Constraint Logic Programming for Resolution of Relative Time Expressions

Henning Christiansen

Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, Denmark
E-mail: henning@ruc.dk

Abstract. Translating time expression into absolute time points or durations is a challenge for natural languages processing such as text mining and text understanding in general. We present a constraint logic language $CLP(Time)$ tailored to text usages concerned with time and calendar. It provides a simple and flexible formalism to express relationships between different time expressions in a text, thereby giving a recipe for resolving them into absolute time. A constraint solver is developed which, as opposed to some earlier approaches, is independent of the order in which temporal information is introduced, and it can give meaningful output also when no exact reference time is available.

1 Introduction

Humans often prefer relative time expressions in text instead of explicitly time stamping every event. Wordings like "two days later" are preferred when the reference date is known, and it often gives good sense to the human reader even without a reference date. For automated document analysis, correct identification of the actual time or date may be important for text understanding and data mining, where the goal may be to provide a time stamped list of events.

A document may also contain expressions that are relative to an implicit time of writing. Here it may be interesting also to identify the time of writing. References to known events may give a clue as in "Two years ago, when the last Venus transit for the next 100 years took place, ...", where a knowledge base of astronomical events can help concluding that the text was written in 2014.

There are several problems involved in annotating a text with correct time stamps. First of all, the time expressions, both relative and absolute, must be identified; this a task for taggers and parsers for natural language. Secondly, the relationship between the different time expressions must be determined ("two years later than *what?*"), and thirdly, a bit of reasoning is needed to calculate the correct time stamps from these relationships.

For the third task, constraint programming presents several advantages compared to ad-hoc techniques. Constraints have been suggested for this earlier, but has not gained popularity in any major text processing systems. We demonstrate here how constraint *logic* programming (CLP) may give rise to an effective

mechanism for the third step, and also provide a flexible language in which to express relationships between time expressions, thus also overlapping sub-task two. CLP introduces a well-defined semantics, meaning that the calculated times are correct solutions to the network of relationships set up by steps one and two. It implies a robust and incremental evaluation scheme that is independent of the order in which time expressions occur in the text: it can still manage if the only absolute time is given at the very end of a text, or even if no absolute anchor point in time is available. It also integrates in a natural way with knowledge bases of known events which may help to situate a document in time.

We present a constraint logic language *CLP*(*Time*) equipped with a constraint solver tailored to the pragmatics of standard usages related to the Gregorian calendar. It is implemented in the programming language of Constraint Handling Rules [1], which provides a modular, rules-based and easily extendible architecture. While time and duration in principle can be represented by integer and interval arithmetic, we develop specific datatypes relating to calendric notions, so we may, e.g., add two months to a given month without knowing the duration of those months or the year. It is language independent, but may be extended with new sorts of constraints that reflect special usages. *CLP*(*Time*) is currently being developed to support arbitrary intervals, such as "in the late nineties" or "between May and August", which is not described here.

Related work and state of the art are reviewed in section 2 as a background for the present approach. The facilities of the constraint language are introduced in section 3, and it is briefly shown how it can be tested together with Prolog's Definite Clause Grammars and with pre-tagged text. Section 4 demonstrates how it applies for different sorts of text. The implementation of a constraint solver with Constraint Handling Rules is sketched in section 5. Some conclusions and ideas for future work are given in the final section.

2 Related work

To resolve time expressions in a text, one needs to 1) identify those expressions, 2) assign a formula to each such expression that sets the relationship to the context of other time expressions in the text, and 3) to evaluate these formulas.[1]

The HeidelTime system [2,3], which is considered state of the art, is based on a tagger which, via a specific rule format, can be adapted to different languages and specific usages. Such rules are manually crafted, and they combine matching of textual patterns with the building of a limited sort of arithmetic expressions; this may involve relative distance to an assumed anchor time not specified explicitly in the rule. Evaluation has two different modes. In "narratives' mode", all expressions are evaluated sequentially from the start of the document and

[1] In the literature, e.g., [3], the term "normalization" has been used for the third phase. Constraints provide more flexibility in phase 2 for specifying relationships. Phase 3 is a matter of a correct implementation of a constraint solver. Thus constraints tend to make "normalization" a combination of phases 2 and 3.

in "news' mode" every expression is evaluated relative to a fixed document creation time. Evaluation is problematic for narratives until a first absolute time is met, and for news articles when document creation time is missing, but good recall and precision figures are reported for selected classes of documents [3]. Handling of inconsistency (over-specification) is not described. Machine learning have been used for identifying the time of writing for news articles from mentions of historical facts, e.g., [4,5,6]; see also [7] for an overview and more references.

