies matching rule(s) to generate morphological forms. User has choice to reject the incorrect/ill-formed word forms. On every application of rule, its usage (correct, incorrect) statistics are updated. Rules with highest correct usage are picked up and applied first. Inflectional word forms and Derivational word forms are presented in separate grids. Structure of morphology rule is already described in previous chapter as well as reported in (Bhavsar & Pawar, 2011). We can recursively generate the inflectional morphology for word forms generated using derivational morphology. The detail algorithm of the Rule Applicator routine is given in the following subsection.

5.2.7.1 Algorithm

The algorithm devised for generating new word form has primarily four parts: rule selection, rule application order, rule application and rule output. The formal specification of the algorithm is as follows:

| Input → word Stem, lexical POS category as per HinMaT ontology notation, GNPC, Semantic features, Morphology Rule data base. |
| Output → Possible word forms based on qualified rules. |

**Formal Specification of the Algorithm:**

1. Rule extraction
   a. Get all rules matching lexical category, GNP, Case, semantic feature, word ending symbol(s) of input word, mark them as ‘original’, and order the rules by, exact match, wild card match and valid count.
   b. Get additional rules by transitivity based upon above fetched rules i.e. find new set of rules where target side features are on source side in the rule, by matching above(original) rule’s lexical category, GNP, Case TAM, recursively for each rule above. Mark them as
‘transitive’ rule, and order the rules by, exact match, wild card match and valid count.
c. Rule Sets obtained in a. and b. are merged retaining their order.

2. Rule Application
For each rule in Rule Set
i) Form morphed word from the rule specification by trimming the required characters from input word and conjoining the affix to input word.
ii) Initialize other features of morphed word as specified in rule. Add morphed word to output if not already added by other rules.

3. From the output list, save correct word forms with their full specification (update feature specification manually, if required). Update the success and failure count for the rules depending upon whether the word form was saved or not by user. Facility for editing is also provided (if required), in which case success count is not updated.

5. Stop after all rules are applied.

5.2.7.2 Tool UI

This tool is integrated with SL/TL Lexicon, hence after inputting a word, if the user wants to generate the morphology of the word, user is switched to this tool by clicking the button, ‘Generate Morphology’. The tool can also be invoked by choosing a word from list of entered words as shown in the drop down box. The UI shows the inflectional morphology and derivational morphology in separate data grid view. Each row in the data grid has option to save the generated word form as well as mark it as ‘valid’. Feature specification in the row cells is represented using drop down control so that it can be altered, if required. The screen shot for Morphological Generator is given in following Figure 5.8.
5.2.8 SL/TL Dependency Grammar Manager

This tool allows data entry and overall management of SL/TL Dependency Grammar rules based on Paninian framework, using proprietary format designed exclusively for HinMaT. Dependency Grammar (DG) specifies asymmetric binary relation between pair of words based on the notion of ‘dependency’, which may be syntactic, morphological or semantic. In case of Paninian framework, the dependency notion is based on syntactico-Semantic (Bharati, Chaitanya, & Sangal, 1995). In every dependency relation one word is nominated as ‘Head’, while other is called as ‘dependent’. The rule encodes criterion for testing dependency between word pair. POS category is an important eligibility criterion. Syntactico-Semantic dependency suffices GNPC agreement and semantic agreement between word pair. Detail rule format has been discussed in previous chapter. Linguist can specify the grammatical category of ‘Dependent’ and its’ proposed ‘Head’, as defined in HinMaT POS ontology. Besides this, the tool also allows to specify, ‘Karanta’ (word ending) information for root words (stems) of both dependent and head words, as this information is very important in Paninian Dependency framework for deciding the nature of GNPC agreement test. Use of wild card (‘*’ symbol) symbol in category
specification, provides flexibility and generality in rule application. We can also specify the possible word position (left or right or both) of the dependent word w.r.t. its Head word for proposed dependency relation, which is formally represented by drawing arc from ‘Head’ to ‘Dependent’ with arrow pointing towards ‘Dependent’. The arc label formally nomenclatures dependency relation. We have adopted the IIT Hyderabad Tag set by-and-large for labelling the dependencies (Begum, Husain, Dhwaj, Sharma, & L. Bai, 2008) along with our own tags in some cases (pl. see Appendix C). The feature agreement test specifies, how to perform the GNPC & semantic agreement tests i.e. whether feature in both ‘Dependent’ and its ‘Head’ should match or mismatch or feature test is not relevant for that context? User can also provide short description of the rule. Tool UI like other tools, supports grid based navigation of the entered words. The tool provides facility to navigate through the morphology rules as well as normal DBMS operations like addition/updation and deletion of rules. The language wise rule summary statistics are shown in the data grid’s header. Following screen shot (pl. see Figure 5.9) shows the UI of the tool.

![Figure 5.9 SL/TL CPG Grammar Manager](image)

5.2.9 Transfer Grammar Builder

This tool is used to store the transfer grammar (TG) for language pair under MT system, which is very crucial computational resource for RBMT systems like HinMaT. This tool aids creation and overall management of the transfer grammar for HinMaT. It is primarily responsible for storing the CPG Dependency Grammar Rule
mappings between SL and TL language grammar rules. UI design is similar to that of Transfer Lexicon Builder tool. It presents the SL and TL dependency grammars along with Transfer Grammar in three separate panels. The detail structure of Transfer Grammar has been discussed in previous chapter. The grammar rule ‘precision’ and ‘divergence’ flags are computed automatically. Tool provides navigation facility for SL, TL grammar and Transfer Grammar rule database. Grid view is presented in all three panels to show the total data in the grammar resource. Navigation through either of the SL or TL panel is reflected in the controls of the Transfer Grammar panel at real time. Like the Transfer Lexicon tool, the divergence/precision flags for each entry in the Transfer Grammar are recomputed at the time of launching of this tool to avoid any inconsistencies in the data. Transfer Grammar summary statistics are shown in the data grid view header.

Though the tool permits one to many mappings of rules, in our till date observation only one to one mapping is observed. The following Figure 5.10 shows the UI of the tool.

![Figure 5.10 Transfer Grammar Builder Tool](image)

5.2.10 SL/TL Root Verb Lexicon Manager

The verb abstraction process has been already discussed in the previous chapters 3 and 4. Accordingly, verb lexicon creation process involves three major steps viz. creation
of Root Verb Lexicon, creation of Verb Morphology Rule data and development of Verb Morphology and Demand Frame Generator tool. The Root Verb Lexicon data is fed to the Verb Morphology Generator program, which then produces different verb forms by the application of verb morphology rules. Out of the generated forms (more than 200+) correct forms as selected by user are then stored to SL/TL Verb Lexicon. This tool is responsible for creation of root verb master data. As already discussed, the semantic requirement of verb as denoted by semantic frame remains constant across verb forms, hence we have associated that with the Root Verb itself. This tool captures the data such as root verb word, its language, its verb POS category in HinMaT ontology, word ending (computing automatically), verb class (state, motion etc.) and semantic frame. The semantic requirement denotes the GNPC features, semantic features of participating noun, its POS category as defined in HinMaT POS ontology, existence requirement i.e. whether Karaka in context is mandatory/optional/not required and its placement in the sentence w.r.t. to verb word i.e. left, right or both. The Karaka theory recommends 07 Karakas, but in our specification, we have also included semantic specification for three additional Karakas i.e. JK1, PK1 and MK1 to specify Prayojya Karta (JK1), Prayojak Karta [Sponsor] (PK1) and Madhyasth Karta [mediator] (MK1). These Karakas are required for causative forms of verb. Designing of the UI for this tool was really challenging task. The screen shot for this tool is given in following Figure 5.11.

The Data entry panel is subdivided into three panels viz. Verb-Basic Information, Semantic Frame, Semantic Frame grid view. Besides this, the tool has a Navigation panel and data grid view for showing all entered root verbs. For capturing the semantic frame, we have used a Tab control. Each page in the tab, collects information pertaining to specific Karaka. It is important to note here that, one can specify multiple entries for each Karaka slot. All Karaka entries are shown using separate grid view. Clicking on a row in this grid will populate entries on the Karaka tab. The navigation panel helps to navigate through the entered root verb data as well as addition/ updation/ deletion operations for root verb. Like other tools, all grid views show summary statistics.
5.2.11 Verb Morphology Rule Manager

This tool is used to manage the verb morphology rules. The structure of verb morphology rule is described in the previous chapter. Since the rules are applied on the root verbs, basic rule eligibility criterion is restricted to verb language, its word ending symbol (Karanta) and most importantly the POS category in HinMaT ontology. The output specification of the rule specify the TAM label, TAM suffix that gets conjoined with the root verb word, number of characters to chop from root verb, GNP and TAM specification of the proposed new verb form and the verb agreement information. This data is auto populated on selection of TAM Suffix and TAM Label from drop down boxes. The tool also provides standard HinMaT features such as navigation button panel and facility for addition/updation/deletion of new/existing morphology rules. The data grid view showing summary statistics of entered morphology rules is also included on the UI of this tool. The new verb form is
obtained by concatenation operation on root verb form and TAM suffix using following equation:

\[ NewVerbForm = \text{Concat} (\text{Substr} \ RtVerb, \text{Chars to chop} , \text{TAM suffix}) \]

Where,

\[ \text{Concat}(string_1, string_2) \] – Performs concatenation of two argument strings i.e. \( string_1 \) and \( string_2 \).

\[ \text{Substr}(Str, nchars) \] - Substring function returns the substring of \( Str \) by chopping \( nchars \) right characters from the string \( Str \).

We have done extensive thorough work and created good number of morphology rules for both Hindi as well Marathi, which is one of the major contributions of our research work. The present rule statistics is given by following table (Table 5.3).

