Abstract. Pāṇini's *Aṣṭādhyāyī* is often compared to a computer program for its rigour and coverage of the then prevalent Sanskrit language. The emergence of computer science has given a new dimension to the Pāṇinian studies as is evident from the recent efforts by Mishra [**?**], Hyman [**?**] and Scharf [**?**]. Ours is an attempt to discover programming concepts, techniques and paradigms employed by Pāṇini. We discuss how the three sūtras: *pūrvatrāsiddham* 8.2.1, *asiddhavad atrābhāt* 6.4.22, and *ṣatvatukor asiddhaḥ* 6.1.86 play a major role in the ordering of the sūtras and provide a model which can be best described with privacy of data spaces. For conflict resolution, we use two criteria: utsarga-apavāda relation between sūtras, and the word integrity principle. However, this needs further revision. The implementation is still in progress. The current implementation of inflectional morphology to derive a speech form is discussed in detail.

Key words:
Pāṇini, *Aṣṭādhyāyī*, Computer Simulation, Conflict Resolution, Event Driven Programming, Task Parallelism

1 Introduction

Pāṇini's *Aṣṭādhyāyī* is often compared to a computer program for its rigour and coverage of the then prevalent Sanskrit language. It is well known that the *Aṣṭādhyāyī* is not as formal as a computer program. But at the same time, we find many features of modern day computer programming in it. This is a quest to discover programming concepts, techniques and paradigms employed by Pāṇini in writing a grammar for Sanskrit in the form of the *Aṣṭādhyāyī*. In order to do so, we try to 'simulate' the grammar following modern programming practices. We hope, in the process, we may encounter something that may lead to new programming concepts altogether.
As has been rightly pointed out by Scharf[**?**], it is very much crucial to define

what exactly one is going to simulate: is it the *Aṣṭādhyāyī* alone or the *Aṣṭādhyāyī* and *vārtikās* or the grammar as described by Pātañjali in his *Mahābhāṣya*? At this point in time, we are still open, however our efforts will be to restrict ourselves to the *Aṣṭādhyāyī* as far as possible.
Within the *Aṣṭādhyāyī* itself, sūtras are interpreted in more than one way, there are controversies over the domain of adhikāra sūtras, etc. The sūtras being too many, it is not within the reach of a human being to ensure the consistency with different interpretations. It is therefore a challenge for computer scientists to design a system that simulates the *Aṣṭādhyāyī* and at the same time provide a facility to test the consistency of the whole system for the chosen hypothesis/interpretation.
The paper has been organised as follows: In section 2, we describe the earlier efforts that deal with the implementation and principles of the *Aṣṭādhyāyī*. In section 3, we describe Pāṇini's process from a programming perspective. The role of three sūtras: *pūrvatrāsiddham* 8.2.1, *asiddhavad atrābhāt* 6.4.22, and *ṣatvatukor asiddhaḥ* 6.1.86 in the ordering of the sūtras is discussed. The discussion of these sūtras lead to a model that can be best described with privacy of data spaces. Section 4 deals with the actual implementation, in particular modules for automatic rule triggering and conflict resolution. Challenges, exceptions and problems are described in section 5,6 and 7 respectively. Finally, we discuss future work and give an example of derivation of *rāmāṇām*, genitive plural of the nominal stem *rāma*.

2 Earlier efforts

There have been attempts to model Pāṇini's *Aṣṭādhyāyī* by Mishra[**?**] and Scharf[**?**]. Mishra has proposed a structure for developing a lexicon on Pāṇinian principles. His system shows the rules involved in the morpho-phonemic derivation using manually developed lexicon rich with feature structure. This may serve as a good model to discover different attributes of the lexicon used by Pāṇini, especially the non-formal or extra linguistic features which Pāṇini has used. But this system in its true sense is not the simulation of the *Aṣṭādhyāyī* as it does not give any insight about choice and ordering of the rules in the derivation process.
Scharf has implemented sandhi, nominal declension and verbal conjugation, closely following Pāṇinian rules. However, in his implementation of nominal declension also the rules for different stems are selected and ordered manually. He himself admits that "... it is not a close model of Paninian procedure"[**?**]. In a sense, his implementation of noun declensions is closer to the arrangement of sūtras in the *Siddhāntakaumudī*[**?**]. It does not give us a flavour of the Pāṇinian arrangement of rules.
If one decides to simulate the *Aṣṭādhyāyī* on computer, two important questions one needs to answer are:

- How are the rules triggered?
- If more than one rule is triggered then how is the conflict resolved?

Tradition has extensive literature on conflict resolution. The *Paribhāṣenduśekhara* of Nāgeśa Bhaṭṭa discusses many *paribhāṣā*s ('metarules') that are necessary to understand the interaction of rules, and conflict resolution. The major paribhāṣā dealing with conflict resolution is

paranityāntaraṅgāpavādānām uttarottaraṃ balīyaḥ

Kiparsky[**?**] explains how the principle of economy (*lāghava*) leads to the concepts of normal blocking and collective blocking which govern conflict resolution.

3 *Aṣṭādhyāyī*: A Programming Perspective

The whole purpose of the Pāṇinian enterprise is to model the communication process. The communication process has two parts – the speaker expressing his thoughts through a language string (generation) and the listener understanding the language string (analysis). Pāṇini's *Aṣṭādhyāyī* models the generation. The process of generation is further assumed to have intermediate levels as in the following figure (Bharati et al.[**?**]: 63).

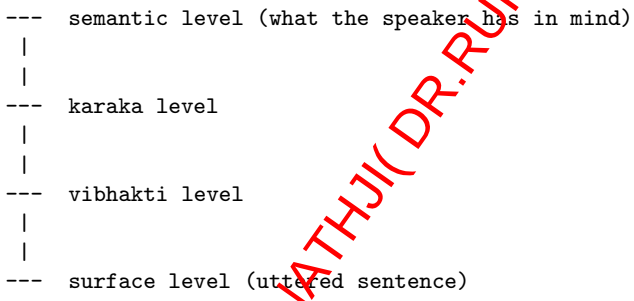

Fig. 1. Levels in the Paninian model

Thus the input for the *Aṣṭādhyāyī* is the semantic level description of thoughts in the speaker's mind in terms of the Sanskrit lexicon, and the output is the sentence in Sanskrit which conveys the same thoughts within the constraints of the language.

The sūtras in the *Aṣṭādhyāyī* are broadly classified into the fillowing six types:[4]

[4] *saṃjñā ca paribhāṣā ca vidhir niyama eva ca*
atideśo'dhikāraś ca ṣaḍvidham sūtralakṣaṇam

1. *saṃjñā*
2. *paribhāṣā*
3. *vidhi*
4. *niyama*
5. *atideśa*
6. *adhikāra*

Most of the rules in the *Aṣṭādhyāyī* are of the type *vidhi*.
Traditionally recognized divisions of the *Aṣṭādhyāyī*, that is, eight *adhyāyas* each divided into four *pādas*, do not, in general, mark off the topics found therein. The topics dealt with in each of the eight adhyāyas are as follows:

- The first adhyāya serves as an introduction to the work in that it contains
  1. The definitions of the majority of the technical terms used in the work,
  2. Most of the metarules, and
  3. Some operational rules.

