Constraint Handling Rules and Tabled Execution

Tom Schrijvers[1]*, David S. Warren[2]

[1] Dept. of Computer Science, K.U.Leuven, Belgium
[2] Dept. of Computer Science, State University of New York at Stony Brook, USA

Abstract. Both Constraint Handling Rules (CHR) and tabling - as implemented in XSB - are powerful enhancements of Prolog systems, based on fix point computation. Until now they have only been implemented in separate systems. This paper presents the work involved in porting a CHR system to XSB and in particular the technical issues related to the integration of CHR with tabled resolution. These issues include call abstraction, answer projection, entailment checking, answer combination and tabled constraint store representations. Different optimizations related to tabling constraints are evaluated empirically. The integration requires no changes to the tabling engine. We also show that the performance of CHR programs without tabling is not affected. Now, with the combined power of CHR and tabling, it is possible to easily introduce constraint solvers in applications using tabling, or to use tabling in constraint solvers.

1 Introduction

XSB (see [19]) is a standard Prolog system extended with tabled resolution. Tabled resolution is useful for recursive query computation, allowing programs to terminate in many cases where Prolog does not.

Parsing, program analysis, model checking, data mining, diagnosis and many more applications benefit from tabled resolution. We refer the reader to [1] for a coverage of XSB's SLG execution strategy.

Constraint Handling Rules, or CHR for short, are a high level rule-based language (see [10]). As opposed to the top-down execution of XSB, it performs a bottom-up fixpoint computation. While CHR has currently many applications, it has been designed for writing constraint solvers in particular. Indeed, its compact and declarative syntax is excellent for prototyping succinct constraint solvers. Although its performance is not on par with constraint solvers written in lower-level languages, its flexibility and ease of use do favor a wider use. We refer the reader to [11] for an overview of CHR.

CHR is not a self-contained but an embedded language. Although versions of CHR for Haskell and Java do exist, Prolog is its original and natural host. Just as the traditional combination of Constraint and Logic Programming [14], the

* Research Assistant of the Fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

combination of CHR with Logic Programming seems to be the most powerful and expressive combination.

However, by integrating CHR with not just any standard Prolog system, but XSB with tabled resolution, we get an even more powerful and expressive system. This system integrates both the bottom-up and top-down fixpoint computations, the favorable termination properties of XSB and the constraint programming of CHR. This combined power enables programmers to easily write highly declarative programs that are easier to maintain and extend.

This proposed integration implies quite a number of implementation challenges that have to be dealt with. Firstly, a CHR system for non-tabled use in XSB is needed, as it does not come with one already. In Section 2 we explain how an existing CHR system was ported to XSB with minimal changes to that CHR system. The attributed variables implementation in XSB was extended to full functionality in the process. Benchmarks show that no performance overhead is incurred for non-tabled use and that it is even on par with the commercial SICStus Prolog system.

Secondly, the CHR system requires integration with tabled execution. In Section 3 we present the different interaction issues. We propose a solution to the conflict of the global CHR store with the required referential transparency of tabled predicates. Also two representations of tabled constraints are analyzed and a mechanism for answer constraint projection is presented. In addition, we study issues regarding answer entailment checking and advanced techniques for aggregate-like answer combination. The performance impact of tabling and several of the mentioned concepts are measured on a small constraint program. Throughout, we propose a declaration-based approach for enabling particular mechanisms that, in the spirit of both CHR and XSB, preserves the ease of use.

Finally, we conclude this paper in Section 4 and discuss related and possible future work.

2 The CHR System

Initially the CHR system described in this paper was written for the hProlog system. hProlog is based on dProlog (see [8]) and intended as an alternative backend to HAL (see [7]) next to the current Mercury backend. The initial intent of the implementation of a CHR system in hProlog was to validate the underlying implementation of dynamic attributes (see [6]).