<table>
<thead>
<tr>
<th>Input Language</th>
<th>Simple Verb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hindi</td>
<td>6,497</td>
</tr>
<tr>
<td>Marathi</td>
<td>5,531</td>
</tr>
</tbody>
</table>

Another, good outcome of our study on verb systems of Hindi and Marathi was identification of TAM suffixes and TAM labels. Following table (Table 5.4) shows number of TAM suffixes and TAM labels in HinMaT repository:

<table>
<thead>
<tr>
<th>Input Language</th>
<th>Simple Verb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TAM Suffix</td>
</tr>
<tr>
<td>Hindi</td>
<td>413</td>
</tr>
<tr>
<td>Marathi</td>
<td>641</td>
</tr>
</tbody>
</table>

The following Figure 5.12 shows UI screen shot of Morphology Rule Manager.
5.2.12 Vibhakti Frame Manager

Vibhakti frame is integral part of verb’s Demand Frame, it is required during generation of verb form’s demand frame. The relevant table structure for Vibhakti Frame Manager has been already discussed in the previous chapter. Vibhakti frame specify Vibhakti Marker (post position marker) symbol for each Karaka slot. Theoretically, it depends on the TAM feature of verb form. The TAM features are symbolically represented by TAM label, which can be thought of as stem/root form of TAM suffixes. The UI permits users to define a new TAM label and Vibhakti marker specification for each Karaka for that label. To avoid typo errors, user can select Vibhakti Marker (Post-position marker) symbols from the prefilled dropdown box. The tool has facility to add/update/delete Vibhakti frame(s) and navigate through the Vibhakti frames as displayed in the data grid. Grid shows the summary statistics of the Vibhakti frames. The UI of this tool is given below Figure 5.13.

Figure 5.12 Verb Morphology Rule Manager
The current Vibhakti frame data at the time of writing this thesis is presented in following Table 5.5.

Table 5.5 Vibhakti Frames Data Summary

<table>
<thead>
<tr>
<th>Language</th>
<th>Vibhakti Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hindi</td>
<td>206</td>
</tr>
<tr>
<td>Marathi</td>
<td>149</td>
</tr>
</tbody>
</table>

5.2.13 TAM Suffixes & TAM Label Manager

TAM Suffixes and TAM Labels are required for Verb Morphology Rule creation. This simple tool is used for managing the TAM suffixes and TAM labels along with their GNP specification, TAM specification and verb agreement information. This data serves as master data for Verb Morphology Rule manager, whenever TAM Label and TAM suffix is selected in Rule Entry screen other fields such GNP/TAM etc. for the morphology rule are auto populated. The algorithm for computing TAM label has been described in (Bharati, Chaitanya, & Sangal, 1995). In simplest words TAM label is nothing but the stem form of a TAM suffix i.e. one TAM label suffices many TAM suffixes that are inflected for their GNP features. In Paninian analysis, Vibhakti
frames are tightly coupled with the TAM labels. The UI has standard HinMaT UI design features, such as navigation button panel, buttons to add/update/delete entries, and data grid view with summary statistics. Following screen shot Figure 5.14 shows the UI for this tool.

![Image of TAM Label Entry Screen](image)

**Figure 5.14 TAM Suffixes and TAM Label Manager**

### 5.2.14 Verb Morphology Generator

Verb morphology generation is more complex than morphology generation for non-verb POS categories. In CPG the verb arguments are not restricted to only subject, object or indirect object (in case of di-transitive verb) as in the case of CFG or other western grammar formalisms, hence verb morphology can’t be thought of as mere generation of new word forms. Every verb has a Demand frame associated with it (*Detail discussion on demand frame conceptualization is presented in section 4.4.2.2, Chapter-4*). Formally, a demand frame denotes argument structure of verb and is composed of Karaka slots, which specify semantic requirements for a karaka and case marker (Vibhakti symbol) for each Karaka. These constraints include semantic and syntactic requirements of verb w.r.t. each Karaka (*karta, karma* etc.). Syntactic
requirements specify the case and case marker (Vibhakti symbol) for a Karaka associated with particular morphological form of verb, this syntactic requirement depends on TAM features for that morphological form. The collective syntactic requirement for all Karakas is known as Vibhakti frame. Semantic requirement specify semantic constraints on nouns including its ontological class and semantic features. The collective semantic requirement specification for each Karaka slot is called as Semantic frame. The semantic frame of verb is common across all morphological forms of verb i.e. it is associated with verb root. Semantic frame along with Vibhakti frame specify the demand frame for verb form. The semantic frame is associated with verb’s root while Vibhakti frame is associated with particular TAM label and hence verb form.

Demand frames may be shared with certain forms of a verb, which means maximum number of demand frames, for a root verb, is equal to number of verb forms (GNP/TAM inflections of verb root) e.g. Hindi verb ‘खाना’(eat) require that it’s Karta karaka(subject) must have the ability to eat i.e. it should have +animate feature. Also in past tense form of this verb form i.e. ‘खाया’, this Karta must take Vibhakti marker ‘ने’ while in non-past forms like ‘खाता है’ will not take any Vibhakti marker.

We have to also think about the verb frame or demand frame generation in the same process. Before we discuss this Morphology Generator tool,

5.2.14.1 Algorithm for Verb Morphology and Demand Frame Generator

The steps for generating Verb Morphology and Demand Frames are put in the form of formal algorithm. The following algorithm is used for generating the verb morphology as well as Demand frame for each morphological form.

Input → Root Verb (हरा), lexical POS category (in HinMaT ontology notation), Semantic frame, Vibhakti Frames database, Verb Morphology Rule database.

Output → possible verb forms and their Demand Frame based on qualified rules.

1. Start
2. Morphology Rule Extraction: Pick the matching rules from Rule Database by matching the POS category, Karanta, verb class and language fields with that of input verb root.

3. Rule Applicator:
   For each picked up rule, apply the rule to arrive the new verb form as well as its Demand frame as,
   
   a. New_Verb_Form = (root_verb - No. of Characters to chop as specified in rule) + TAM suffix as specified in rule.

   b. New_Verb_Form’s demand frame = Merge(SEMANTIC FRAME_{Root_Verb}, Vibhakti Frame_{TAM LABEL}),
      Where,
      
      $\text{Merge}(\text{Semantic Frame}, \text{Vibhakti Frame})$:
      This function merges each Karaka slot of Semantic Frame of Input root verb with corresponding Karaka slot in the Vibhakti Frame i.e.
      
      Demand_Frame.K_n = SemanticFrame.K_n \oplus VibhaktiFrame.K_n
      
      $K_n$: $n^{th}$ Karaka slot, where $n = 1, 2, 3, 4, 5, 7, JK1, PK1, MK1.$
      
      $\oplus$: merges $i^{th}$ element of Semantic frame with $i^{th}$ element of Vibhakti frame.
      
      Vibhakti Frame is obtained from TAM Label (as derived from TAM Suffix (Bharati, Chaitanya, & Sangal, 1995) as specified in the Morphology Rule.

   c. Collect the generated verb form and its’ demand frame in Nested Data Grid Views.

4. User will examine the generated forms and select the correct forms for storing them to Verb Lexicon.

5. End.
5.2.14.2 Morphology Generator UI

This is perhaps the most complex but still neatly designed UI of HinMaT, in which user can drill down to lowest level of abstraction. This tool is integrated with Root Verb Lexicon Manager. So that this tool may be invoked either after entering the root verb with Root Verb Lexicon Manager (explained in 5.2.10 above) tool or directly from main menu of HinMaT Lexical Resource Manager UI menu. In case of direct invocation, user has to put the already entered root verb in otherwise case, entered root verb gets automatically populated in the text box. We have also made provision on tool’s UI for inputting intensifiers\(^{31}\) for populating the compound verb forms for the root verbs. However presently, this tool can’t generate Compound verb forms. After clicking on the ‘Generate’ button, the above stated algorithm gets operated with verb morphology rule database. The algorithm generates the possible verb forms along with their respective demand frames for chosen morphology rules. The generated output is shown in first grid view, where user can inspect the generated verb forms and their auxiliary information. Demand frame for verb form(s) is shown in the next data grid view. The user can intern view the individual Karaka slot details, for each of the Karaka entry by clicking on appropriate Karaka slot denoted by ‘K1’ or ‘K2’ etc. The semantic frame slot and Vibhakti frame slot for the chosen Karaka are shown in two separate grid views placed side by side. User can select the correct forms and save them to SL/TL verb lexicon by clicking the ‘Save’ button. Each Data Grid shows the summary statistics of the displayed data. The UI screen is presented by following Figure 5.15

---

This tool is used to manage the verbs generated using Verb Morphology Generator tool. Before the development of morphology generator tool, this tool was used to enter the verb forms in the lexicon. It is also used for entering verb forms which can’t be generated by morphology generator. It allows to specify the verb features as well as its demand frame. Searching of verb form is also possible in this tool besides the regular add, update and delete operations. The screen shot of this tool is given in following Figure 5.16
5.2.16 Verb Transfer Lexicon Builder

This tool is used to create the Verb Transfer Lexicon, which stores SL and TL verb pair mappings for chosen Language Pair, along with GNP, TAM, agreement and divergence information. The divergences are computed automatically by inspecting and comparing the features in SL and TL verbs. Like Transfer Lexicon Builder tool, SL side and TL sides shown in two separate panes and each is coupled with data grid view and navigation button panel. The bottom panel shows the details of Verb Transfer Lexicon. It contains UI controls showing the IDs of SL and TL verbs, verb words, their languages and divergence information. Buttons for adding/updating/deleting transfer mappings are also provided along with navigation buttons and data grid view showing all mapped entries. Data displayed in upper SL/TL panels is kept in read only mode. All grids show summary statistics. The tool UI is shown in following Figure 5.17. Data search facility is also implemented for searching SL or TL verbs within SL or TL lexicon as well as Verb Transfer Lexicon.
This simple tool is responsible for managing the predefined phrases used during the phrase marking activity of Pre-processing step. This tool stores the phrases in the Phrase Table. Tool allows addition/updation/deletion operations as well as navigation through the phrasal lexicon. Phrase duplication is checked, while insertion of new phrase. Note here that phrases are stored with space character between words but the phrase marker replace it with hyphen (‘-‘) character whenever it finds matching phrase in the Phrasal Lexicon. Language wise summary statistics are shown in grid view header. Tool UI is shown in the following Figure 5.18.
This tool is used to store and manage another crucial Pre-processing resource used during sentence extraction. We store abbreviations which end with dot (‘.’) like delimiter symbol along with their input language as this symbol conflicts with the popular sentence delimiter full stop used by most of the natural languages (NL), such use of delimiter symbol can create problem during sentence extraction process. Whenever sentence extractor tries to extract sentence based on full stop symbol, we first look into this resource to decide whether to extract the sentence or skip the current full stop symbol? Tool UI is simple and intuitive, which allows navigation and normal DBMS operations like Add/Delete/Update operations. Abbreviation duplication is checked, while insertion of new record. The tool UI is presented using following Figure 5.19.
5.2.19 NER Phrases Lexicon Builder