HeidelTime and other works referenced above do not use constraint technology for specification and evaluation of interdependencies, which could lead to simpler and more transparent formulations; in fact constraint techniques are not mentioned at all. Constraints, and especially constraint logic programming, provide a uniform framework for expressing different dependencies and (under-) specifications, which otherwise may give rise to a complex nomenclature (as demonstrated by, e.g., [8,9]). Standard machine learning techniques have been used to train recognizers of time expressions, but it helps only little for learning how to evaluate them; we shall refrain from giving a literature overview of these directions as the goals are different from ours.

Logically based formalisms for reasoning about time exist such as temporal logics, the event calculus [10] or Allen's theory [11], but they do not relate to calendar conventions and everyday usages concerned with time and date. Constraint solvers related to time, dates and calendric data have been seen suggested, e.g., [12,13]; these approaches involve very complex solving algorithms and do not approach the problem of finding partial solutions in case of inconsistency.

No work has been found on constraint *logic* programming related to temporal information in language usage. The work of [14] uses Constraint Handling Rules (CHR) for relative time expressions in text already tagged with the methods of [2,3] in order to correct for mistakes and unresolved expressions. This approach used CHR as a programming language for flexible search back and forth in the text in order to find reference points, but did not develop a proper constraint solver as suggested in the present paper.

CHR has been used for semantic-pragmatic language analysis in combination with parsers written in Prolog's Definite Clause Grammar notation [15,16,17] or using CHR for parsing [18,19]. CHR based techniques have been used for extracting UML diagram from use case text [20,21]; this work includes an (although simplistic) approach to pronoun resolution that has similarities to relative or indirect time indications. Other applications of CHR for language processing include parsing from Property Grammars [22], analyzing biological sequences [23,24] and Chinese Word Segmentation [25].

3 A Constraint Language for Time Expressions

A constraint language called $CLP(Time)$ is defined upon Prolog using its extension of Constraint Handling Rules [1]. We take over the basic nomenclature of Prolog such as its terms, variables and operators, which may greatly enhance readability. In its present form, $CLP(Time)$ can be tested immediately in

language analyzers written with Prolog's Definite Clause Grammars and with pre-tagged text imported from other systems. An implementation under development is available at `http://www.ruc.dk/~henning/clptime`.

3.1 Datatypes and Basic Constraints of *CLP*(*Time*)

In the present version, the finest granule of time corresponds to dates, and distinguished types of terms are used to represent different units of time.

⟨date⟩ ::= `date(`⟨month⟩`,` n`)` ⟨decade⟩ ::= `decade(`⟨century⟩`,` n`)`
⟨month⟩ ::= `month(`⟨year⟩`,` n`)` ⟨century⟩ ::= `century(`n`)`
⟨year⟩ ::= `year(`⟨decade⟩`,` n`)`

The occurrences of "n" represent integer numbers of relevant size. For example, the term `date(month(year(decade(century(20),1),4),1),1)` represents the date of New Years day 2014.[2] Terms or subterms may be replaced by variables, so that `decade(C,9)` may represent "the nineties" in a yet unknown century. Such terms should only be instantiated as to represent legal times according to the Gregorian calendar since 1582. Type constraints are available, e.g., `month(M)` states that variable `M` can only be bound to terms of type month (type constraints for variables can be left out when the type is clear from context).

Expressions of the different types can be formed by adding or subtracting units of a number of granules of similar size. Examples:

```
date(month(year(decade(century(20),1),4),1),1) + 3 days
month(Y,1) + 12 months
```

The first one refers to the 4th of January 2014 and the second one to the month of January in the year following whatever year `Y` may end up representing. Equality terms of the same type can expressed using constraints `*=*` and order of time by `*<*` and `*=<*`. Examples:

```
Y1 *=* Y0 + 1 years
month(Y,3) *=<* M, M *<* month(Y1,5), Y1 *=* Y + 1 years
month(Y,3) *=<* M, M *<* month((Y + 1 years),5)
```

The first one means that `Y1` and `Y0` are years with `Y1` being one greater than `T0`. Two next ones state in different ways that `M` is a month between March in the year given by Y and April the following year; the comma understood as conjunction. Restrictions on possible values for numerical variables can be specified by interval notation as follows; notice that this compound expression denotes a single year with some uncertainty and not an interval of several years.

```
Y *=* year(decade(century(20),1),4) + N years, N in [3;6]
```

[2] This notation makes it easy to write different conditions involving time, but may seem clumsy for writing specific dates. Utilities are supplied for this so that the date shown can be created and assigned to variable `D` by `mk_date(2014-01-01,D)`.