The hProlog CHR system consists of a preprocessor and a runtime system:

- The preprocessor compiles embedded CHR rules in Prolog program files into Prolog code. The compiled form of CHR rules is very close to that of the CHR system by Christian Holzbaur, which is used in SICStus [13] and YAP [3]. The preprocessor allows for experimentation with optimized compilation of CHR rules, both through static inference and programmer declarations.
- The runtime system is nearly identical to that of Christian Holzbaur: suspended constraints are stored in a global constraint store. Variables in suspended constraints have attributes on them that function as indexes into this global

store. Binding of these attributed variables causes the suspended constraints on them to trigger again.

Little difficulty was experienced while porting the preprocessor and runtime system from hProlog to XSB. The main problem turned out to be XSB's overly primitive interface for attributed variables: it did not support attributes in different modules. Moreover, the actual binding of attributed variables was not performed during the unification, but it was left up to the programmer of the interrupt handler. This causes unintuitive and unwanted behavior in several cases: while the binding is delayed from unification to interrupt handling, other code can be executed in between that relies on variables being bound, e.g. arithmetic. Due to these problems of the current XSB attributed variables, it was deemed acceptable to model the attributed variables interface and behavior more closely to that of hProlog. This facilitated the porting of the CHR system considerably.

2.1 Performance evaluation

We have not adapted the compilation scheme of the CHR preprocessor in any way to accommodate the integration of CHR with tabling (see Section 3). We have only given in to a few minor changes for performance reasons, such as replacing calls to `once(G)` with `(G -> true)` as XSB does not inline the former.

Hence we show that the performance of CHR without tabling is not compromised in XSB. To do so we compare the results of eight benchmarks [17] in XSB with those in hProlog, the origin of the CHR preprocessor, and SICStus, the origin of the runtime system and one of the first systems to feature CHR.

The following causes for performance differences are to be expected:

- Firstly, we expect the outcome to be most heavily influenced by the relative performance difference on Prolog code as the CHR rules are compiled to Prolog: we have observed that plain Prolog in SICStus is on average 1.39 times faster than in XSB and hProlog is 1.85 times faster than XSB.
- Secondly, the results may be influenced by the slightly more powerful optimizations of the CHR preprocessor in hProlog and XSB. To eliminate these effects we have disabled all advanced optimizations not performed by the SICStus CHR compiler. In addition, the checking of guard bindings has been disabled in both systems. This does not affect the benchmarks, since no binding or instantiation errors occur in the guards. This increases the fairness of comparison since hProlog's analysis of redundant checks is more powerful and it does not intercept instantiation errors.
- Thirdly, the implementation and representation of attributed variables differs between the three systems. The global constraint store of CHR is represented as an attributed variable and it may undergo updates each time a new constraint is imposed or a constraint variable gets bound. Hence, the complexity and efficiency of accessing and updating attributed variables easily dominates the overall performance of a CHR program if care is not taken. Especially the length of reference chains has to be kept short, as otherwise

accessing the cost of dereferencing the global store may easily grow out of bounds.

In hProlog much care has been taken in choosing a representation and corresponding implementation of the necessary operations, to take these considerations into account. SICStus and XSB have different low-level representations that do not allow to limit the cost of dereferencing equally well.

Table 1 shows the results for the benchmarks. All measurements have been made on an Intel Pentium 4 2.00 GHz with 512 MB of RAM. Timings are in milliseconds. The Prolog systems used are hProlog 2.4, SICStus 3.11.0 and our extension of XSB 2.6.

From the results we learn that hProlog is the fastest for CHR, as was the case for plain Prolog. Both SICStus and XSB are much slower than hProlog for CHR than for plain Prolog. However, on average XSB is SICStus's equal.

The huge difference between hProlog and the other two systems cannot simply be explained by the difference in runtimes of plain Prolog. The representation of attributed variables and the implementation of operations on them, seems to play a very important role in most of the benchmarks. The more efficient implementation of hProlog and XSB's CHR runtime and generated code mostly explains the smaller gap between SICStus and XSB.