This tool is used to create NER expressions and store them in NER lexicon, which are used by NER phrase marker entity. The NER expressions are built using standard regular expression notations. The tool allows to Add/Update/Delete NER expressions as well as allows navigating through NER lexicon. The tool UI is given in following Figure 5.20.
5.2.20 Resource Summary Dashboard

Though every lexical resource tool shows the summary statistics information in its’ grid view, still we felt the need for having all summary statistics under one hood, hence we have designed and developed this dashboard, it shows summary of entire HinMaT repository on single screen. The dashboard screen design is self-explanatory it encompasses all resources discussed so far ranging from lexical resources to grammar resources, tree bank etc. It is worth mentioning here that we have used Link Buttons to label the resource name next to each textbox. Clicking the button will actually open the resource in new blank frame form. The screenshot of the dashboard is shown in following Figure 5.21

**Figure 5.20 NER Phrases Lexicon Builder**

![NER Phrases Lexicon Builder](image_url)
5.3 Pre-processing Tools

The MT workflow of HinMaT has been already described in the Chapter 3. According to that workflow MT life cycle consists of three phases viz. Pre-processing, Core MT process: Parsing/Generation and Post-processing. The details of Pre-processing activities and their need has been already discussed in the previous chapter, here we are going to discuss the technical design of the Pre-processing tools. These tools are integrated in the main HinMaT UI. As such these tools don’t have UI, they operate on the input document. In this section we are going to describe the various pre-processing tools as designed and developed for HinMaT.

5.3.1 Phrase Marker

Phrase Marking is inevitable pre-processing activity in most NLP applications. Phrase Marking marks group of words as a phrase i.e. computationally one unit for subsequent NLP processes like tokenization, chunking, parsing etc. In the context of MT, it helps to improve the quality of MT output, because there may not be one-to-one correspondence between SL and TL words in the MT input text, hence for getting correct translations, we need to treat group of words as one unit and directly map it to its literal translation. The phrase marker routine performs the same task. The phrase marking can be done in two modes viz. automatic and manual. In automatic mode, the program marks the phrases by looking into the Phrasal Lexicon. Though the phrasal
lexicon is expected to have extensive list of phrases, still it may not contain the phrase which user expects in the input text, hence this tool allows manual phrase marking in the input document for such cases. The tool also adds manually marked phrase(s) to the Phrasal Lexicon. Input to the Phrase Marker is input document. Words in the marked phrase are hyphenated i.e. blank spaces are replaced with ‘-’ (hyphen symbol). The algorithm of the phrase marker is given below.

5.3.1.1 Algorithm

Input: Input Text, Phrasal Lexicon

Output: Phrase marked Input Text

1. Start
2. For Each Word \( w_i \) \((i = 1, 2, 3\ldots, \text{length of input text})\) in the input text
   
   2.1 Fetch all phrases \( P_n \) starting with \( w_i \) in descending order of their lengths.
   
   2.2.1 For each phrase \( P_i \) \((i = 1..n)\)
   
   2.2.2.1 Try to Match \( P_i \) in substring of Input text starting with index position \( j \) of \( w_i \) in the Input Text i.e. \( \text{StrMatch(SubStr(Input Text, } j', n), P_i) \). There are two cases i.e. either match is found or not found.
   
   \[ \text{StrMatch(str1, str2)} \rightarrow \text{Returns true, if str1 and str2 are same otherwise false.} \]
   
   \[ \text{SubStr(str1,start, no. of charcters)} \rightarrow \text{Returns substring sliced from str1 starting at index position start.} \]

   2.2.2.2 If the match is found, Replace the matched phrase in the input text with hyphenated phrase i.e. replace blank space(s) between the words of the matched phrase with hyphen ('-') character.
2.2.2.3 In case none of fetched phrase matches, in the extracted input substring, skip \( w_i \) and move to next input word.

3. Return hyphenated input text for further MT processing.

4. Stop.

5.3.2 NER Phrase Marker

In recent years, the Named Entities (NE) has become ‘hot topic’ in NLP discussions. NE refers to all such words which are patterns of some kind and they serve special purpose (use). Common examples of such NERs are all dates (with different writing styles), telephone numbers, all numbers, currency figures, all sorts of encoded ID numbers (with particular encoding logic), PIN code, Credit card PIN number etc. Practically it is not feasible, rather impossible to store all instances of such words in the lexicon as their instance cardinality is too high. They pose a big challenge for knowledge based NLP applications. The problem is severe for Machine Translation problem because MT involves at least two languages. Problem to this issue is addressed by tool called ‘Name Entity Recognizer’ (NER). Basic function of NER is to identify such patterns or NEs in the input document and tag them properly. Further they should be generally stored to temporary lexicons\(^{32}\) because NEs are scoped to specific document/sentence instance. NEs may be composed of single word (or connected multiple words) or multiple words separated with blank space character. In the context of MT after a NE is identified, we have to store it to temporary SL lexicon. We need to either transliterate NE or partly translate it using bilingual dictionary and partly transliterate to obtain the equivalent TL form. We also need to store SL/TL mapping to Transfer Lexicon (preferably temporary) because it is required for generating TL output. Our implementation of NER is rule based. Our NER uses NER phrase lexicon to load the NER pattern definitions (regular expressions) and find their matches in the input document. All matched instances are hyphenated and replaced in the input document. They are also stored to temporary SL lexicon and their transliterated and/or translated Marathi form is stored to TL lexicon. The SL/TL

\(^{32}\) For MT context only, otherwise they may be used for other purposes and hence stored in permanent lexicon
mapping is also stored to temporary Transfer Lexicon. If a pattern qualifies for more than one Named Entities, they are resolved using resolve logic or by asking user. As such this tool does not have any formal UI. The Algorithm for NER marker is given below:

5.3.2.1 Algorithm

Input: Input Text, NER Lexicon

Output: NER Phrase marked Text, NER_Temp_SL_Lexicon, NER_Temp_Transfer_Lexicon.

1. Start
2. Fetch entire NER Lexicon
   2.1 For each NER Phrase \( NP_i \) \((i=1..n)\)
   2.2.1 Try to match NER phrase \( NP_i \), \( \text{MatchRegEx(Input Text, NP}_i) \).
       There are two cases i.e. either match is found or not found
       2.2.2.1 If the match is found, do followings:
           a. Replace the matched phrase in the input text with hyphenated phrase i.e. replace blank space(s) between the words of the matched phrase with hyphen ('-') character.
           b. Store the hyphenated phrase to temporary SL lexicon.
       2.2.2.2 In case NER phrase does not match, move to next NER Phrase.
3. Check for words with multiple NE markings, Resolve such multiple NE markings with resolve logic (priority rankings) or user input. Delete unqualified entries.
4. Transliterate/translate hyphenated phrases on the fly and store them to Transfer Lexicon.
5. Return hyphenated input text for further MT processing.
6. Stop.

5.3.3 Sentence Extractor

The sentence extractor is required for extracting the sentences from input document. The sentence boundary is marked with sentence delimiter symbol generally full stop i.e. ‘.’ (dot) symbol. The job of sentence extraction appears to be trivial but it isn’t, this is due to the fact that full stop (dot symbol) is overloaded and may be confused with abbreviations like ‘Dr.’, ‘Mr.’, ‘Mrs.’ etc., hence we need to take care of such cases, for this reason, we have developed Abbreviation Lexicon, which contains all abbreviations ending with full stop, whenever we find the sentence delimiter, we need to test whether it is valid sentence end or an abbreviation. Depending on the result of this test, we extract sentence(s) from the input document or skip to next matching delimiter symbol. The extracted sentences are returned for further MT process. As such Hindi uses ‘।’ or ‘.’ to mark the sentence delimiter. In case of ‘।’ (khadi paai), sentence extraction can be viewed as trivial activity but if the user uses ‘.’ (full stop), then it is a matter of concern. As such this tool does not have any GUI. It uses following simple algorithm.

5.3.3.1 Algorithm

Input: Input Text, Abbreviation Lexicon.

Output: Array of Sentences Input_Sentences[] extracted from Input Text.

1. Start
2. Initialize search indices i=0, j=0.
3. Fetch entire Abbreviation Lexicon.
4. Initialize sentence_Begining=i
5. Search Input Document for sentence delimiter symbol 
   (most preferably full stop i.e. ‘.’) character. i.e. 
   \[ j = \text{Find(Input Text, } i, \text{ delimiter_symbol)} \]

---

33 Use of dot (‘.’) symbol is observed in modern usages in Hindi Newspapers or literary writings.
5.1 If delimiter is not found \((j=-1)\), Add Input Text to Input_Sentences[] Go to Step 6.

5.2 Extract the word \(w\), preceding delimiter character and check it in the Abbreviations Dictionary, if this word matches, it means this is not valid sentence boundary w.r.t. to delimiter hence skip this delimiter. Update the search index \(i = j+1\). Go to step 5.

5.3 The word is not found in the list means this is a valid sentence boundary, hence extract the sentence from input document,

\[
\text{Sentence} = \text{Substr(InputText(Sentence Beginning, j))}
\]

and add it to the Input_Sentences[] array.

5.4 Update the search index \(i = j+1\). Go to step 4.

6. Return extracted sentences array i.e. Input_Sentences[].

7. Stop.

5.4 Core MT engine Tools

This section describes the core of HinMaT MT. It consists of three parts viz. HinMaT UI, CPG Parser and CPG Generator. UI is a central point where all processing tools are listed in Figure 5.1 above are plugged in. The CPG parser actually parses the phrase marked input document sentence by sentence. The UI is also responsible for showing and holding the extracted input and output sentences as well as their derivation trees. The parsing is carried out on the principles of Paninian framework in two phases i.e. intra-chunk(intra-clausal) and Inter-chunk(inter-clausal) levels as proposed by Prof. Rajeev Sangal (Bharati, Sangal, & Reddy, 2002), (Bharati, et al., 2009) with couple of modifications at our level. Our parsing also uses the fundamental parsing algorithm proposed by Michael Covington (Covington M. , 1990), (Covington M. A., 2001). The Generator algorithm is solely worked out by us. The main features of the parsing and generation are, checking of feature compatibility of words as may
be coded in CPG rules. The following subsections describe all tools under this category in detail.