  The chapter contains rules mainly related to the indicators (*it*) (1.3.2-1.3.9), assignment of *ātmanepada* and *parasmaipada* suffixes (1.3.12-1.3.93), *kāraka* definitions (1.4.23-1.4.55) and *nipātas* (1.4.56-1.4.97).
- The second adhyāya contains four major divisions:
  1. Rules of compounding (2.1.3-2.2.38)
  2. Assignment of cases (2.3.1-2.1.73)
  3. Number and gender of compounds (2.4.1-2.4.31)
  4. *luk* elision (2.4.58-2.4.84)
- The third, fourth and fifth adhyāyas deal with suffixes. The third adhyāya contains the suffixes that are added to a verbal root, while the other two contain the suffixes that are added to a nominal stem.
  In short, the first five adhyāyas build the basic infrastructure that is necessary to set up the proper environment to carry out the derivations smoothly.
- The sixth, seventh and eighth adhyāyas deal with the sūtras that bring about a series of transformations related to continuous text (*samhitā*), word accent, and stem shape.

3.1 Data + Algorithm = Program

The main body of the *Aṣṭādhyāyī* is called the *sūtrapāṭha* (set of rules) and consists of around 4,000 rules. It is accompanied by five ancillary texts: the 14 *pratyāhāra sūtras*, the *dhātupāṭha* (list of verbal roots), the *gaṇapāṭha* (several collections of particular nominal stems), *uṇādi sūtras*, and *liṅgānuśāsana* (sūtras describing the gender of different words). Thus we see a clear separation of data (the ancillary texts) from the algorithms (sūtras).

3.2 Data Encapsulation

The third generation languages were based on the fundamental concept of Data + Algorithm = Program. On the other hand, in Object Oriented Programming

there is an encapsulation of data and algorithms in the form of objects. Both these aspects are not contradictory to each other. We require the separation of extensive data from algorithms. At the same time in certain cases, we also require the active binding of data with procedures. In Pāṇini's system we notice an intelligent use of both these features. As reported earlier, there is a clear separation of data from algorithms. At the same time, the data in the database consisting of the ancillary texts is encapsulated. All the indicators used by Pāṇini trigger some functions: For example, the indicators that accompany each root (*dhātu*) in the *dhātupāṭha* mark the dhātu as either ātmanepada or parasmaipada, the *ñi* indicator in verbal roots indicates that such a root takes the suffix *kta* (which is otherwise a past passive participle) in the sense of present tense, as in

ñidhṛṣā + kta - > dhṛṣṭa

3.3 Subroutines

The rules related to a particular task are grouped together. For example, consider the following sūtras which identify sounds used as markers (*anubandha*) in the texts that comprise the grammar.

- *upadeśe aj anunāsika it* 1.3.2
- *hal antyam* 1.3.3
- *na vibhaktau tusmāḥ* 1.3.4
- *ādir ñiṭuḍavaḥ* 1.3.5
- *ṣaḥ pratyayasya* 1.3.6
- *cuṭū* 1.3.7
- *laśakv ataddhite* 1.3.8

If we take into account the recurrence (*anuvṛtti*) of terms from preceding sūtras, the rules may be rewritten (indicating the anuvṛtti by indentation) as

- *upadeśe*
  - *ac anunāsika (=) it* 1.3.2
  - *hal antyam* 1.3.3
    - *na vibhaktau tusmāḥ (=it)* 1.3.4
  - *ādiḥ*
    - *ñiṭuḍavaḥ (=it)* 1.3.5
    - *pratyayasya*
      - *ṣaḥ (=it)* 1.3.6
      - *cutū (=it)* 1.3.7
      - *laśaku (=it) ataddhite* 1.3.8

Translation of this set of rules into a simple algorithm will show the parallel between Pāṇini's sūtras and a computer algorithm.

```
if(input is from UPADE‘SA)
     Mark the ANUNASIKA AC as INDICATOR
  if(last var.na(X) is HAL)
     if(the input is neither VIBAKTI nor TUSMA)
        Mark X as INDICATOR
     endif
  endif
  if(the beginning syllable(Y) is ˜ni or .tu or .du)
     Mark Y as INDICATOR
  endif
  if(the input is a PRATYAYA)
     if(Y is .sa)
        MARK Y as INDICATOR
     endif
     if(Y is from ca_varga or .ta_varga)
        Mark Y as INDICATOR
     endif
     if(the input is NOT TADDHITA)
        if(Y is either la or ‘sa or ka_varga)
           Mark Y as INDICATOR
        endif
     endif
endif
```

There are many such instances of well-defined subroutines spread all over the *Aṣṭādhyāyī*.

3.4 Operations

The nature of the problem the *Aṣṭādhyāyī* deals with indicates that the typical operations involved are various kinds of string operations. They are of four types.

- assigning a name
- substitution
- insertion
- deletion

It has been already recognized ([**?**], [**?**]) that Pāṇini expresses all such rules as context sensitive rules. He ingeniously uses cases (*vibhaktis*) to specify the context. A typical context sensitive rule is of the form

$$\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$$

Pāṇini uses 5^{th} and 7^{th} case to indicate the left and right context respectively, 6^{th} case to indicate the element that will undergo a change and 1^{st} to indicate what it will change to. Here is an example from the *Aṣṭādhyāyī*.

ato ror aplutād aplute (ut ati saṃhitāyām) 6.1.113

at{*5*} *ru*{*6*} *apluta*{*5*} *apulta*{*7*} *(ut*{*1*} *at*{*7*}*)*
apluta_at **ru** *apluta_at* == > *apluta_at* **ut** *apluta_at*
gloss: change the 'ru' preceeded and followed by an 'a-pluta a' to 'u'.

3.5 Ordering of the rules

Three sūtras in the *Aṣṭādhyāyī* play an important role in deciding the order of the rules. These sūtras are

- *pūrvatrāsiddham* 8.2.1
- *asiddhavad atrābhāt* 6.4.22
- *ṣatvatukor asiddhaḥ* 6.1.86

We discuss each of these sūtras and show how they govern the ordering.

Asiddham Traditionally the *Aṣṭādhyāyī* is divided into 2 parts – *sapāda-saptādhyāyī* and *tripādī*. The rule *pūrvatrāsiddham* (8.2.1) makes the output of the rules in the latter part unavailable to the earlier rules. Further this rule being the adhikāra sūtra makes the output of each of the following sūtras unavailable to the previous rules within the *tripādī*. This necessarily implies that the *tripādī* should follow the *sapāda saptādhyāyī* and that the rules within the *tripādī* should also be followed linearly or sequentially. Based on this, it is very likely that one would be tempted to model *tripādī* as a single subroutine, where all the rules are applied sequentially and the intermediate output is stored in local variables.
But then it would not be a faithful representation. Pāṇini did not state it this way. The sequential ordering of sūtras in the *tripādī* and application of *tripādī sūtra*s after the application of rules from the *sapāda saptādhyāyī* is an inference we draw from the adhikāra sūtra. Instead of inferring, let us try to understand precisely what Pāṇini has said. The word *asiddham* means – (regarded as) not existing ([**?**], p. 120, col. 3). To give an analogy, in programming paradigm, the variables local to a subroutine are regarded as not existing(or not visible) with respect to the calling function. We model the sūtra *pūrvatrāsiddham* as follows: The result of applying the sūtras in this section should not be available to the earlier sūtras in the *sapāda saptādhyāyī*. Similarly, the sūtra being the adhikāra sūtra, the result of the later sūtra in the *tripādī* should also be not available to the earlier sūtras in the *tripādī*. It essentially means that each of the sūtras in the *tripādī* section should have its own data space and the data space of the later sūtras be invisible to the earlier sūtras. Thus this model does not implement *tripādī* as a single subroutine, but keeps each of the rules (or a group of rules forming a subroutine) as a single separate unit. The invisibility of the data spaces of later rules to the earlier rules ensures that the rules are applied only sequentially.