All in all, XSB is not far behind SICStus in performance of CHR programs. This is quite remarkable considering that SICStus is clearly faster for plain Prolog programs. Future optimizations of the XSB CHR compiler may improve XSB's CHR performance even further. However, it is certainly worthwhile to consider changing the low-level implementation of attributed variables in XSB to follow more closely that of hProlog. Such a change would be orthogonal to the CHR system and other applications of attributed variables would benefit as well.

Table 1. Runtime performance of 8 CHR benchmarks in hProlog, SICStus and XSB.

Benchmark	hProlog		SICStus		XSB	
	runtime	relative	runtime	relative	runtime	relative
bool	870	100.0%	2,010	231.0%	1,729	198.7%
fib	1,240	100.0%	1,620	130.6%	2,509	202.3%
fibonacci	1,510	100.0%	4,260	282.1%	3,500	231.8%
leq	1,020	100.0%	1,250	122.5%	1,889	185.2%
primes	960	100.0%	1,870	194.8%	2,609	271.8%
ta	1,190	100.0%	2,240	188.2%	2,378	199.8%
wfs	850	100.0%	1,620	190.6%	2,039	239.9%
zebra	1,500	100.0%	7,050	470.0%	3,578	238.5%
average	-	100.0%	-	226.2%	-	221.0%

3 CHR and Tabled Execution

The main challenge of introducing CHR in XSB is integrating the forward chaining fixpoint computation of the CHR system with the backward chaining fixpoint computation of tabled resolution.

A similar integration problem has been solved in [4], which describes a framework for constraint solvers written with attributed variables for XSB. The name Tabled Constraint Logic Programming (TCLP) is coined in that publication.

The main difference for the programmer between CHR and attributed variables for developing constraint solvers, i.e. the fact that CHR is a much higher level language, should be carried over to the tabled context. Hence tabled CHR should be a more convenient paradigm for programming constraint solvers in than TCLP with attributed variables.

Indeed, the internals that are presented in the following can easily be hidden from the user. All the user then needs to supply is the following declaration:

```
:- table_chr f(_,_) with Options.
```

meaning that the predicate `f/2` should be tabled and it involves CHR constraints. A list of additional options may be provided. The meaning of this declaration and the possible options are explained in the rest of this section.

In [4] the general framework specifies three operations to control the tabling of constraints: call abstraction, entailment checking of answers and answer projection. It is left up to the constraint solver programmer to implement these operations with respect to his solver implementation.

In the following we formulate these operations in terms of CHR. The operations are covered in significant detail as the actual CHR implementation and the representation of the global CHR constraint store are taken into account. Problems that have to be solved time and again for attributed variable constraint solvers are solved once and for all for CHR constraint solvers. Hence integrating a particular CHR constraint solver requires much less knowledge of implementation intricacies and decisions can be made on a higher level.

The different steps in handling a call to a tabled predicate are depicted in Figure 1. The different steps are explained in the rest of this section. Section 3.1 covers the representation of CHR constraint stores in tables. Next Section 3.2 explains how a call to a tabled predicate is abstracted and how the answer to the abstracted call is related to the original call. Section 3.3 proposes a high-level way of specifying projection and Section 3.4 discusses entailment. Finally, Section 3.5 finishes with an evaluation of tabling with several optimizations on a small shipment problem.

3.1 Tabled Store Representation

As constraints can be part of a call or an answer of a tabled predicate, some representation for them in call and answer tables is needed. In general, this representation should allow comparing the constraints in a call with all call patterns in the call table to select either a variant (under variant based tabling) or the most specific generalization (under subsumption based tabling). Furthermore, it should be possible to convert from the ordinary representation of the constraints and back, for insertion into the call table and retrieval from the answer table.

In the following we investigate two possible representations and discuss the complexity of variant checking.