5.4.1 HinMaT UI

Main UI of HinMaT is inspired from the concept of Integrated Development Environment (IDE), which provides support for all development activities like input coding, compilation, debugging etc. In our case as such users will not do any coding but they will provide input sentences and then initiate the translation process (compilation). We have designed the UI screen, which is broadly divided into three parts: Menu & Toolbar, Input/Output area and debug panel. All supporting tools are integrated into Main UI through Menu & Toolbar. The tool bar shows buttons for invoking manual phrase and automatic phrase marking, sentence extraction, parsing/generation and tree viewer utility etc. Following block diagram (Figure 5.22) shows the schemata of HinMaT UI. Top panel on the UI consists of Menu and Toolbar, while middle portion shows source (SL) input text and translated (TL) output text aligned side by side. The debug panel is located at the bottom; this panel is tabbed panel in which translation details of individual sentences can be viewed in separate tab.

![Figure 5.22 HinMaT UI Layout](image)

5.4.2 HinMaT CPG Parser

Natural Language Parser is a very important tool for the majority of NLP applications and it is considered as the heart of most NLP systems. Role of parser is to analyze the
input sentence to check its’ membership to given natural language using its’ grammar and present the formal proof of this membership using systematic analysis denoted by tree notation called ‘derivation tree’. Parsers are broadly classified as Chart based parsers and Data driven parsers. Each parser uses a parsing algorithm which is specific to particular grammar formalism. Though parsing algorithms are specific to grammar formalism, they are based on certain type of approach called ‘parsing style’ like Earley style, CYK style etc. Good numbers of algorithms are available for Context Free Grammar (CFG) formalism as compared to Dependency Grammar (DG), CFG parsing is also known to be more matured and trivial than DG (Rambow & Joshi, 1997). Since HinMaT uses CPG formalism, deciding the parsing strategy was a big challenge. Hence we surveyed various parsing algorithms available for DG. We found that the DG parsing algorithms can be classified as those based on DG relations, graph models, and data driven (machine learning based). Rule based algorithms can be further sub-classified as stack based and constraints based. Dependency Relation based algorithms perform tests on participating words to validate the dependency. It may apply additional constraints to guarantee appropriateness of dependency relation. Various DG parsing algorithms proposed by Covington (Covington M. A., 2001), Chi-liu-Edmond (Liu Y. J., 1965), (Edmonds, 1967), Lesmo & Lombardo (Lombardo & Lesmo, 1996), Jaokim Nivre (Nivre, et al., 2007), Ryan McDonald (R. McDonald, 2005), (McDonald, 2008), Jason M. Eisner (Eisner, 1996), T.Jarvinen & P. Tapanainen (Jarvinen & P. Tapanainen, 1997) Rajeev Sangal (Sangal & Bharati, 1993) (Bharati, Sangal, & Reddy, 2002), (Bharati, et al., 2009), were thoroughly studied. We have adopted hybrid approach for HinMaT parser, which is based on Covington (Covington M. A., 2001) and Sangal (Sangal & Bharati, 1993), (Bharati, et al., 2009), (Bharati, Husain, Sharma, & Sangal, 2009). Since CPG analysis is based on Karaka theory, verb is at the center stage of parsing activity. Besides its basic lexical feature information, verb also stores, its demand frame, which specifies verb’s argument structure (valences) in terms of Karakas and their Vibhaktis. Detail anatomy of demand frames is explained in section 4.4.4.2 above. CPG views sentence as series of word groups also referred to as local word groups (LWG) (Bharati, Chaitanya, & Sangal, 1995) these are primarily noun groups and a verb group(s). Noun group may be viewed as congruent to noun phrases in CFG, which contain noun and its modifiers (dependents) like adjectives, determiners etc. while verb group consists of main verb
and its modifiers like auxiliary verbs, adverbs etc. These word group units are also referred to as *chunks* (in our implementation they are referred as *Nodes*). There is subtle difference in CFG phrase and chunk(LWG) i.e. chunk is non recursive in nature. So the sentence can now be viewed as sequence of *chunks* and the parsing has to take place at two levels, first the individual chunks are parsed (IntraNode) for dependency, where the chunk heads are identified, after this Head identification, dependency parsing between chunk heads is performed. The internal structure/architecture of HinMaT CPG parser is depicted using following block diagram (Figure 5.23).

**Figure 5.23 HinMaT Parser Architecture**

**Parsing Steps:**

1. **Tokenization**

Pre-processed Phrase marked sentences are fed to the parser sequentially. The Tokenizer module breaks input sentence into valid tokens (words) by separating words using white space characters. It also uses user defined delimiters as well as abbreviation lexicon for marking valid tokens. Besides tokenization Tokenizer module also populates the lexical data for tokens, which is explained in step 2 below.

2. **Populating Token Lexical data (Token Vector)**
After tokenizing the sentence, lexical data associated with these tokens needs to be loaded from SL/TL Word Lexicon for further process, this data includes POS category (as defined in HinMaT POS ontology), syntactic, semantic features and usage domain. A word may appear with more than one POS categories, all entries for the word are populated. This population of tokens results into a jagged array like structure. Since we are not using any POS tagging in HinMaT, we need to try each POS for parsing. So we need to strip out vectors (single column arrays) from this jagged array i.e. we have to generate, multiple Token Vectors by enumerating permutations of POS categories of all words. These Token Vectors are then sent for parsing. It is important to note here that for verb words, demand frame of the verb is loaded from SL/TL Verb Lexicon. Number of token vectors $N$ is determined by following equation:

$$N = \prod_{i=1}^{k} \text{Col}(R_i)$$

Where,

- $\text{Col}(R_i)$: number of columns in $i^{th}$ row ‘$R_i$’ of jagged array (i.e. number of categories associated with word $w_i$),
- $k$: number of rows in jagged array (number words in input sentence)

Following Figure 5.24 illustrates the Token Vector stripping process.

Jagged array is special array construct in which every row in the array can have different number of columns. This kind of storage construct is supported by modern programming languages like JAVA, C#, VB.NET.
3. Chunking

After iterating $N$ Token vectors, the dependency parsing process formally starts, with Chunking. During this process, words are grouped (clustered) on the basis of certain grammatical traits and clues. As stated earlier this process is also referred as local word grouping (LWG) (Bharati, Chaitanya, & Sangal, 1995). Concept of chunk appears analogous to phrase in phrase structured theory but it is important to note here that the chunks are non-recursive, while phrases may be recursive. Our Chunker is in-line with the HinMaT’s philosophy; hence like other tools it has been developed on rule based approach. It uses various syntactic cues such as category of preceding and following words to mark chunk boundaries and chunk the data. Following are the examples of chunk types as given in Annotation Guidelines proposed by IIIT, Hyderabad (Bharati, et al., 2006):

**ADJ**: Adjective chunk contains adjective and its’ modifiers viz. determiners, intensifiers.

**Noun**: Noun and its’ modifiers like adjectives, determiners. Noun chunks are closed on Vibhakti markers and/or particles or the noun itself.

**ADV**: Adverb chunk contains adverbs.

**CONJ**: contains conjunctives like and/or etc.

**SYMB**: contains punctuation marks like dot (‘.’), comma (‘,’), etc.

**Verb**: contains verb and its’ modifiers like auxiliary verbs, adverbs etc.
During chunking process, a chunk may get merged into another chunk like `ADJ` gets merged into `Noun`, if adjective is immediately followed by noun. In such cases merged chunk losses its’ identity.

Chunking process iterates over the input Token Vector sequentially observing POS category of each input token. Chunk formation process is solely governed by POS category of current input token, its successor and current chunk type. It follows a simple idea that current token may be added to current chunk or it may open new chunk by closing current open chunk (if any). The POS category of preceding and following tokens is also consulted during chunk formation as well as for observing the POS sequence. POS sequence is also used for identifying wrong constructions. In case of wrong sentence construction, chunking process is terminated for the current sentence by communicating appropriate error message thereof.

4. Intra-Chunk Dependency parsing (Phase-I)

Phase-I of the dependency parsing starts, after the chunk formation is complete. During this phase, each chunk is parsed for dependency parsing. As proposed by Covington (Covington M. A., 2001), all word pairs are tested for dependency. CPG rules matching the POS category of input words in the word pair are loaded from the SL grammar lexicon. These Rules are applied sequentially by the Rule Applicator, till first successful application. The dependency testing procedure checks the syntactic and semantic feature compatibility of participating words, as encoded in the CPG rule in the form of feature test. The Rule details are explained in Chapter 4, section 4.4.3.1. Test may enforce matching or mismatching of individual feature values or a feature test may not be applicable. In such case, feature test is not performed. If none of the rule application is successful then sentence parsing failure is reported and other strategies like partial parsing etc. are applied in such cases. After successful application of the CPG rule, the input words in the pair are formally bound by the dependency relation and either is nominated as ‘Head’, while other is nominated as ‘Dependent’, as specified in the rule specification. Dependency relation is formally nomenclature by mnemonic label called as `arc_label`. The dependency relations are stored using edge data structure represented by triplet `(Dependent, Head, arc_label)`, where `Dependent`, `Head` words have usual meaning and `arc_label` denotes the dependency relation label. Features of participating tokens are tuned, if required using
the methodology described below. Head of each chunk is nominated and updated as
the parsing progresses. Head count of each word is also updated during parsing
process. It is desired that each word should have maximum one Head so that Chunk
will have only one head but it is a common phenomenon that a word may qualify in
multiple dependency tests as dependent. Head count value of greater than one, for any
word hints multiple parses or Head conflicts or syntactic ambiguity. Such conflicts are
resolved by separate mechanism, which is based on certain semantic features of the
participants. Information for resolving conflicts is encoded in separate rule database.
In case a word does not qualify in any dependency relation i.e. it can’t be parsed with
other words in the chunk, it may have been wrongly included in the chunk or sentence
construction itself may be wrong. In such event unparsed tokens are collected in
separate data structure. If all unparsed words are not sequential in their word order, it
may mean violation of projectivity constraint of DG, Parsing error is flagged during
such cases, as we are right now handling only projective constructions. These
unparsed words lay either at the front (beginning) or rear (end side) of the chunk.
These unparsed words are tried with other chunk Heads in next phase i.e. Phase-II of
parsing. GNP features for each chunk are maintained separately. These feature values
may differ from the GNP features’ values of the chunk root, this happens when
singularity transforms to plurality (E.g. Ram, Shyam and Gita Went to Market). This
may have effect on GNP agreement of chunk with other chunk(s) during Phase-II of
parsing. Successful dependencies are stored using ‘edge’, data structure. If parsing
error flag is raised during parsing of a chunk, parse flag of the chunk is set to false.
Parse flag for the sentence is set to false, if there exists at least one chunk with its
error flag set to true. In case of such event, Word2Word MT is invoked. Otherwise,
parsed chunks are sent for Interchunk dependency parsing i.e. Phase-II.