Additional constraints are available for stating the day of the week, a leap year, etc.. For example,

```
D1 *=* D0 + N days, N in [1;7], dayOfWeek(D1,2)
```

means that date `D1` stands for "next Tuesday" relative to `D0`.

A constraint `failed(···)` is used for handling inconsistencies caused by problems in the text; it should not be used in specifications of dependencies, but used solely by the constraint solver; described in the end of section 5.

3.2 Using *CLP*(*Time*) with Definite Clause Grammars and Prolog

CLP(*Time*) can be used directly from Prolog programs, in particular its grammar notation as demonstrated in the following fragment.

```
event(E,D) --> event(E), [happened, on], date(D).
date(D) --> [the], ordinal(O), [of], month(Mn), year(Yn),
            {mk_year(Yn,Y), D *=* date(month(Y,Mn),O}.
year(_) --> [].
year(N) --> [N], {integer(N)}.
event(venus_transit) --> [the, transit, of, 'Venus'].
```

Parsing the fragment such as "The transit of Venus happened on the sixth of June", produces a syntax tree node `event(`e`,`d`)`, where e describes the event and d a date whose value is to be determined by the constraint solver. Calling a constraint within the curly bracket part of a grammar rule means to cast it off into the constraint store, so that the constraint solver can evaluate it.

Pre-tagged text may be converted into a Prolog list and processed by grammars as above, adapted to take the different tags into account. CHR can also be used for traversing a text represented as token constraints indexed by position numbers as done by [14] or using CHR Grammars [19].

4 Semantics and Evaluation of Time Expressions

Here we demonstrate how *CLP*(*Time*) can model time dependencies in different sorts of texts. Since this paper is not about syntax analysis, we show text fragments (first column below) together with constraints (second column) modelling their content with respect to time and indicate the solutions (third column) produced incrementally by the solver. The function symbols that define the data types for the different time units are abbreviated to save space.

4.1 Narrative with Initial Time Indication

The following shows a text where every time expression can be evaluated immediately from the value of the previous one, similarly to the narrative mode of HeidelTime [2,3].

It all began in 1864 ...	`Y0 *=* y(de(c(18),6),4)`	`Y0 = y(de(c(18),6),4)`
... three years later ...	`Y1 *=* Y0 + 3 years`	`Y1 = y(de(c(18),6),7)`
... the 17th of May that year ...	`D2 *=* d(m(Y1,5),17)`	`D2 = d(m(y(de(c(18),6),7),5),17)`

4.2 Narrative without Anchor Time or with Anchor Time at the End

Also without a given anchor time, *CLP*(*Time*) and its constraint solver still give meaningful output as it propagates also partly know information as far as possible.

... some year ...	`Y0`	`Y0` (uninstantiated)
... three years later ...	`Y1 *=* Y0 + 3 years`	`Y1 *=* Y0 + 3 years`
... the 17th of May that year ...	`D2 *=* d(m(Y1,5),17)`	`D2 = d(m(Y1,5),17)`
... the 18th of June the following year ...	`Y3 *=* Y1 + 1 years` `D3 *=* d(m(Y3,6),18)`	`Y3 *=* Y0 + 4 years` `D3 = d(m(Y3,6),18)`

... which was 1867 ...	`Y3 *=* y(de(c(18),6),8)`	`Y3 = y(de(c(18),6),8)`
		`Y1 = y(de(c(18),6),7)`
		`Y0 = y(de(c(18),6),4)`
		`D3 =`
		`d(m(y(de(c(18),6),8),6),18)`
		`D2 =`
		`d(m(y(de(c(18),6),7),5),17)`

If the story had ended immediately before the last phrase, the collected result can still be taken as meaningful output. When a time point that serves as anchor is introduced at the end, everything resolves into definite times.