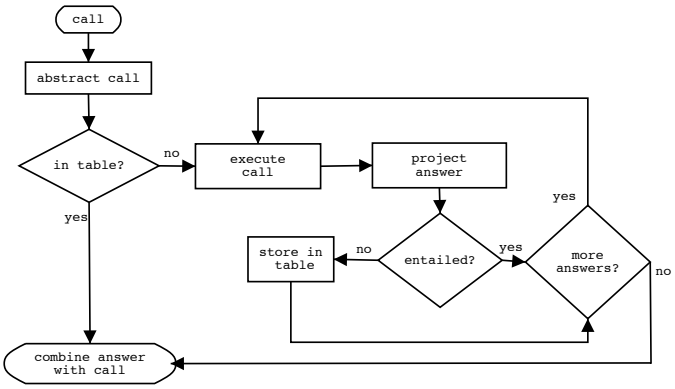

Fig. 1. flowchart

Two tabled CHR store representations Before considering constraint store representations for tables, it is necessary to know a little about the ordinary representation of the CHR constraint store. The global CHR constraint store is an updatable term, containing suspended constraints grouped by functor. Each suspended constraint is represented as a suspension term, including the following information:

- the unique ID for sorting and equality testing
- the goal to execute when triggered, this goal contains the suspension itself as an argument, hence creating a cyclic term.
- the propagation history containing for each propagation rule the tuple of identifiers of other constraints that this constraint has interacted with

Furthermore, variables involved in the suspended constraints behave as indexes into the global store: they have the suspensions stored in them as attributes.

One piece of information that is not maintained during the normal CHR execution is the order in which constraints have been called. We will not do so for the sake of tabling either as this would overly complicate matters. Hence, we need to restrict ourselves to confluent[3] programs to avoid unpredictable behavior.

Two different tabled CHR store representations have been explored with: the *suspension* representation and the *naive* representation. A discussion of their respective merits and weaknesses as well as an evaluation follow.

Suspension representation Here we aim to keep the tabled representation as close as possible to the ordinary representation. The idea here is to maintain the propagation history of the tabled constraints. In that way no unnecessary re-firing of propagation rules will occur after the constraints have been retrieved from the table.

However, it is not possible to just store the ordinary constraint suspensions in the table as they are. Firstly, the tables do not deal with cyclic terms. This can be dealt with by breaking the cycles before storage and resetting them

[3] A set of rules is confluent if the order in which the constraints are imposed does not matter for the final state of the constraint store.

after fetching. Secondly, the unique identifiers have to be replaced after fetching by fresh ones as multiple calls would otherwise create multiple copies of the same constraints all with identical identifiers. Fortunately, attributed variables themselves can be stored in tables (see [5]).

Naive Representation The naive representation aims at keeping the information in the table in as simple a form as possible: for each suspended constraint only the goal to pose this constraint is retained in the table. It is easy to create this goal from a suspension and easy to merge this goal back into another constraint store: it needs only to be called.

When necessary the goal will create a suspension with a fresh unique ID and insert it into the constraint store. However in many cases it may prove unnecessary to do so because of some simplification through interaction with constraints in the calling environment.

The only information that is lost in this representation is the propagation history. This may lead to multiple propagations for the same combination of head constraints. For this to be sound, a further restriction on the CHR rules is required: they should behave according to set semantics, i.e. the presence of multiple identical constraints should not lead to different answers modulo identical constraints.

Evaluation of both representations To measure the relative performance of the two presented representations, consider the following two programs:

prop

```
:- constraints a/1.
  a(0) <=> true.
  a(N) ==> N > 0
       | M is N - 1, a(M).

  p(N) :- a(N).
```

simp

```
:- constraints a/1, b/1.
  b(0) <=> true.
  b(N) <=> N > 0
       | a(N), M is N - 1, b(M).

  p(N) :- b(N).
```

For both programs the predicate `p(N)` puts the constraints `a(1)`...`a(N)` in the constraint store. The `prop` program uses a propagation rule to achieve this while the `simp` program uses an auxiliary constraint `b/1`. The non-tabled version of the query `p(N)` or `a(N)` has time complexity $\mathcal{O}(N)$ for both the `simp` and the `prop` program.