5. Interchunk Dependency Parsing (Phase-II)

Interchunk (Phase-II) parsing is performed after successful Intra-Chunk parsing. As
described earlier, successful Intra-Chunk parsing results into dependency parsed
chunks for whom their respective unique heads are nominated as well as they may
contain sequential unparsed token chunk(s) (if any). Primary task of Interchunk
parsing phase is to test individual chunk heads of all chunks for dependency parsing
with each other in the same fashion as described above.
The unparsed sequential tokens of each chunk are first tried for dependency parsing with chunk heads of remaining Chunks. Whenever unparsed token is parsed with chunk Head of other Chunk, it is deleted from the list of unparsed tokens. After test iteration of current chunk is over, the parsing process progresses only if chunk Head of the current chunk is parsed with some chunk head of other chunk(s) as well as the unparsed tokens (if any) of the current chunk are also parsed with chunk Head(s) of other chunks without violating the projectivity constraint. Failure of either case hints parsing error and parsing process is halted. After successful parsing of the current chunk, current chunk is then sent for resolving the syntactic ambiguities (head/label conflicts) (if any) followed by routine for building the derivation (parse) tree for current chunk. Dependencies between chunk heads are also stored using edge data structure (as described above), in separate edge set. Like Intra-Chunk parsing, feature tuning routine is called after successfully parsing the chunk and resolving its conflicts (if any). Sentence root is maintained and updated during this parsing phase. After complete iteration of all chunks, this edge set is sent for resolving the syntactic ambiguities. Successful Interchunk dependency parsing results into connected dependency tree with single Head. The edge set is sent to Derivation Tree Builder module for building the derivation tree, described in following section. This tree is sent for TL i.e. Marathi sentence generation to Generator module. If Interchunk dependency parsing is unsuccessful due to some reason, Word2Word MT engine is invoked upon user’s consent. To speed up the parser, we have implemented certain heuristics from linguistic observations like, if a noun appears with Vibhakti symbol (case marker), we need not to check it following nouns, because according to Karaka theory, it will mostly go only with verb. According to Karaka theory, most of the chunk heads are Nouns, parsing at this stage mostly involves checking of chunk heads with demand frame slots of verb. Non Noun-Verb dependencies are tested using CPG rules as described in previous section. Demand frame constraints w.r.t. to Karaka existence specification i.e. mandatory, optional and not-required as denoted in the semantic frame of the verb are checked at the end of parsing. Violation of existence specification for either Karaka results into parsing error. The projectivity constraint is...

\[35\] Chunk head may get parsed with more than one chunk heads due to syntactic ambiguity
\[36\] with couple of exceptions like Generative case and Sahakarakatva (Samanadhikaran) in case of Copula verbs
also tested during this parsing process. Non-projective structures are not permitted in the present implementation.

6. Feature Tuning routine

Feature values are also required to be tuned due to feature overloading feature of languages as discussed in Chapter 3. This routine is used to tune (scale down) generic (broader) feature values of words’ features to specific value in the current context. E.g. consider case वह (this) मेरा (my) घर (house) (is) here features of ‘घर’<M, *, 3rd, D> and ‘वह’ <*, *, 3rd, D> are wider i.e. number feature of both is ‘*’ meaning any value is permissible for number feature, also value of gender feature for word ‘वह’ is ‘*’. We need to specify the exact feature values of ‘घर’ or such words for their correct Marathi translation. In case of ‘वह’, it maps to three Marathi equivalents ‘हा’<M, S, 3rd, D>/<ह> <F, S, 3rd, D>/<हे> <N/M, S/P, 3rd, D>. Hence its’ feature values must be determined (fixed) before starting of generation phrase. These generic (broader) feature values can be scaled (narrowed) down from copular predicate ‘घर’(house) of demonstrative pronoun ‘वह’(that) and features of ‘घर’ (house) can be decided from the its dependent(modifier) word मेरा<M, S, 3rd, D>. Since dependency structures are flat and binary, feature value tuning operates on cascading manner hence this function needs to be called after every successful application of dependency Test as well as after syntactic ambiguity resolution of InterChunk Parsing. Feature tuning is done for all gender, number, person and case features. Feature tuning can be also performed during lexicalization/chunking with the help of syntactic cues by consulting the preceding and successor words.

7. Conflict Resolver

This module performs the most important task of resolving the dependency conflicts. Our thorough study on the parsing revealed that dependency conflicts can be classified into two types: Label conflicts, Head (parent) conflicts, the same are explained below.

37 <G, N, P, C> denotes gender, number, person and case, feature template
**Label Conflicts:**

This type of conflict may arise, whenever same label is used for marking dependency between dependents (more than one) of same head. E.g. consider dependency between A(dependent) and D(head) labelled with ‘k1’ represented using edge data structure as (A, D, ‘k1’) and another dependency, between C(dependent) and D(head) also labelled with ‘k1’ i.e. (C, D,’k1’) here label ‘k1’ is used to mark two dependency relations of same head ‘D’.

However, it is important to note here that this type of conflict is highly label dependent i.e. for few labels like ‘NMod’ i.e. adjectival noun modification; noun is permitted to have multiple adjectival modifications. In nutshell this type of conflict is specific to dependency label under consideration and more prone to verb frame *Karakas* k1 & k2. In case of complex and compound sentences, same label may be repeated between different pair of words in constituent clauses/sentences, in such event they can’t be treated as label conflict.

**Head Conflicts:**

This type of conflict arises, whenever a word participates in more than one dependency relation as ‘dependent’ with same or different ‘Head’. Dependency labels may or may not be same in such cases.

Generally, occurrence of both types of conflicts is common phenomenon. Semantic features play crucial role in resolving such conflicts. Interestingly, in some cases (shared karaka), linguistically such multiple heads are permissible, e.g. राम ने खाना (खाते हुए /खाकर) गीता से फोन पर बात किया (Ram talked to Gita on phone after/while having food). Here राम is subject of both verbs खाना (खाते हुए /खाकर) and बात किया. In such cases first verb is treated as modifier of noun i.e. arrow direction is inverted i.e. from Dependent to Head. These are very limited situations, which are taken care at grammar rule design time.

The conflict resolving mechanism, first resolves the label conflicts and then the Head conflicts. The two types of conflicts may be induced by each other i.e. label conflict may also result into Head conflict or vice-versa. Hence in such cases, we have observed that most of the times resolving label conflicts, may also resolve the Head
conflicts. E.g. suppose we have edges (x, root, ‘k1’), (x, root, ‘k2’), (y, root, ‘k1’), (y, root, ‘k2’). Here based on semantic features, if we resolve the label ‘k1’ in favour of dependent x and label ‘k2’ in favour of y, the edges (x, root, ‘k2’), (y, root, ‘k1’) would get deleted, hence we have only two edges left (i.e. (x, root, ‘k1’) and (y, root, ‘k2’)) in the edge set without any conflict.

For resolving the label conflicts, we have encoded rules in the database, which compute scores for edges (mostly for dependents) based on dependent/Head semantic features as may be encoded in the rule. Edge earning maximum score for a label wins the bid and other conflicting edges for that label are deleted for subsequent analysis. The Head conflicts are generally resolved on the basis of arc labels of conflicting Heads, this happens mostly due to overloading of Vibhakti symbols, for example Vibhakti marker ‘से’ is overloaded with two karakas viz. k3 (instrumental case), k5 (ablative case) as well as used in comparison (karta-samanadhikaran/karma samanadhikaran, Sahakaraktva). These conflicts are handled on case-to-case basis, e.g. suppose, we have two edges (d, H1, ‘lbl1’) and (d, H2, ‘lbl2’), we want to choose a right Head, amongst H1 and H2. We resolve this with the help of labels lbl1 and lbl2. We look into linguistic constraints for dependency relations marked by these labels and from linguistic cues to resolve these. So far we have handled 14 label conflicts between label pairs.

8. Derivation Tree builder

After resolving the dependency conflicts, we arrive at an edge set which is connected and has single Head. This edge set can be used to build the derivation tree or parse tree, which can be used for carrying out the ‘Transfer’ phase of MT architecture. This tree structure encodes the syntactic as well as semantic information within itself. This module builds the tree structure from the edge set and it encodes it in HinMaT’s proprietary notation (format) which is explained below. Since this tree is an n-ary tree, our tree node structure is inspired from 360° view of the node i.e. looking ‘up’, ‘down’, ‘left’ and ‘right’. We look ‘up’ for the parent node, both ‘left’ and ‘right’ represent left sibling and right sibling while ‘down’ side points to children nodes of the current node. The exact structure of node data structure is described below (Pl. see Figure 5.25):
The algorithm for building tree iterates over the entire dependency edge set and upon visiting each edge it builds the tree incrementally using above node data structure. Every build iteration creates appropriate links with siblings, children as well as root, so that resulting sub-structure is a connected tree. In present implementation final tree has single root and no crossings (projective). This derivation/parse tree is marshalled to Generator tool for TL sentence generation. This exact derivation tree is represented using LISP like proprietary notation for post analysis and tree viewer program. Every tree node and its’ subtree is represented using following recursive notation:

\[
Node = (Word: \text{Lexical Category}[<\text{GNPC features}>, <\text{Semantic features}>])(
(LeftChild Node-1), \ldots \, (#) \, (RightChildNode-1), \ldots )
\]

\[
Derivation\ Tree = (Node)^+ \\
\#	ext{: indicates word position of Word (root word of this node),} \\
\text{+: recursion i.e. one or more such nodes in the tree}
\]

5.4.3 HinMaT CPG Generator

The Generator module is responsible for TL sentence generation from the SL parsed sentence. The main function of Generator program is to build the TL parse tree from SL tree. SL parsed tree is said to be transferred to TL parse tree. The yield of this tree is nothing but the TL sentence. Designing of the Generator for CPG (DG) is one of the major contributions of our research.