Asiddhavat Within the *sapāda saptādhyāyī* there is a section known as the *asiddhavat* section. The sūtra

asiddhavad atrābhāt 6.4.22

is translated by Vasu([**?**]) as
"The change, which a stem will undergo by the application of any of the rules from this sūtra up to 6.4.129 is to be considered as not to have taken effect, when we have to apply any other rule of this very section 6.4.23-6.4.129".
As an example, let us consider the derivation of *śādhi* from *śās + hi*.
Two sūtras

hujhalbhyo her dhiḥ 6.4.101

and

śā hau 6.4.35

are applicable.
6.4.101: *śās* + **hi** ⇒ *śās* + **dhi**
6.4.35: **śās** + *hi* ⇒ **śā** + *hi*

As is evident from this, if 6.4.101 is applied, then the conditions for applying 6.4.35 are not met and hence it would not be applicable. Similarly, if 6.4.35 is applied first, then the conditions for 6.4.101 would not be met and it would not be applicable. The word *asiddhavat* means 'as if it is not applied'. So after applying 6.4.35, though *śās* changes to *śā*, still the result is not visible to 6.4.101 and hence 6.4.101 changes *hi* to *dhi*. As a result of both these rules, *śās + hi* changes to *śādhi*. Thus instead of stating the rule as

R: a b ⇒ c d

Pāṇini states it as a combination of two rules:

$$R_1 : ab \Rightarrow cb$$

$$R_2 : ab \Rightarrow ad$$

and thereby one may conclude that Pāṇini achieves economy.[5] However, if one looks at the complete asiddhavat section, one finds only handful of examples that require parallel application of rules, and hence it is not worth stating that economy is achieved. Nevertheless, it provides an example of task parallelism. Further this also has an impact on parameter passing. Since the same input should be available to all the rules in this section, the input should be passed as a value. But at the same time, a local copy of it will undergo necessary changes. One or more processes run in parallel, and the consolidated result of all these processes is then passed back to the calling function.

[5] If there were n_1 rules of type R_1 and n_2 rules of type R_2, then there would have been $n_1 * n_2$ (possible combinations) rules of type R. However, by making them applicable in parallel there are only $n_1 + n_2$ rules, and thus economy is achieved.

Though this interpretation also seems to be consistent with what is said in the sūtra, still it is also an inference. Pāṇini never mentions that the sūtras are to be applied in parallel(Bronkhorst[**?**]). He uses the term *asiddhavat*. So to be faithful to the Pāṇini's system then, the results of the application of sūtras in this section should not be visible to other sūtras of the same section. This is possible, if we assign a separate data space to the rules in this section, which is not visible to the rules within this section.

Asiddhaḥ The third type of *asiddha* is provided by the sūtra

ṣatvatukor asiddhaḥ 6.1.86

This rule, which occurs under the adhikāra *ekaḥ pūrvaparayoḥ* 6.1.84 in effect up to 6.1.110, says that the single replacement (*ekādeśa*) that will result through the application of rules under this adhikāra, is *asiddha* with respect to the two processes, viz. *ṣatva* and *tuk*. That is, the result of application of rules in the *ekādeśa* section is invisible to the rules which correspond to the *ṣatva* or *tuk* processes. Thus here again, there is a concept of data space, the result of the operations in the *ekādeśa* section are written to this data space, which are unavailable to the rules performing *ṣatva* and *tuk* operations.

3.6 Programming Model

The typical grammarians view of the *Aṣṭādhyāyī* may be stated as follows:
'The rules in the *sapāda saptādhyāyī* seek for an opportunity to act on an input by finding conditions (*nimitta*) in which they are applicable. In case there is a conflict, there are certain conflict resolution techniques (described as *paribhāṣā*), which come into play. The conflict resolver selects one rule and effects changes in the data space.'
This model is described in the Figure **??**.

Thus we notice a striking similarity between the event driven programming and triggering of *Aṣṭādhyāyī* rules. The *nimitta* or the context triggers an appropriate rule. The saṁjñā sūtras for example, assign different 'attributes' to the input string, thereby creating an environment for certain sūtras to get triggered. The adhikāra sūtras assign necessary conditions to the sūtra for getting triggered. Paribhāṣā sūtras provide a meta language for interpreting other sūtras. The niyama and atideśa sūtras put restrictions on or extend the domains in which the sūtras are to be applied. Finally it is the vidhisūtras which effect the transformations.

The rules fall under four categories: *asiddham*, *asiddhavat*, *asiddha* and the rest[6]. Each of these rules has its own data space where it writes its own output (see Table 1). The visibility of these data spaces to different categories of rules is described in Table 2.

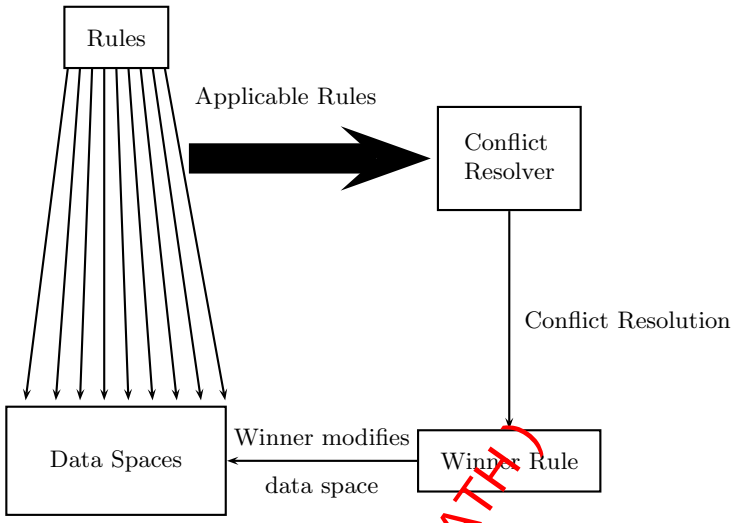

Fig. 2. Application of rules

Now we illustrate, with the help of examples, how the privacy of data spaces lead to the correct generation in case of *asiddhavat*, *asiddham* and *asiddhaḥ* sections.

– *asiddhavat* Consider the derivation of *śādhi*: imperative second person singular form of the root *śās*. The derivation of our interest starts with the data space D_0 having *śās* + *hi* in it. Now two sūtras, viz. *hujalbhyo her dhiḥ*(6.4.101) and *śā hau*(6.4.35) are applicable. The constraint resolver returns both the rules. Thus there are two different orders of applying these rules. Let us assume that 6.4.101 is applied first. Then *hi* changes to *dhi*. This result will be available in the data space D_1 which is not visible to sūtra 6.4.35. As such, 6.4.35 applies on the contents of the data space D_0 and changes *śās* to *śā*. This change also gets stored in D_1. Thus now D_1 contains *śā* + *dhi*. As is clear, even if the order of application of sūtras is changed, we get the same result as in D_1.
Consider another derivation: *jahi*, from the root *han*. The step in the derivation in which we are interested is when the data space D_0 has *han* + *hi* in

[6] This model is an improvement over the one emerged while teaching a course on 'Structure of Ashtadhyayi' at Tirupati, in 2006-07 (described in 'Conflict Resolution Techniques in Astaadhyaayii' by Varkhedi and Sridhar (To appear)): the major shift is in modelling with private data spaces. The interactions of the rules belonging to *ṣatva* and *tuk* sections with the other rules is also made explicit. The current model based on privacy of data spaces has its seed in the discussions Amba Kulkarni had with Vineet Chaitanya.