The following examples show that the solver adds offsets to dates whenever it is safe, and propagates other safe information, but needs to delay in case an end of a month, whose number of days is uncertain, is exceeded.

—	`D1 *=*` `d(m(y(De,7),2),20) + 10 days`	`D1 =` `d(m(y(De,7),3),2)`

—	`D2 *=*`
	`d(m(y(c(19,De),Y),2),20)`
	`+ 10 days`
		`D2 *=*`
		`d(m(y(c(19,De),Y),2),20)`
		`+ 10 days`
		`D2 =`
		`d(m(y(c(19,De),Y),3),_)`

In the first example, we add 10 days to Feb. 20 in a year ending with -7 so we know that the month has 28 days, and the addition and shift of month is safe. In the second we do the same but concerned with a February month in some unknown year in the 20th century. The addition cannot be made, but the year and new month being 3=March can be propagated into `D2` which may trigger yet other evaluations to be made.

4.3 News Style Article with Mixed Anchor Points

Here we illustrate a news style text which also has narrative style relationships.

(day of newspaper is 2014-06-23)	`DayOfPrint *=*` `d(m(y(de(c(20),1),4),6),23)`	`DayOfPrint =` `d(m(y(de(c(20),1),4),6),23)`
... 2 years ago ...	`DayOfPrint *=* d(YearOfPrint,_)` `Y0 *=* YearOfPrint - 2 years`	`YearOfPrint = y(de(c(20),1),4)` `Y0 = y(de(c(20),1),2)`
... the Venus transit took place June 6 ...	`D1 *=*` `d(m(Y0,6),6)`	`D1 =` `d(m(y(de(c(20),1),2),6),6)`
... and a week later ...	`D2 *=* D1 + 7 days`	`D2 =` `d(m(y(de(c(20),1),2),6),13)`

This example may be varied, assuming that the issue date for the newspaper is not known, but there is a knowledge base about astronomical events and the dates when they occurred in terms of a predicate `event(`*event*`,`*date*`)`. Then we might have as result in the second line that `Y0` is still unknown, and then in the third line a call `event(venus_transit,d(m(Y0,6),6))` would instantiate `Y0=y(de(c(20),1),2)` and following that `YearOfPrint=y(de(c(20),1),4).`

5 A Constraint Solver for *CLP*(*Time*)

As mentioned, the constraint solver is implemented in Constraint Handling Rules [1] (CHR). For reasons of space, we can only give a very brief sketch of the principles. CHR can be understood as rewriting rules over constraint stores; it has different sorts of rules, but in the fragment shown below we use only simplification rules. A rule of form *constraints-before* `<=>` *guard* `|` *constraints-after* can apply if a collection of constraints matching the pattern *constraints-before* is found in the store and the test in *guard* succeeds; in that case *constraints-before* are replaced by *constraints-after*. The *after* part may also refer to auxiliary code written in Prolog. We show here some of the rules for processing constraints of the form "$\cdots$ `*=*` *date* + *n* `days`" (they are preceded by rules that brings all applications of plus into this form). The version shown here is slightly simplified as it ignores constraints of the form "`in` *numeric-interval*".

```
T *=* date(M,Dn) + N days  <=>
      ground(Dn), ground(N), DnN is Dn+N,
      lastSafeDateInMonth(M,Max),
      DnN =< Max
    |
      T *=* date(M,DnN).

T *=* date(M,Dn) + N days  <=>
      ground(Dn), ground(N), DnN is Dn+N,
      lastDateInMonth(M, Max), DnN =< Max
    |
      T *=* date(M,DnN).

T *=* date(M,Dn) + N days  <=>
      ground(Dn), ground(N), DnN is Dn+N,
      lastDateInMonth(M,Max), DnN > Max
    |
      Dn1 is DnN-Max-1,
      T *=* date(M1,1) + Dn1 days,
      M1 *=* M + 1 months.
```

The two first rules apply when an addition can be made giving a new day-of-month guaranteed not to lead into the following month. The first one uses the auxiliary predicate `lastSafeDateInMonth`; if the month is completely unspecified or is a Feb. in an unknown year, it returns 28, and when more information is present it returns the highest safe number (28 or 29 for Feb., 30 or 31 for others). The second rule takes care of cases not caught by the first rule using a more

precise test (due to the first rule, actually only involved when the incremented date will be a 29th, 30 or 31). The last rule applies when the incremented date is in a later month.