Generally, major challenges in designing the Generator module are handling of the issues like word order (SOV, SVO, VSO etc.), various divergences as reported in the Chapter-1 such as morphological, syntactic and semantic. Morphological divergences include word formation process divergence and agglutinative nature, Categorial divergences (POS category of SL and TL equivalents are different), GNPC feature...
divergences (GNPC features of SL words and their equivalent TL mappings are different), GNPC specification differences of SL and TL, syntactic/structural divergences, language construct formation (phrase/clause/sentence) differences i.e. internal structure of language constructions for forming sentences from basic constructs. Our Generator is capable of handling the GNPC divergence and word order divergence. The internal architecture of the Generator is given in the following Figure 5.26.

Figure 5.26 CPG Generator’s Internal Architecture

The important blocks of the architecture are explained in detail in following subsections.

5.4.3.1 BFT controller (Derivation Tree Traversal)
The TL parse tree is built incrementally, by traversing and inspecting the SL Parse Tree nodes. Different tree traversal methods such as Breadth First Traversal (BFT) and Depth First Traversal (DFT), InOrder, PreOrder, PostOrder have been prescribed for traversing tree data structure. Choosing either is a design question. Answer to this question depends on nature of the problem solution in terms of processing order. In our case, considering our application and algorithmic solution, we have decided to go for BFT. The justification for our choice is presented below:

1. In our DG parse tree representation, all dependents of a head word are represented at the same hierarchical level, the dependency is syntactic and morphological in nature. Since a sentence is anchored at main verb, its dependents are nouns (Karakas) or adverb group or compliments. At subsequent levels, we have nouns and their modifiers (determiners, adjectives, nouns etc.)

2. In DG formalisms Head is supposed to govern the features of its dependents, however, in case of verb, its morphology is governed by either of Karta Karaka or Karma Karaka or none, and hence it is necessary to process all dependents of a node in single pass so that their cumulative and cascading effects can also be handled properly. Most importantly the GNPC divergences can be elegantly handled this way because whenever there is divergence in any word, it affects its modifiers and in some cases its’ Head word (Noun Karaka as discussed above). Due to recursive nature of tree definition, same is true for words at any level in the tree. This processing order demands BFT method.

3. When we further explored the problem, we observed that as such nodes at grand children level, does not have any direct effect on grand parental nodes. The same fact is endorsed linguistically also as nodes at immediate upper level dominate the nodes at lower level in recursive manner. The modifiers can help to scale the generic feature values for their Head node to specific values, this may have a cascading effect till tree root but it is also very important to note here that this effect is already handled for SL side during SL parsing, but for TL side, it may have some effect in case of scaling down feature values from generic to specific but again this can’t be really very significant and driving factor. Hence option for Depth First Traversal (DFT) was ruled out.
4. Since our problem is to build TL parse tree from SL parse tree, in nutshell, what we have to do is, observe SL parse tree nodes, obtain their TL equivalents and add them to appropriate position in TL parse tree by checking its dependency with parent node (proposed Head word), under whose subtree, it would get added. So building this tree in horizontal incremental fashion is computationally better, also we need not to maintain different variables for levels. This incremental construction hints BFT traversal.

5.4.3.2 Feature Compatibility Test

This test is performed between SL (Hindi) word and its TL (Marathi) mapped words. A word may have multiple mappings in the Transfer Lexicon (dictionary). Those words may be synonymous but disagreeing in their GNP feature values. If there is no divergence at the parent level, the TL option having such divergence should be outright rejected. This is what is done by this testing. This test also helps to scale the generic feature values with the help from SL side, especially in the cases of Genitive case (Sambandhavachak Karaka). Word forms are overloaded with multiple feature templates i.e. word ‘x’ may appear with \(<g1, n1, p1, c1>\) and \(<g1, n2, p1, c2>\) templates as explained in Chapter 3. Exact template may depend upon particular instance and context. This test helps in deciding certain feature values before a token goes for TL dependency parsing. It further helps to avoid wrong outputs.

When, we pondered on the question as to what feature values can be fixed or scaled and how? we observed followings:

i) Hindi Common noun forms are ambiguous for most of the Common Nouns, \(<M, P, 3^{rd}, D>\) word form is same as \(<M, S, 3^{rd}, Obl>\), e.g. लड़के (boys), बचे (kids).

ii) Mostly, Hindi masculine nouns are mapped to Neuter gender in Marathi.

iii) Few Marathi Nouns are also ambiguous in terms of their feature specification e.g. मुल (girl) \(<F, S/P, 3^{rd}, D/Obl>\) can represent plural direct-case \(<F, P, 3^{rd}, D>\), singular direct-case \(<F, S, 3^{rd}, D>\) as well as Singular Oblique \(<F, S, 3^{rd}, Obl>\) cases, which is not observed in Hindi there are mostly separate word forms for singular-direct and plural-direct cases.
iv) Number specifications are also ambiguous in Marathi. For genitive
(Sambandhvachak karaka i.e. r-6) Vibhakti symbols.

E.g.

a> ची <FN, *, 3rd, GD> actually stands for <F, S, 3rd, GD> (चा मुलगी
   ची/हुडी + <N, P, 3rd, GD>(चा मुले))

b> चा <M, S, 3rd, GD> actually stands for = <M, S, 3rd, GD>(चा मुलगा/काका)

b> चे <MN, *, 3rd, GD> actually stands for = <M, P, 3rd, GD>(चे भाऊ/काका)


v) After having come across these interesting observations, there are some
questions from computational points of view, the question here is how can
we choose the right feature template or fix feature values from the Hindi
word? Our study revealed that case feature can be used primarily to accept
or reject a target word, if case does not match (exception: *, GDO), we
reject TL word under consideration. For */GDO/GD/GO case values, we
set the case feature from case value of SL word (equivalent of TL).

vi) Coming back to the question of what feature values can be fixed(scaled)
and how, before doing TL dependency parsing, We did in depth analysis
for each of the GNPC features in isolation as well as in pairs and found
followings:

a> Case feature:

We can scale ‘case’ feature value in case generic specifications like
‘*’/’GDO’.

---

38 GD-genitive direct i.e. in शामचा भाऊ (Sham’s brother), here word ‘भाऊ’ (brother) appears in direct
case as it is not followed by any case marker, if it is then the case of ‘चा (of) is marked as GO i.e.
genitive oblique for which there is different word form चा.
'*' value is directly set it to source word, as ‘*’ value is used for all POS categories other than Vib.Krk.rel and Genitive Pronouns. ‘GDO’ is generic specification for genitive pronouns and genitive case markers tagged with Vib.Krk.rel

POS symbol, they represent two cases GD (genitive direct) and GO (genitive Oblique)

\[ \text{GDO} = \text{GD} + \text{GO}, \]

GD also stands for various combinations of Gender, Number values like MSD, MPD, NSD, NPD, FSD, FPD. Co-incidently these values are overlapped for different gender/number combinations like <MP/NS, 3\(^{rd}\), GD>, <FS/NP, 3\(^{rd}\), GD>, <FP, 3\(^{rd}\), GD>, <F*/M*, N*, GO> hence after fixing the GD except <F, P, 3\(^{rd}\), GD> case, we can't fix number values for other GDs, even if they are directly coming from source word. This is because number value of 'S' or 'P' are bound/dependent to/on different gender values like 'M', 'N', 'F' which have overlapping, hence just by looking at Gender value of source word, we can't decide the number and gender. Also there may be divergence in Gender from source to target hence this scaling may prove to be wrong one during dependency parsing. Our general observation is that in maximum cases gender divergence is observed between Hindi-Marathi pair for nouns i.e. Hindi masculine nouns are mapped to neutral gender values in Marathi. For Hindi Feminine gender nouns, only couple of examples of feminine to masculine gender divergence have been cited so far (Hindi word डेल्फिन <feminine>(shir), Marathi word डेल्फिन <masculine>(shirt)). We should leave this fixing up of Number values to dependency parsing also for GDO separation to either GD or GO, we must scale it properly for GD to <F, P, 3\(^{rd}\), D> and for GO <*, *, 3\(^{rd}\), GO> so that there are no overlapping.

E.g.: ची <F/N, S/P, 3\(^{rd}\), GD>: ची रामाची मुलगी (Ram’s daughter), ची रामाची मुले (Ram’s childern), ची झाडची फुले (tree’s flowers)

\(^{39} \text{GDO- genitive direct as well as oblique.} \)
b> ‘Gender’ feature:

Value of Gender can’t be fixed for two reasons.

**Reason1:** if there is gender divergence from source to target, we can’t fix the gender feature value as it is from SL word. Secondly, SL word may have generic feature value (‘*’), so we can’t fix the value.

E.g. English: baby Boy(M) = Hindi: बचा (M) = Marathi: मुल (N),

English: Tree(N) = Hindi: पौड़ (M) = Marathi: झाड (N),

English: Flower (N) = Hindi: फूल (M) = Marathi: फुल (N),

यह [this] (*) = हा (M/S/D)/हे (M/P,N/P,GD)/ह (F/S/NP,DG)/या (F/P/GD, */*/GO)

**Reason2:** We may set the value, but if root word of this word may be dependent of some other word and that word’s mapping word may have Gender divergence so it may break morphological dependencies. Even if we scale values that word form may/will not be exact required word form, and what is the divergent gender is known only to the Head word its TL mapped word, but dependency for that word is not yet tested so we can’t decide for such cases we obtain appropriate morph(word) form उदा मेर कमीज़ (my shirt). मेर (F) = माझी (F) कमीज़ (F) = सदरा (M) नमुक्त (F) != माझा (M) not possible, hence there is no point in setting Gender value or otherwise just set it and let it get rejected in dependency test upon divergence.

c> ‘Number’ feature:
We have observed that Number feature value can be fixed in most of the cases. Following are the two scenarios to justify our claim.