Table 1. Affected Data Spaces

Rule	Affected Data Space
siddha	D_0
asiddhavat	D_1
asiddhaḥ (rules in ekādeśa section)	D_2
asiddham (*Tripādī - ṣatva*)	$D_{8.2.1}$ $D_{8.2.2}$ $D_{8.2.3}$ . . $D_{8.4.68}$

Table 2. Visible Data Spaces

Rule	Visible Data Spaces
Rest	D_0, D_1, D_2
asiddhavat	D_0, D_2
ṣatva and tuk	D_0, D_1
asiddham(*Tripādī - ṣatva*)	D_0, D_1, D_2, $D_{8.2.1}$
8.2.2	D_0, D_1, D_2, $D_{8.2.1}$, $D_{8.2.2}$
8.2.3	D_0, D_1, D_2, $D_{8.2.1}$, $D_{8.2.2}$, $D_{8.2.3}$
.	.
.	.
8.4.68	D_0, D_1, D_2, $D_{8.2.1}$,..., $D_{8.4.68}$

it. *hanter jaḥ* (6.4.36) changes the input to *ja* + *hi*. Since this sūtra is from the *asiddhavat* section, the output is written to data space D_1, and hence is not visible to any other rule in the *asiddhavat* section; in particular to *ato heḥ* (6.4.105). Therefore there is no question of *ato heḥ* getting triggered.

– *asiddhaḥ* Let the data space D_0 has *adhi* + *i* + *lyap*. Now two sūtras, viz. *akaḥ savarṇe dīrghaḥ* (6.1.101) and *hrasvasya piti kṛti tuk* (6.1.71) are applicable. With the data space model, we explain how the order of the rules gets fixed automatically. Let us assume that rule 6.1.101 is applied first. Since this rule belongs to the *ekādeśa* section, the result, viz. *adhī* + *lyap* will be stored in the data space D_2, and hence will not be visible to 6.1.71, as such the latter rule operates on the data in D_0, and changes it to *adhi* + *i* + *tuk* + *lyap*. Again at this stage, 6.1.101 can see the contents of D_0, and hence changes it to *adhī* + *tuk* + *lyap*. Had we applied 6.1.71 first and then 6.1.101, it would have resulted in the same string. Or in other words, the order 6.1.71 and then 6.1.101 is optimal, than 6.1.101, followed by 6.1.71 followed by 6.1.101 again. Application of 6.1.101 before 6.1.71 being redundant, implementation of preferred order is very straight forward.
Consider another example. The input of our interest is *ko* + *asicat* stored in the data space D_0. *eṅaḥ padāntād ati* (6.1.109) changes it to *kosicat*. This

result is stored in the data space D_2. As such this result is not visible to the *ādeśapratyayoḥ* (8.3.59) from the *ṣatva* section and thus 8.3.59 is not applicable. Thus the undesirable result *koṣicat* is not generated.

– *asiddham* Finally take the example of *vac* + *ti*. Two sūtras, viz. *coḥ kuḥ* (8.2.30) and *stoḥ ścunā ścuḥ* (8.4.40) are applicable. Since the result of 8.4.40 (stored in $D_{8.4.40}$) is not visible to 8.2.30, application of 8.4.40 before 8.2.30 will be redundant. After application of 8.2.30, there will not be any scope for 8.4.40, giving the desired result *vakti*.

Thus, these examples illustrate how the concept of data spaces represent the simulation of *Aṣṭādhyāyī* faithfully.

4 Implementation

Simulation of the *Aṣṭādhyāyī* involves the following factors:

1. Interpretation of sūtras using the metalanguage described by Pāṇini in the *Aṣṭādhyāyī*,
2. Faithful representation of sūtras,
3. Automatic triggering of rules, and
4. Automatic conflict resolution.

In this paper we have concentrated only on 3. For conflict resolution we have used two main principles: An *apavāda* rule and a rule belonging to the *aṅga* section are given priority over the others. Regarding the representation of sūtras, we use regular expressions to represent the patterns, and positions to represent the left and right context, the string that undergoes a change and the resultant string.

The model of the *Aṣṭādhyāyī* that we are implementing is presented in the Figure 3.

As the diagram shows, we have classified the rules in the following way:

1. E - Rules belonging to *ekādeśa*
2. A - Rules belonging to *asiddhavat*
3. R - Rules belonging to *ṣatva vidhi*
4. W - Rules belonging to *tuk vidhi*
5. T - Rules belonging to *tripādī*, excluding *ṣatva vidhi*.
6. O - All other rules

The diagram shows the data flow from one data space to another by the invocation of different types of rules. A rule represented by an arrow takes input from the data space at the tail of the arrow, and writes the output to a data space