Inconsistency Handling

An inconsistent set of constraints may arise due to misconceptions in a text. A basic constraint solver, incapable of handling inconsistency, may include the following rule.

```
T *=* month(Y,Mn) <=> T=month(Y,Mn).
```

It is executed when an absolute date is entered or an increment has been successfully added. An inconsistency manifests itself by the left and righthand sides being non-unifiable, and thus the execution of the equality predicate will result in the whole computation failing – which is logically correct as there is no solution to the total set of constraints. We can avoid this and still get a partial solution using a constraint `failed(···)` in the following way.

```
T *=* month(Y,Mn) <=> (T=month(Y,Mn) -> true ; failed((T=month(Y,Mn))).
```

The arrow-semicolon notation stands for if-then-else, so if the unification succeeds, everything is fine, otherwise we record that failure would have occurred if the unification had been enforced. Evaluations before and after the critical point are still executed. The constraint solver can be extended, so it indicates the position of a possible source of inconsistency in the text.

6 Conclusions

A constraint logic language $CLP(Time)$ is introduced which can specify a wide range of dependencies between time expression in a natural language text. A constraint solver is demonstrated that can evaluate these expressions independently of the order in which anchoring times may be introduced, and it can produce meaningful results also when such anchors are absent. These properties, which do not hold for some state-of-the-art systems, are inherent in constraint solving, so the main message of this paper is to advocate constraint technologies for resolving time expressions.

$CLP(Time)$ is intended to be used together with language analyzers capable of setting up relevant constraints, which is not a trivial task, and the results obtained will critically depend on the quality of the language analyzer. However, realistic tests together with a capable analyzer are yet to be made. Fragments of the constraint solver programmed in a rule-based fashion were shown, indicating a highly modular and easily extendible structure. This makes it a reasonable task to add new facilities, for example to match special usages in a particular language or more specific, idiomatic expressions used in a particular corpus.

For the implementation, we avoided finite domain techniques that have some drawbacks for calendric data. These techniques are not fitted for an incremental, detailed propagation of values necessary to figure out, say, that adding two days to a date which is the 28th of some yet unknown month in the 1900 years will preserve the century (and carry over variables referring to decade, year in decade and perhaps to the month), as we have demonstrated. Furthermore, a finite domain constraint solver typically works in two phases, first it collects and simplifies constraints, then at the end there is a grounding phase that attempts to assign concrete values to the variables; this is opposite to the incremental propagation of as much information as possible as we have aimed at. The modular structure of an implementation in Constraint Handling Rules makes it possible to add rules one by one for different special cases taking care of the particularities of calendric data and relationships that distinguish them from plan integer arithmetic. As mentioned, $CLP(Time)$ and its constraint solver is currently being extended to handle different sorts of time intervals that we did not describe in the present paper. Our approach to handle inconsistency introduces a new research topic of identifying a best or minimal repair of an inconsistent set of time constraints, where our current version just pics an arbitrary one determined by the constraint solver's internal evaluation order.

References

1. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
2. Strötgen, J., Gertz, M.: Heideltime: High quality rule-based extraction and normalization of temporal expressions. In: Proceedings of the 5th International Workshop on Semantic Evaluation. SemEval '10, Stroudsburg, PA, USA, Association for Computational Linguistics (2010) 321–324
3. Strötgen, J., Gertz, M.: Multilingual and cross-domain temporal tagging. Language Resources and Evaluation **47**(2) (2013) 269–298
4. de Medeiros Caseli, H., Villavicencio, A., Teixeira, A.J.S., Perdigão, F., eds.: Computational Processing of the Portuguese Language - 10th International Conference, PROPOR 2012, Coimbra, Portugal, April 17-20, 2012. Proceedings. In de Medeiros Caseli, H., Villavicencio, A., Teixeira, A.J.S., Perdigão, F., eds.: PROPOR. Volume 7243 of Lecture Notes in Computer Science., Springer (2012)
5. Hovy, D., Fan, J., Gliozzo, A.M., Patwardhan, S., Welty, C.A.: When did that happen? - Linking events and relations to timestamps. In Daelemans, W., Lapata, M., Màrquez, L., eds.: EACL, The Association for Computer Linguistics (2012) 185–193
6. Chambers, N.: Labeling documents with timestamps: Learning from their time expressions. In: ACL (1), The Association for Computer Linguistics (2012) 98–106
7. Verhagen, M., Gaizauskas, R.J., Schilder, F., Hepple, M., Moszkowicz, J., Pustejovsky, J.: The TempEval challenge: identifying temporal relations in text. Language Resources and Evaluation **43**(2) (2009) 161–179
8. Pustejovsky, J., Castaño, J.M., Ingria, R., Sauri, R., Gaizauskas, R.J., Setzer, A., Katz, G., Radev, D.R.: Timeml: Robust specification of event and temporal expressions in text. In Maybury, M.T., ed.: New Directions in Question Answering, AAAI Press (2003) 28–34