**Case-1:**
Number may have ‘*’ value on target side, in such case we can decide its value from the SL word’s number provided SL word does not have Number divergence, in such cases we can use SL word’s number value. If case feature value is * or GDO, we will fix it from the SL word’s case or Marathi language characteristics value of ‘D’ for case value mostly leads to value ‘P’.

E.g. Hindi: लड़कयाँ (M, P, 3rd, D) [girls] = Marathi: मुल (F, *, 3rd, *), here SL word case value ‘D’ triggers that TL word number will have ‘P’ value i.e. लड़कयाँ (M, P, 3rd, D) = मुल (F, *→P, 3rd, *→D). This type of feature template overloading (<F, S, 3rd, Obl> and <F, P, 3rd, D>) is common for Marathi Common Nouns appearing in feminine gender, the same is discussed in detail in Chapter 3.

More examples are:

This does not hold in common for Marathi neutral and masculine gender values. As separate word forms are used for denoting direct and oblique cases for masculine gender and so does for ‘plural’ number, e.g. मुला <N, S, 3rd, D>, मुलगा <M, S, 3rd, D>[boy], मुले <N, P, 3rd, D>[boys], झाड <N, S, 3rd, D>[tree], झाडे <N, P, 3rd, D>[trees], झाडा <N, S, 3rd, Obl>, झाडां <N, P, 3rd, Obl>[tree].

**Person** feature:

For person feature, we have not come across such cases, as there is no divergence for person category between Hindi and Marathi pronouns.
and Marathi pronouns are more diversified than Hindi. So we need not fix the value of person feature.

vii) Similar, thing is true for Verb forms also. The verb form overloading is much more complex than non-verb word overloading as discussed in the Chapter 3.

In present implementation 55 GNP feature overloadings under different TAM features have been handled neatly by HinMaT.

5.4.3.3 Dependency Testing

Since HinMaT is based on Shake & Bake MT paradigm, TL side parsing is a must. It solves the problem of word ordering as well as divergence handling. The TL Grammar rule ID is obtained from Transfer Grammar with the help of SL Grammar rule ID, which is encoded in SL parse tree. For Noun-Verb word pair, dependency is checked as per the demand frame constraints, while for other POS categories, dependency testing is performed as encoded in TL grammar rule. SL parsing information such as ‘arc label’, dependent side is useful for TL parsing.

5.4.3.4 Divergence handling

HinMaT is capable of handling GNPC and categorical divergences. The divergence information is coded in the transfer lexicon mapping itself. So whenever a TL word with divergence appears, divergence handling routine takes over the process.

For GNPC divergence, whenever we come across divergent word, we search for appropriate feature compatible word form for its morphological dependents. These dependents are then tried for TL dependency parsing. GNPC divergence has two fold effect in dependency parsing i.e. it affects the GNPC agreement of dependents as well as Head word of divergent word. The effect may be cascaded till sentence root. The latter is possible whenever divergent word is a noun appearing in either Karta or Karma Karaka and the verb form for such sentence agrees with either and masculine to neutral gender divergence is quite common in Hindi-Marathi pair. E.g. consider Hindi sentence for, बचा<M, S, 3rd, D> खेल रहा है<M, S, 3rd, D>। (The boy is playing), मुलं<N, S, 3rd, D> खेळते<N, S, 3rd, D>आहे. Here the Hindi word बचा<M, S, 3rd, D> appears in masculine gender and verb form of play i.e. ‘खेल’ रहा<M, S, 3rd, D> agrees
with it hence appears with \(<M, S, 3^{rd}, D>\) feature template. But on Marathi side the equivalent word \(मुलं\) appears in neutral gender, hence, we need to obtain neutral gender matching form for Marathi verb phrase, which is \(खेळते आहे<N, S, 3^{rd}, D>\) [is playing]. But standard dictionary mapping for Hindi verb phrase \(खेल रहा है<M, S, 3^{rd}, D>\) [is playing] is \(खेळतो आहे<M, S, 3^{rd}, D>\) [is playing]. Using standard mapping will yield wrong translation i.e. \(मुलं \)खेळतो \(आहे\) (wrong agreement).

For handling categorical divergence cases, we simply fetch rules matching in the POS categories of participating words directly from the TL grammar if they are not present in Transfer grammar. Generally such mappings are not stored in transfer grammar because either their occurrence is very rare or secondly they may qualify for unwanted cases also\(^{40}\). Though HinMaT has provision for storing such rules. E.g. Let’s consider a input sentence \(xyz\) in which tokens appears with some categories as \(x\) (cat:p1) \(y\) (cat:p2) \(z\) (cat:p3), we would expect that transfer lexicon mappings for \(x \rightarrow Tx\) (cat:p1), \(y \rightarrow Ty\) (cat:p2) and \(z \rightarrow Tz\) (cat:p3) with same SL POS categories. But if either has different category, we say that categorical divergence as occurred. This type of divergence can lead to thematic divergence in which semantic roles(Karta, Karma etc.) may change to other. This type of change may affect the verb agreement also. Such type of agreement divergence is recorded along with verb forms, while storing them to Verb_Transfer Lexicon.

5.4.3.5 Target Sentence Generation

Our dependency parse tree encodes grammatical as well as lexical information. The leaf nodes represent words, while other nodes encode grammatical information. Yield of the tree i.e. all leaf nodes in left to right sequential order represent the parsed/translated sentence. The algorithm for generating TL sentence traverses the tree and builds the target sentence. This algorithm is recursive in nature.

5.4.3.6 Generator Algorithm

The algorithm used for implementing the Generator program is presented below:

\(^{40}\) Ids of SL rules are encoded during SL parsing. If these rules have divergent rule mappings, they will also qualify regardless of they are applicable or not.
Generator(SL_Parse_Tree, Transfer_Lexicon, Transfer_Grammar)

Input: SL_Parse_Tree, Transfer_Lexicon, Transfer_Grammar

Output: Array of Translated Sentences

1. Start
2. Check whether source side Agreement noun (either K1 or K2) has divergent mapping on TL side? If 'yes', obtain compatible TL Root Word (Verb) forms. Add it to TL_Words[] array and goto step 4, else goto step 3. Multiple entries in TL_Words[] array hint multiple output sentences.

3. Visit the Root node of SL_Parse_Tree and obtain the SL(Hindi) word from the root node
   
   \[
   \text{SL}\_\text{Root} = \text{SL}\_\text{Parse}\_\text{Tree}.\text{RootNo}de;
   \]
   
   \[
   \text{SL}\_\text{word} = \text{SL}\_\text{Root}.\text{SLword};
   \]
   
   Queue.Push(SL_Root)

3. Populate equivalent TL (Marathi) word(s) of the SL_Word from SL-TL Transfer Lexicon (Hindi-Marathi).
   
   \[
   \text{TL}\_\text{Words[]} = \text{Transfer}\_\text{Lexicon}(\text{SL}\_\text{Word})
   \]

4. We need to iterate over each TL_word in the TL_Words[] array (each option at this level means multiple TL output sentences)

4.1. Check Feature_Compatibility between TL_word and SL_Word, features should be compatible, unless feature divergence is flagged in the Transfer_Lexicon. This compatibility testing function also fixes the generic feature values of TL_word to specific ones, features like case, gender, number play important role in this type of checking.

In case of SL and TL words are not compatible, skip the current TL_word go to step 4, in otherwise case do following:
5. We need to store the mapping of SL node and TL node to Dictionary data structure. As reference to TL node is crucial for building the TL_Derivation tree of TL node, when it is selected in BST.

```
Dict.Add(SL_Root, Trut);
```

6. Now that the root node is traversed, we need to traverse the remaining SL_parse tree in BST fashion by first traversing the left children and then right children in left to right linear order. Queue data structure is most suitable for implementing BST traversal. We have to maintain ‘Trut’ variable for target(TL) side node corresponding to SL derivation node.

7. To traverse SL_Parse subtree for a node, pop it from Queue's Front side. Hold its reference to some local tree node variable say ‘rut’. We also need to have reference of ‘Trut’, i.e. corresponding TL side node for ‘rut’, which is obtained from Dictionary object

```
rut = Queue.pop();

Trut = Dict(rut);
```

8. For every Child node 'running' of rut in SL_Parse tree do followings:

8.1 If running = null Then

```
running = rut.Leftchild(); //for first time only
```

```
Queue.Push(SL_Child_Node running);
```
8.2 Populate the equivalent TL (Marathi) word(s) (there may be more than one mappings for given SL word in TL) from SL (Hindi)-TL (Marathi) Transfer Lexicon. Also create new TL_PrsTree_Node (If mapping SL to TL mapping is not present in the Transfer Lexicon SL word is passed as it is to TL side).

\[ TL_{Words[]} = \text{Transfer\_Lexicon}(SL_{Word}) \]

//Create TL_Parse_Node

TLcurrent_ = new TL_DerivationNode()

8.3 Iterate over each TL_word in the TL_Words[] array.

8.3.1 Check compatibility as described above for TL_word with its SL equivalent.

8.3.1.1 skip TL_word in case of failure of compatibility test and go to 8.3.

8.3.1.2 Compatibility test is successful, Since HinMaT is based on Shake & Bake paradigm hence, we need to check the dependency of TL_word with its parent TL_word, previously created TL_PrsTree_Node's word (proposed root for TL_word).

8.3.1.3 Test dependency between TL_word previously created TL_PrsTree_Node's word with the help of Transfer Grammar and SL/TL CPG Grammar rules.