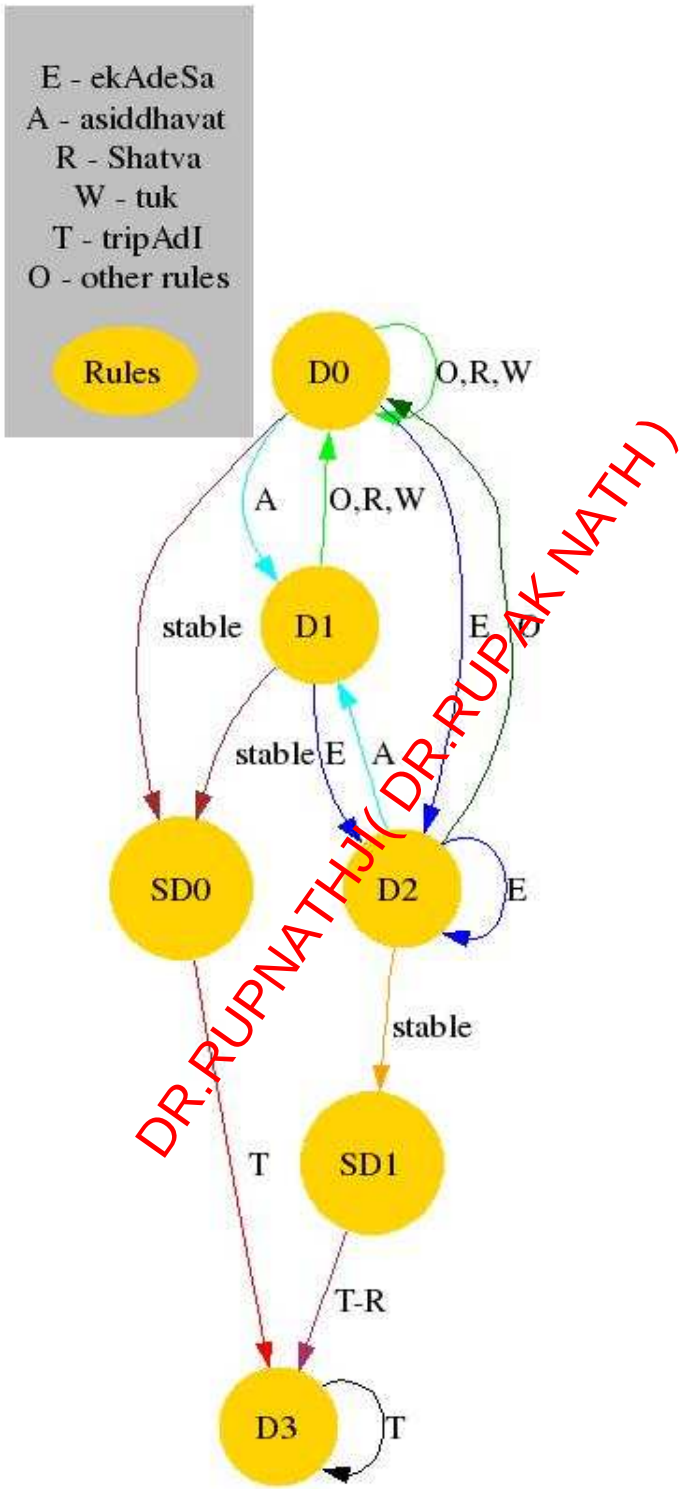

Fig. 3. Modelling Aṣṭādhyāyī

indicated by the head of the arrow. In the beginning, the input string is stored in the dataspace D_0. A rule from O section can see the contents of only three data spaces viz. D_0, D_1 and D_2. At any stage, when a rule from O section is applicable, among these three data spaces, the data space with latest information is chosen as an input to the rule. If D_0 is the input data space, the output is written to itself. But if D_1 has the latest data, then the output is written to D_0. In case D_2 has the latest data, then the output is written to D_0. We illustrate this with an example from asiddhavat section.

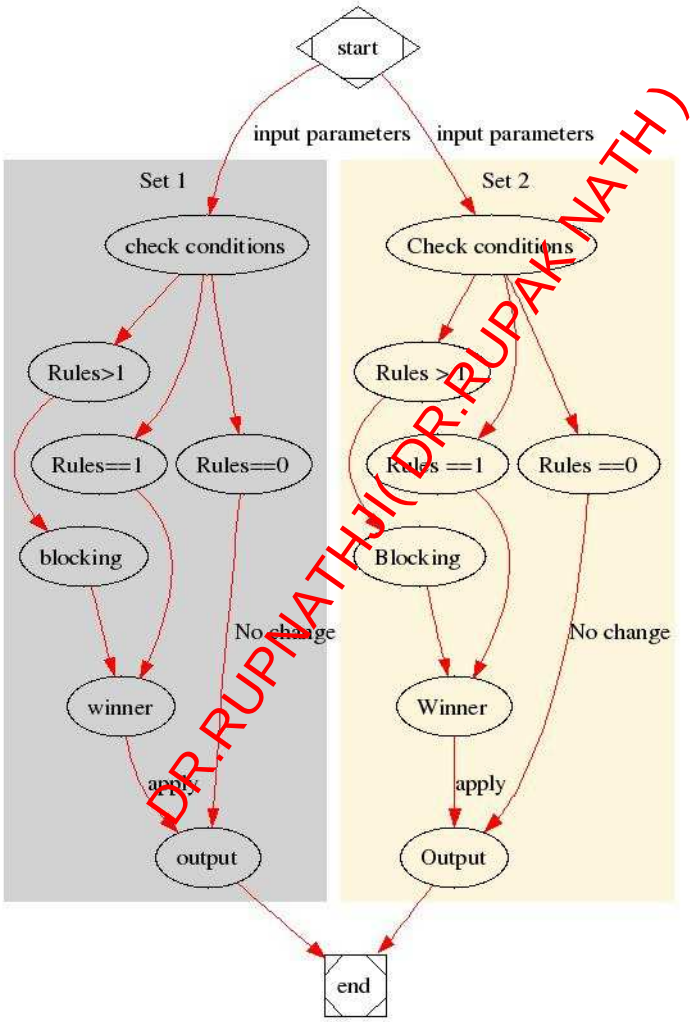

Fig. 4. parallel execution in asiddhavat

Let D_0 contain *han + hi*, and suppose it has the latest data among all the data spaces. Now the rule *hanter jaḥ* 6.4.36, which is from the asiddhavat section (A) is applicable. This rule then takes the latest data (in this case from D_0), and writes the output to D_1. So D_1 contains *ja + hi*, whereas D_0 still continues to have *han + hi* in it. The rule *ato heḥ* 6.4.105 can see only D_0 and D_2, and can't see D_1. Hence, 6.4.105 will not be applicable, thereby the wrong generation is stopped. Similarly when the rules from the *ekādeśa* section are triggered, they write the output to the data space D_2. When no more rules are triggered, then the system enters into a stable state SD_0. This stable state is different (SD_1) if the system is in the state corresponding to *ekādeśa*. While in one of the stable states, the rules in the *tripādī* section get triggered sequentially. If the system is in the *ekādeśa* state, and no rule from *tripādī* gets triggered, then the rules from *ṣatva* section also do not get triggered. But if the rules in *tripādī* have been applied, then *ṣatva* rules may get triggered.
If more than one rule from the *asiddhavat* section is triggered, then their behaviour is shown in the Figure **??**.
We have started the implementation for getting *śabdarūpa* of nominals given the *prātipadika*, along with the *vacana*, *liṅga*, and *vibhakti* for which we need to decline.

4.1 Hierarchy

We take the input as a sentence. Thus given an input such as *rāma(puṁliṅgam +ekavacanam+kartṛ) vana(napuṁsakaliṅgam+ekavacanam+karma) gam(laṭ+kartari)*, machine should produce *rāmaḥ vanam gacchati*, along with the trace of an algorithm showing the exact sequence of the applied rules, and conflict resolution, if any. A sentence has padas as its children. Each pada will have a root word along with the attributes. We will first run the program on the leaf nodes and will keep on merging these together until we reach the sentence. Thus for the example taken, we have the following children in the beginning:

- *rāma(puṁliṅa+ekavacana+kartā)*
- *vana(napuṁsakaliṅa+ekavacana+karma)*
- *gam(laṭ+kartari)*

Each child has a root word along with the attributes. Henceforth we call this structure 'input DS'. We pass the children one by one through the program. Finally, we have different padas, and that will be the output (*rāmaḥ vanam gacchati*).

4.2 Structure of input DS

An input DS contains an array of words along with their attributes. It allows the addition of a particular attribute as well as removal if necessary. We have allowed substitution as well as augmentation in the structure. Let us briefly discuss these two operations:

Augmentation The augmentation process needs two parameters: a string to be augmented and the position where it is to be augmented. The function will look at the rules applicable (such as *ādyantau ṭakitau* 1.1.46 and *midaco'ntyāt paraḥ* 1.1.47), and the change is effected at appropriate position in the input DS.

Substitution Substitution is also permitted within the input DS. The parameters specified are the same as in augmentation. The rules such as *ṅit ca* 1.1.53, *anekālśit sarvasya* 1.1.55, *ādeḥ parasya* 1.1.54 and *alo'ntyasya* 1.1.52 are the governing rules out of which one is applied depending upon the input DS and after the substitution the DS is returned.