9. Mazur, P.P., Dale, R.: LTIMEX: representing the local semantics of temporal expressions. In Ganzha, M., Maciaszek, L.A., Paprzycki, M., eds.: FedCSIS. (2011) 201–208
10. Eshghi, K.: Abductive planning with event calculus. In Kowalski, R.A., Bowen, K.A., eds.: ICLP/SLP, MIT Press (1988) 562–579
11. Allen, J.F.: Towards a general theory of action and time. Artif. Intell. **23**(2) (1984) 123–154
12. Han, B., Lavie, A.: A framework for resolution of time in natural language. ACM Trans. Asian Lang. Inf. Process. **3**(1) (2004) 11–32
13. Bry, F., Rieß, F.A., Spranger, S.: CaTTS: calendar types and constraints for web applications. In Ellis, A., Hagino, T., eds.: WWW, ACM (2005) 702–711
14. van de Camp, M., Christiansen, H.: Resolving relative time expressions in Dutch text with Constraint Handling Rules. In Duchier, D., Parmentier, Y., eds.: CSLP. Volume 8114 of Lecture Notes in Computer Science., Springer (2012) 166–177
15. Christiansen, H., Dahl, V.: HYPROLOG: A new logic programming language with assumptions and abduction. In Gabbrielli, M., Gupta, G., eds.: ICLP. Volume 3668 of Lecture Notes in Computer Science., Springer (2005) 159–173
16. Christiansen, H.: Executable specifications for hypothesis-based reasoning with Prolog and Constraint Handling Rules. J. Applied Logic **7**(3) (2009) 341–362
17. Christiansen, H.: Constraint programming for context comprehension. To appear (2014)
18. Christiansen, H.: Abductive language interpretation as bottom-up deduction. In Wintner, S., ed.: Natural Language Understanding and Logic Programming. Volume 92 of Datalogiske Skrifter., Roskilde, Denmark (2002) 33–47
19. Christiansen, H.: CHR Grammars. Int'l Journal on Theory and Practice of Logic Programming **5**(4-5) (2005) 467–501
20. Christiansen, H., Have, C.T., Tveitane, K.: From use cases to UML class diagrams using logic grammars and constraints. In: RANLP '07: Proc. Intl. Conf. Recent Adv. Nat. Lang. Processing. (2007) 128–132
21. Christiansen, H., Have, C.T., Tveitane, K.: Reasoning about use cases using logic grammars and constraints. In: CSLP '07: Proc. 4th Intl. Workshop on Constraints and Language Processing. Volume 113 of Roskilde University Computer Science Research Report. (2007) 40–52
22. Dahl, V., Blache, P.: Implantation de grammaires de propriétés en CHR. In Mesnard, F., ed.: Actes des 13èmes Journées Francophones de Programmation en Logique avec Contraintes, Hermes (2004)
23. Bavarian, M., Dahl, V.: Constraint based methods for biological sequence analysis. Journal of Universal Computing Science **12**(11) (2006) 1500–1520
24. Dahl, V., Gu, B.: A CHRG analysis of ambiguity in biological texts. In: CSLP '07: Proc. 4th Intl. Workshop on Constraints and Language Processing. Volume 113 of Roskilde University Computer Science Research Report. (2007) 53–64
25. Christiansen, H., Li, B.: Approaching the chinese word segmentation problem with CHR grammars. In: CSLP 2011: Proc. 4th Intl. Workshop on Constraints and Language Processing. Volume 134 of Roskilde University Computer Science Research Report. (2011) 21–31