//Following issues occur and resolved,

8.3.1.3.1 Handling of GNPC divergence case:

If rut node's SL word's equivalent containing TL_word (Trut.TL_Word) is marked with divergence flag (GNPc) and it is required to be feature compatible with TL_word, obtain feature compatible TL_word from Transfer_Lexicon
8.3.1.4 If the dependency test is successful, Append TL_word to 'TLcurrent'

\[ \text{TLcurrent.word} = \text{TL_word} \]

8.4 Now that all TL_words[] are exhausted, Add Current node to its parent node of the TL_Parse Tree i.e. Current node should be added to the appropriate side as encoded in the dependency rule. Also we need to store the mapping of 'TLcurrent' and 'running' as it is required when 'running' will be popped off the queue. This mapping is stored in the Dictionary data structure.

\[ \text{Dict.Add(running, TLcurrent);} \]

8.5 Move to the sibling node (next left or next right in left to right order) in the SL_parse tree.

\[ \text{running = running.NextSibling();} \]

Go to step 8.1;

9. Now that all sibling SL_Child_Nodes at particular level are traversed, we need to start with next level, hence we need to pop element from the Queue's front. Same is done in step-8,

Go to step 7.

10. All nodes in the SL Parse tree are now traversed, which is indicated by empty Queue. We should generate the TL(Marathi) sentence by obtaining the yield of the derivation tree. We must also log the entire Translation cycle data through Data_LOGGER Utility which includes SL/TL parse Trees, Parsing/Generation Time statistics, Token Vectors.

\[ \text{TL_Sentences[] = GenerateTLSentence(Trut)} \]

Call Data_LOGGER Utility(All Intermediate Processing Data);

Go to step 4(Process Next node in BFT order).
5.5 Post-processing Tools

Like Pre-processing, Post-processing is an equally important phase of MT life cycle. Post-processing requirements may vary depending upon our abstraction and who is the consumer? Consumers may include, actual user who has run the MT cycle, computational linguist, the developer and the MT system itself.

From end user point of view, post editing is the most important task in this phase. Post editing generally involve word reordering, word(s) addition, deletion and selection from dictionary or manual entry.

From linguistic point of view Post-processing tasks involves the analysis of SL parsed trees and TL generated parsed trees and importantly MT evaluation. From MT system point of view, MT input, output sentence pairs may be stored for future reference as Translation Memory. Parsed/Generated structures may be used for Tree bank building purpose. From MT s/w developer’s point of view, all intermediate data along with time and memory resource requirements should be stored for error analysis, further functional enhancement and efficiency improvement purposes.

These tasks may be performed automatically or manually or both partially. This section discusses various such tools, which were developed for HinMaT MT system. These include Post editing utility, Karaka Disambiguator, SL/TL Parse Tree Viewer Utility, and MT process data logger & Derivation Tree data logger.

5.5.1 Post-editing utility

This has been integrated in the HinMaT UI. It allows to choose appropriate word from the list of options, whenever there is one to many mapping between SL (Hindi) to TL (Marathi) words. The TL words having such multiple translations are shown in highlighted and different color in the translated output sentence. The list of options is popped upon hovering of the mouse pointer over such words. Word reordering has to be done manually only.
5.5.2 SL/TL Parse Tree Viewer Utility

The need for this tool was felt due to the very fact that the derivation tree can be better understood and analyzed, if it is presented using graphical form. The Tree Viewer utility does the same thing. It displays the Hindi and Marathi derivation trees of the input document sentences as well as entire Tree Bank data. The tool first loads the parse (derivation) tree string represented using HinMaT’s proprietary format, into the Tree data structure designed for this utility. The tool allows configuration of visual appearance of tree by customizing display of tree nodes and arcs using different colors for leaf and non-leaf nodes of the derivation tree. When connected to the Tree bank data, one can navigate through various sentences in the Tree Bank. If the tree spans are large, tool provides facility to scroll the view. The crux of the problem in visual representation is to fix the \((x, y)\) co-ordinates of the nodes. The trees depth i.e. height (farthest leaf node from tree root) and tree width i.e. distance between leftmost leaf node and rightmost leaf node are used to decide the \((x, y)\) co-ordinates of the nodes.

The tool allows simultaneous (next to each other) displaying of input Hindi (SL) as well as translated Marathi (TL) sentences’ derivation trees. Most importantly, the tool also performs comparison of SL and TL derivation trees, which is represented visually. In visual representation matching parts (partial structures) of derivation tree are shown using green color nodes while mismatching parts are shown using red color.

User can select color options for tree links and tree nodes. The tree can be viewed in hierarchical tree mode or as dependency graph mode. In the tree mode POS categories represent intermediate nodes, while words are shown at leaf node level. The dependency graph notation is flat representation, in which POS categories are not shown, only sentence is shown and arcs amongst the words are drawn above words.

Another notable feature of this tool is that, we can save the graphical forms of trees in \(.pdf\) or \(.jpg\) formats.

5.5.2.1 Algorithm

1. Start
2. Load Derivation Tree in HinMaT’s proprietary notation in to Tree Data Structure as discussed above. This process also computes the Height (maximum distance between root and deepest node) and Width (maximum distance between leftmost and rightmost node) of Tree.

3. Initialize the Horizontal_Spacing and Vertical_Spacing variables. Horizontal_Spacing denotes distance between horizontal nodes at a level, while Vertical_Spacing denote vertical distance between nodes of two levels.

4. Start Derivation Tree Traversal from Root Node. Initialize the \((x, y)\) co-ordinate values as
   
   Root Node = (Screen width/2, 1)

5. Compute the \((x, y)\) co-ordinate values of Derivation Tree nodes using following process

5.a For any Parent node(root or intermediate), set 
   
   \((x, y)\) co-ordinate values of its left Children nodes as:

   \[
   \text{Left}_\text{Node}.x = \text{Left}_\text{Node}.\text{Parent}_\text{Node}.x - \left(\text{Horizontal}_\text{Spacing} \times \text{Left}_\text{offset}\right)
   \]

   (i.e. \text{Left}_\text{offset} for left child Node is its cardinal rank amongst the left siblings, the left siblings are numbered \(n\) to 1 from left to right i.e. value 1 means it’s a last left child of parent node amongst all left child nodes, while maximum value of \(n\) indicates, it’s leftmost(first) child node)

   \[
   \text{Current}_\text{Node}.y = \text{Current}_\text{Node}.\text{Parent}_\text{Node}.y + \text{Vertical}_\text{Spacing}
   \]

5.b Set the \((x, y)\) co-ordinate values for the right children nodes of any parent node as:
Right_Node.x = Parent_Node.x + Horizontal_Spacing * Right_offset

(i.e. its' number amongst the right siblings, all right siblings are numbered 1 to n from left to right, i.e. value of 1 means current node is first right node amongst the right nodes and maximum value of n means this is n\text{th} i.e. rightmost child node of parent node)

Right_Node.y = Current_Node.Parent_Node.y + Vertical_Spacing

6. Stop

5.5.2.2 Tool UI

The screen shots of this utility in different operational mode are shown in following Figure 5.27a, Figure 5.27b.
A sentence has to undergo various stages before it is being translated in HinMaT framework. Sometimes a sentence may or may not get translated due to various syntactic or semantic errors. We need to debug the translation cycle and carry out the error analysis for such cases. The intermediate data at various phases is very crucial resource for doing such work. Hence we have developed this utility, which logs intermediate data pertaining to the translation cycle. This data includes output from various tools of MT pipeline such as sentence extraction data, lexicalization of input tokens, token vectors, output of chunking (local word grouping), parsing process details, parser’s output i.e. derivation tree, intermediate details of generation process and MT output as well as derived tree after generation. This data is presently logged to a text file, as well as database table. The time statistics like time required for entire translation process, time required for parsing of a sentence and generation are also logged separately. The creation of this intermediate data is optional. The user has the freedom to choose this option by running HinMaT engine in ‘debug’ mode. In ‘release mode’ this data is not generated by the framework. This data has proved to be very useful, in fixing bugs and fine tuning the internal operations of the MT system. The same data has been exploited for creating Tree Bank for Hindi and Marathi sentences.
Tree Bank is very crucial resource for modern NLP applications. It consists of parsed sentences with their parse structures (derivation tree form). The Tree Bank sentences may be manually paper parsed or generated by NL parser programs. In HinMaT framework, our Tree Bank is of later type. Tree Bank data is created and stored for every input sentence tried in HinMaT framework. Tree Bank data consists of derivation (parse) trees of parsed Hindi/Marathi sentences, the token vectors showing sentence tokens (words) and their POS categories in HinMaT POS ontology. It also stores the frequency of sentence occurrences.

We have stored the Tree Bank data at two levels i.e. instance level (specific sentence at an instance) and generic (instance independent) level.

**Instance Level**

In this type, derivation (parse) trees of parsed Hindi and Marathi sentences (generated by HinMaT), the token vectors showing sentence tokens (words) and their POS categories in HinMaT POS ontology and MT cycle time statistics (lexicalization/parsing/generation) along with the frequency of sentence occurrences are stored for error analysis and future reference. This data can serve as Translation Memory for HinMaT. Another potential use of this data can be made in developing an EBMT/SMT system for Hindi-Marathi pair.

**Generic Level**

In instance independent i.e. ‘generic’ level representation, parse trees are stored using a format in which actual word tokens are replaced by ‘unknown’ tokens represented by ‘??’ (double Question mark) symbol. The advantage of such representation is that, it further compresses the sentence tree bank data and it caters for unique parse structures i.e. sentences based on same parse structure. Such structures would be stored separately in the tree bank but their generic parse tree representation would have only single instance. The token vectors are also stored in instance as well as generic level. The token vector data can be a good source for statistical POS taggers. This data can be used effectively by HinMaT on every input sentence, where we can match the token vectors of input sentence with Tree Bank and directly output the derivation trees, thus bypassing the compute intensive processes like parsing and generation. This would speed up the translation process dramatically as translation can be done using simple lookup and replacement operations. Another potential use of this
data can be made to develop hybrid SMT/EBMT systems. Instance specific words can be introduced by replacing ‘??’ symbol with actual words, matching the POS category and feature specifications.

5.6 Summary

We have described our major research contribution through this chapter, i.e. designing and development of practically working framework. Resource creation and designing of the MT engine are two prime aspects of the HinMaT framework. The tools required for both tasks have been described in detail in this chapter. We have also explained our common philosophy for development of tools along with UI screens where ever appropriate. Detail discussion on DG parsing and internal design of HinMaT parser has been presented along with generation process details of TL parse tree and TL sentence. Minute issues encountered during SL-TL syntactic transfer have been covered at greater length. Algorithms of various Core MT engine tools as well as other tools have also been presented. All tools have been designed and developed from scratch.