4.3 Structure of a rule

A typical vidhi rule of the *Aṣṭādhyāyī* has a context sensitive form:

$$\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$$

α and γ specify the context, β is the domain which undergoes change and δ is the resultant/changed part of the string.
Corresponding to each rule now we require three functions:

1. a function that checks whether the rule is applicable or not,
2. a function to compute the result if the function is applicable, and finally
3. a function that returns the conditions under which the rule is applicable.

Thus:

1. Each rule has been stored as a structure involving patterns that stores the conditions under which the rule is applicable.
2. Then we apply pattern matching to come up with the list of rules triggered for a given input DS. The application part has been stored separately, which takes input as the rule number of the winner rule and changes the input DS, accordingly.
3. In case there is more than one rule applicable, then the conflict resolution module looks at the conditions for each of the applicable rules and chooses the "winner rule", after resolving the conflict if any. Conflict resolution is discussed in the next section.

4.4 Modules

Each *prakṛti* as well as the *pratyaya* part of the *pada* passes through a series of modules. The modules are of 2 types. Some modules just look at the input and assign different names (*saṃjñās*) to the different parts of the input string; others transform the input string.

Module for assigning saṃjñās to the prakṛti A prakṛti is assigned an appropriate saṃjñā using the

- databases, or
- current state of the input string.

For example, a prakṛti gets the saṃjñā *dhātu* if it is found in the *dhātupāṭha*, the saṃjñā *sarvanāma* if it is found in the gaṇa *sarvādi*, etc. Sometimes some pratyaya creates a context for certain saṃjñās such as *bha, ghi, nadī*, etc.

***it* module:** The dhātus in the *dhātupātha*, pratyayas, etc. are given with some markers commonly called *anubandha*s and termed *it* by Pāṅini. These markers need to be identified and marked, before the processing starts. The module takes as input a particular *upadeśa* and marks the varṇa as *it*. The rules 1.3.2 to 1.3.8 described in section 3.3 mark the *it*s. Finally, the phonemes marked *it* are deleted from the input string by the rule *tasya lopaḥ* 1.3.9. However the markers are stored in the DS, as they actively bind the procedures.

Module for *pratyaya vidhi* In this module, a pratyaya undergoes some changes based on the characteristics of the *aṅga*[7] and pratyaya. All the rules of this particular vidhi are listed as an array 'pratyaya vidhi rule' in a particular data structure as follows:

1. *aṅga ending*: Denotes the context of the endings of an *aṅga*.
2. *aṅga attrib*: Represents the attributes (characteristics) of the *aṅga*.
3. *pratyaya*: If there are some special contexts for the pratyaya.
4. *pratyaya attrib*: Represents the attributes of the pratyaya.
5. *rule number*: Keeps the information of the rule number.

The data structure has been made in Java, and constructors are made to directly encode the conditions of a particular *pratyaya vidhi* rule. For example, the rule *ato bhisa ais* has been encoded as:

```
list_pratyaya_vidhi_rule[0]=new pratyaya_vidhi_rule(‘‘7-1-9’’,
‘‘ataH bhisaH ais’’, ‘‘a’’, ‘‘’’, ‘‘’’, ‘‘root(bhis)’’);
```

The '[0]' indicates this is the first rule of the array. The rule constructs a 'pratyaya vidhi rule' which needs the ending of the aṅga to be *a*, and the pratyaya to be the one with its original form as *bhis*. The rule number too has been stored. All the rules of *pratyaya vidhi* are stored in a similar fashion. When a particular 'input DS' is passed through this array, another array 'triggered pratyaya vidhi rule' enlisting all the rules which are applicable is produced. Finally, the conflict resolver decides the 'winner rule".
After the winner rule has been selected, the subroutine 'apply' is called. For example, the above rule substitutes *ais* for the case ending *bhis* after a prātipadika

[7] *In accordance with yasmāt pratyayavidhis tadādi pratyaye 'ṅgam* 1.4.13, the *prakṛti* to which an affix has been provided, is termed *aṅga*.

ending in *a*. The input DS is passed through the substitution subroutine with the information that *ais* is to be substituted. Thus, for the input DS with (*rāma*(attributes)+*bhis*(attributes)), the rule 7.1.9 will apply as 'substitute(input DS, *ais*, 1)', where we pass the information that substitution has to be done at index 1. The result of this substitution will be: (*rāma*(attributes) +*ais*(attributes)).

Module for *aṅga vidhi* In this module, an aṅga undergoes some changes based on the characteristics of the *aṅga* and pratyaya. The rules are listed in the same data structure as used for rules belonging to *pratyaya vidhi*. The rules are stored in an array 'list anga vidhi rule'. Consider the encoding of the rule *aco ñṇiti* which states, "Before the affixes having an indicatory *ñ* or *ṇ*, *vṛddhi* is substituted for the end vowel of a stem".

`list_anga_vidhi_rule[9]=new anga_vidhi_rule(''7-2-115'',`"*acaḥ ñṇiti*",ac,"","",`''N-it|~N-it'');`

This is the tenth rule of the array corresponding to *aṅga vidhi* rules. From the encoding, we can infer the conditions that the aṅga should be ending in a vowel (*ac*) and the pratyaya should have either *ṇ* or *ñ* as a marker.
After passing the input DS through the array, we will get the array 'triggered anga vidhi rule' upon which the conflict resolution module will be called giving the winner rule. Finally the subroutine 'apply' will act. Consider the formation of the nominative singular (*prathamā ekavacana*) of the root *go*. Since the pratyaya is termed *sarvanāmasthāna*, it gets the characteristics of indicatory *ṇ* by the rule *goto ṇit* 7.1.90, a rule in *pratyaya vidhi*. In *aṅga vidhi*, the winner rule will be 7.2.115 and it will do *vṛddhi* on the end vowel of *go* giving the structure as (*gau*(attributes)+*as*(attributes)).

Module for *asiddhavat* In the *asiddhavat* section, we have certain rules belonging to both *pratyaya vidhi* and *aṅga vidhi*. The speciality of this section is that a rule in this section does not see the changes made by other rules in the same section. To implement this, the rules have been grouped in two separate arrays, belonging to *pratyaya vidhi* and *aṅga vidhi*. The same input string is passed to these arrays. The *pratyaya vidhi* rules may change the pratyaya, and the *aṅga vidhi* rules may change the aṅga. The aṅga from the *aṅga vidhi* module and the pratyaya from the *pratyaya vidhi* module are taken together to the input DS which is available to other rules. Let us consider the formation of *śādhi*. The input DS that is passed to this section is (*śās*(attributes) + *dhi*(attributes)). We are passing the same copy to both the *aṅga vidhi* and the *pratyaya vidhi* modules of the *asiddhavat* section. In each of the two *vidhi* modules, if more than one rule is triggered, the conflict resolution module selects one winner for each type of the rules separately. The module is described in the Figure **??**.

Module for sandhi We have enlisted all the rules belonging to sandhi in an array 'list Sandhi rule'. The encoding format is the same as the one used for the

pratyaya vidhi and *aṅga vidhi* modules. For example, the conditions for the rule *eco'yavāyāvaḥ* 6.1.83 have been stored as:

```
list_Sandhi_rule[1]=new Sandhi_rule
(‘‘6-1-78’’,‘‘(eco)83 ayavAyAvaH’’,ec,ac);
```

In this case, we have constructors which build the object 'Sandhi rule' with the information of the last letter of the first word and initial letter of the second word. The rule states, "The vowels belonging to the pratyāhāra *ec*, are replaced by *ay, āy, av, āv* respectively provided the second word starts with a vowel." The processing is the same as discussed in the *pratyaya vidhi* section.

Module for *tripādī* In the rules belonging to this section, we need not have a conflict resolution module since rules are applied in linear order. So, we need only to check the conditions and apply the rule if the condition is satisfied. This module is visited after we are sure that the output has become stable after going through the other modules.

Module for Conflict Resolution: The module is made independent of the category of the rules. It takes as input two rules at a time and returns the rule that blocks the other (it returns the superior rule). Let the two rules be Rule A and B. If the domain of rule A is properly included in the domain of rule B, then A blocks rule B. While applying blocking, we look at the following properties in the decreasing order of their priority:

1. Whether there is conflict (*pratiṣedha*) in the rules: If the two rules present no pratiṣedha, i.e. application of rule A doesn't bleed (depriving of conditions) rule B, and A and B are in the order of the *Aṣṭādhyāyī*, we apply the rules in order.
2. Word integrity principle: If a rule from *ekādeśa* interacts with a rule from the *aṅga vidhi* module, the rule from the *aṅga vidhi* module is the winner using this principle.
3. The environment-changing rule is given precedence over the rule that is not environment-changing. Thus if rule A bleeds rule B, but rule B doesn't bleed rule A, rule A is the winner due to changing the environment.
4. Whether a rule is specific to a particular initially taught item: If a rule specifies a particular initially taught item, it is preferred over other rules. For example:
   We consider the formation of the accusative singular (*dvitīyā ekavacana*) of the base *rāma*. When it passes through the sandhi module, it has the structure (*rāma*(attributes) + *am*(attributes)). All the rules in the sandhi module check for the applicability conditions and the 'triggered Sandhi rule' array contains the following rules:
   (a) *ād guṇaḥ* 6.1.87
   (b) *akaḥ savarṇe dīrghaḥ* 6.1.101
   (c) *prathamayoḥ pūrvasavarṇaḥ* 6.1.102

(d) *ato guṇe* 6.1.97
(e) *ami pūrvaḥ* 6.1.107

From the above stated rules, the rule *ami pūrvaḥ* is applicable only when the second item has *a* belonging to the initially introduced item *am*. This initially introduced item restricts the domain for this rule and it is preferred over other rules.

5. Whether a rule is specific to a particular feature of a domain: Consider the formation of the dative plural (*caturthī bahuvacana*) of the base *rāma*. When it passes through the *aṅga vidhi* module, it has the structure (*rāma*(attributes) + *bhyas*(attributes 'bahuvacana')). After the condition checking the 'triggered anga vidhi rule' array contains the following rules:
   (a) *supi ca* 7.3.102
   (b) *bahuvacane jhaly et* 7.3.103

   Out of the above two rules, rule 7.3.103 requires a special property of *bahuvacana*, which makes it preferable over the rule 7.3.102.
6. Whether a rule is applicable for fewer sounds (*ak*): A rule involving fewer phonemes in the context is given more priority over the one involving more.

5 Challenges

Whether to include vārtikās?
In the traditional view, there are vārtikās that handle those cases where the rule *vipratiṣedhe param kāryam* or the paribhāṣā *paranityāntaraṅgāpavādānām uttarottaram balīyaḥ* doesn't give the right result. The question arises whether it is necessary to go for these vārtikās or we can resolve the conflicts without resorting to the vārtikas. We enlist one of the cases below:
Formation of *vāri* + *ṅe*. We are at the structure *vāri(napuṃsaka)+e(ṅ-it, sup)*. The rules that are triggered are:

- *iko'ci vibhaktau* 7.1.73
- *gher ṅiti* 7.3.111

Rule 7.3.111 is the later rule and should be applied according to the principle that a later rule takes precedence over an earlier one. However, it does not apply; 7.1.73 applies instead. For this there is a vārtikā *vṛddhyautvatṛjvadbhāvaguṇebhyo num pūrvapratiṣedhena* which says that the later rule is not applicable when *num* is ordained by a previous rule and one of *vṛddhi, autva, tṛjvadbhāva* or *guṇa* is ordained by a later rule. Till the implementation of the algorithm presented here, we do not have a satisfactory answer for this.

6 Exceptions

There are certain exceptions in the *Aṣṭādhyāyī* which need to be handled separately. Consider the formation of *śivorcya*.

– We have the form *śivas + arcya*. No rules from the *sapāda saptādhyāyī* are applicable.
– The rule *sasajuṣo ruḥ* 8.2.66 finds scope and is applied. Thus we have the form:

$$śiva + ru + arcya$$

– After the *u* in *ru* will be marked as a marker(*anubandha*), leading to its deletion (*lopa*).

$$śivar + arcya$$

– The rule *bhobhagoaghoapūrvasya yo'śi* 8.3.17 finds scope and will change the structure to

$$śivay + arcya$$

which is an *aniṣṭa* form. The *apavāda* of this rule, *ato ror aplutād aplute (ut ati saṃhitāyām)* 6.1.113 gets *ru* in *śivar* by the *sthānivadbhāva* and checks the application of 8.3.17. 6.1.113 changes the *r* to *u* giving the structure:

$$śiva + u + arcya$$

– The rule *ād guṇaḥ* 6.1.87 is applicable giving the form:

$$śivo + arcya$$

– The rule *eṅaḥ padāntād ati* 6.1.109 is applicable giving the form:

$$śivorcya$$

Thus we clearly see that this is an exception for the adhikāra sūtra *pūrvatra asiddham*.

7 Problems

Consider the formation of *ramā + ṅe*. The structure is:

$$ramā + e$$

The rule *yādāpaḥ* - 7.3.113 which sees the *ṅit* of the *pratyaya* and does the *āgama* of *yāṭ*, which being *ṭ-it*, sits in the front, and we have

$$ramā + yai$$

The problem comes that the *ṅit* attribute is still there in *yai* and the rule gets triggered again and again giving the form *ramā+yāyā...yai*. We need to seek solution for this.

8 Future Work

It is necessary to understand how Pāṇini's *Aṣṭādhyāyī* resolves the conflicts. The current implementation is still primitive and not satisfactory. Pāṇini has not mentioned any conflict resolution rules explicitly, but it seems he assumed them implicitly. In the current implementation, the rules are represented using manually coded patterns. It will be interesting to see if the machine can interpret the rules automatically based on the vibhaktis and the meta rules. What difference the *yoga vibhāga* makes in the form of output of conflict resolution will an interesting issue to explore.

References

[1] Bharati Akshar, Vineet Chaitanya, Rajeev Sangal: Natural Language Processing: A Paninian Perspective, Prentice Hall of India, New Delhi, 1995.

[2] Bhate, S. and Kak, S.: Panini's grammar and Computer Science. Annals of the Bhandarkar Oriental Research Institute vol. 72, 1993, 79–94.

[3] Bronkhorst, J.: Asiddha in the Aṣṭādhyāyī : A misunderstanding among the traditional commentators, Journal of Indian Philosophy, 8, 1980, 69–85.

[4] Cardona, G.: On translating and formalizing Paninian rules, Journal of Oriental Institute, Baroda, vol. 14: 306-14.

[5] Hyman, M. D.: From Paninian Sandhi to Finite State Calculus, Proceedings of the First International Sanskrit Computational Linguistics Symposium, ed. Gérard Huet and Amba Kulkarni, 2007 .

[6] Kiparsky, P.: On the architecture of Pāṇini grammar, the lectures delivered at the Hyderabad Conference on the Architecture of Grammar, 2002.

[7] Mishra, A.: Simulating the Paninian system of Sanskrit Grammar, Proceedings of the First Sanskrit Computational Linguistics Symposium, ed. Gérard Huet and Amba Kulkarni, 2007.

[8] Monier Williams, M.: A Sanskrit-English Dictionary, Clarendon, Oxford, 1872 [Reprint: Motilal Banarasidass, Delhi, 1997].

[9] Roy, P. V. and Haridi, S.: Concepts, Techniques and Models of Computer Programming, MIT Press, Cambridge, 2004.

[10] Scharf, P. M.: "Modeling Paninian Grammar", Proceedings of the First International Sanskrit Computational Linguistics Symposium, ed. Gérard Huet and Amba Kulkarni, 77-94, 2007 .

[11] Subrahmanyam, P. S.: Pāṇinian Linguistics, Institute for the Study of Languages and Cultures of Asia and Africa, Tokyo University of Foreign Studies, Japan, 1999.

[12] Vasu, S. C.: Siddhānta Kaumudi, Motilal Banarasidas Publishers, New Delhi, 2002.

[13] Vasu, S. C.: The Aṣṭādhyāyī of Pāṇini, Motilal Banarasidas Publishers, New Delhi, 2003.

Appendix

We illustrate the formation of *rāmāṇām*, the plural, masculine form in genitive case of the root word *rāma*.

1. The input to the program is: Form: *rāma: bahuvacana, ṣaṣṭhī, puṁliṅga.*
2. *arthavad adhātur apratyayaḥ prātipadikam* 1.2.45
   rAma gets the saṃjñā *prātipadika* after being checked in the database, and we have the form: *rāma (**prātipadika**, bahuvacana, ṣaṣṭhī, puṁliṅga, akārānta,* root(*rāma*)).
3. *su au jas am aut śas ṭā bhyām bhis ṅe bhyām bhyas ṅasi bhyām bhyas ṅas os ām ṅyos sup* 4.1.2
   pratyayaḥ 3.1.1
   paraśca 3.1.2
   We can see the importance of obtaining the attribute *prātipadika* to the nominal stem *rāma*. This encourages us to make a data structure that keeps on adding the attributes to a word for further usage in the rules of the *Aṣṭādhyāyī*.
   The application of this rule needs us to be familiar with the devices of *anuvṛtti* and *adhikāra* adopted by Pāṇini. The device of anuvṛtti aims at avoiding repetition of the same item. The device of adhikāra is used to indicate homogeneity of topic. The adhikāras stand for a subjectwise division of contents of the *Aṣṭādhyāyī*. The adhikāra *pratyayaḥ* 3.1.1 governs the rules in adhyāyas 3-5 and tells us that the items prescribed by these rules are called pratyaya. Further, the rule *paraśca* 3.1.2 -'That which is called a *pratyaya* is placed after the crude form', has its anuvṛtti till the end of chapter five. Thus, both these rules are applicable in the current rule and thus these affixes get the attribute of *pratyaya* and are applied after *rāma*.
   Form: *(rāma(prātipadika, bahuvacana, ṣaṣṭhī, puṁliṅga, akārānta,* root(*rāma*)) + **sup(pratyaya)**.
4. *tāni ekavacanadvivacanabahuvacanāny ekaśaḥ* 1.4.102
   supaḥ 1.4.103
   The array of 21 affixes will be transformed to a 7x3 array with the columns getting the attributes *ekavacana*, *dvivacana*, and *bahuvacana*. Each triad is called *vibhakti*. By matching *vibhakti* and *vacana*, we get the form:
   rāma(prātipadika, bahuvacana, ṣaṣṭhī, puṁliṅga, akārānta, root(*rāma*)) + **ām**(*sup,* **upadeśa, pratyaya, vibhakti, bahuvacana, ṣaṣṭhī,** root(*ām*)).
5. *yasmāt pratyayavidhis tadādi pratyaye'ṅgam* 1.4.13
   suptiṅantam padam 1.4.14
   After the application of these two rules, the structure is:
   (*rāma(prātipadika, bahuvacana, ṣaṣṭhī, puṁliṅga, akārānta,* **aṅga**, root(*rāma*)) + *ām(sup, upadeśa, pratyaya, vibhakti, bahuvacana, ṣaṣṭhī,* root(*ām*))) (*pada*).
6. *yaci bham*: *bha saṃjñā* is given to *rāma*. Form: (*rāma(prātipadika, bahuvacana, ṣaṣṭhī, puṁliṅgaakārānta, aṅga,* **bha**, root(*rāma*)) + *ām(sup, upadeśa, pratyaya, vibhakti, bahuvacana, ṣaṣṭhī,* root(*ām*))) (*pada*).
7. The above data structure is a nimitta of the following sūtras:
   ād guṇaḥ 6.1.87
   akaḥ savarṇe dīrghaḥ 6.1.101
   hrasvanadyāpaḥ nuṭ 7.1.54
   We run the conflict resolution module and the rule 7.1.54 is the winner rule. We have the insertion of *nuṭ* to the pratyaya *ām*. Thus, by the rule *ādyantau*

ṭakitau, we have the following form (after passing through the *it* module:
(*rāma(prātipadika, bahuvacana, ṣaṣṭhī, puṁliṅga, akārānta, aṅga, bha*, root(*rāma*))
+ *nām(sup, upadeśa, pratyaya, vibhakti, bahuvacana, ṣaṣṭhī*, **āgama(nuṭ), ṭ-it, u-it**, root(*nām*))) (*pada*)

8. The above data structure is a nimitta of the following sūtras:
nāmi 6.4.3
supi ca 7.3.102
After running the conflict resolution module, we get 6.4.3 as the winner rule.
Thus, we have the form (after lengthening of the final *a* of *rāma*):
(**rāmā***(prātipadika, bahuvacana, ṣaṣṭhī, puṁliṅga, akārānta, aṅga, bha* root(*rāma*))
+ *nām(sup, upadeśa, pratyaya, vibhakti, bahuvacana, ṣaṣṭhī, āgama(nuṭ), ṭ-it, u-it* root(*nām*))) (*pada*)
9. No other rule from the *sapāda saptādhyāyī* is applicable and the structure moves to the *tripādī* after getting the attribute of *avasāna*.
virāmaḥ avasānam 1.4.110
(*rāmā(prātipadika, bahuvacana, ṣaṣṭhī, puṁliṅga, akārānta, aṅga, bha* root(*rāma*))
+ *nām(sup, upadeśa, pratyaya, vibhakti, bahuvacana, ṣaṣṭhī, āgama(nuṭ), ṭ-it, u-it* root(*nām*))) (*pada*, **avasāna**)
10. *aṭkupvāṅnumvyavāye'pi* 8.4.2
The rule changes the *n* of *rāma* + *nām* to ṇ and the structure is:
rāmāṇām (pada, avasāna) *(rāmā(prātipadika, bahuvacana, ṣaṣṭhī, puṁliṅga, akārānta, aṅga, bha* root(*rāma*)) + *nām(sup, upadeśa, pratyaya, vibhakti, bahuvacana, ṣaṣṭhī, āgama(nuṭ), ṭ-it, u-it* root(*nām*)))