The two possible representations for the answer constraint store can be specified in the tabling declaration as follows:

```
:- table_chr p(_) with [representation(suspension)].
```

and

```
:- table_chr p(_) with [representation(naive)].
```

Table 2. Evaluation of the two tabled store representations.

	`representation(suspension)`		`representation(naive)`	
program	runtime	space	runtime	space
prop	150	2,153,100	1,739	270,700
simp	109	1,829,100	89	270,700

Table 2 gives the results for the tabled query `p(400)` : runtime in milliseconds and space usage of the tables in bytes. For both programs the answer table contains the constraint store with the 400 `a/1` constraints.

Most of the space overhead is due to the difference in representation: a suspension contains more information than a simple call. However, the difference is more or less a constant factor. The only part of a suspension in general that can have a size greater than $\mathcal{O}(1)$ is the propagation history, that for `prop` is limited to remembering that the propagation rule has been used. For the `simp` program the propagation history is always empty.

The runtime of the `prop` version with the suspension representation is considerably better than that of the version with the naive representation. In fact, there is a complexity difference. When the answer is retrieved from the table for the suspension representation, the propagation history prevents re-propagation. Hence answer retrieval is $\mathcal{O}(N)$. For the naive representation on the other hand, every constraint `a(I)` from the answer will start propagating and the complexity of answer retrieval becomes $\mathcal{O}(N^2)$.

On the other hand, for `simp` propagation history plays no role. The runtime overhead is mostly due to the additional overhead of the pre- and post-processing of the suspension representation as opposed to the simpler form of the naive representation.

Variant checking The need to check whether two constraint stores are variants of each other may arise at two occasions:

- With no or only partial call abstraction (see Section 3.2) a constraint store is part of the call to the tabled predicate. The tabling system then needs to check whether a previous call with a variant of that constraint store appears in a table. If that is the case, the answer to the previous call can be reused.
- A limited form of entailment checking (see Section 3.4) is to check whether a new answer constraint store is a variant of any previous answer constraint store for the same call. In that case the new answer can be discarded.

We can consider this equality checking with the previously presented naive tabled representation of constraints. In that representation the tabled constraints are kept as a list of goals that impose the constraints. Any permutation of this list represents the same constraint store. If two constraint stores are identical modulo variable renaming, then they are variants.

In general, variant checking of constraint stores has exponential complexity. A naive algorithm would be to consider all permutations of one constraint store. If any one of the permutations equals the other constraint store, both are identical. With heuristics this algorithm could be improved and for particular constraints or even applications algorithms with a better complexity may exist. However further exploration falls outside of the scope of this paper. The problem can be ignored altogether, with possible duplication in tables as a consequence, or only partially tackled, e.g. by simple sorting and pattern matching.

3.2 Call Abstraction

Call abstraction replaces the called goal with a call to a more general goal followed by an operation that ensures that only the answer substitutions applicable to the particular call are retained. At the level of plain Prolog, abstraction means not passing certain bindings in to the call. E.g. `p(q,A)` can be abstracted to `p(Q,A)`. This goal has then to be followed by `Q = q` to ensure that only the appropriate bindings for `A` are retained.

In XSB call abstraction is a means to control the number of tables. When a predicate is called with many different instantiation patterns, a table is generated for each such call instantiation pattern. Thus it is possible that the information for the same fully instantiated call is present many times in tables for different call instantiation patterns. However, this duplication in the tables can be avoided by using call abstraction to restrict to a small set of call instantiation patterns.

For constraint logic programming, call abstraction can be extended from bindings to constraints: abstraction means removing some of the constraints on the arguments. Consider for example the call `p(Q,A)` with constraint `Q leq N` on `Q`. This call can be abstracted to `p(Q',A)`, followed by `Q'=Q` to reintroduce the constraint.

Abstraction is especially of value for those constraint solvers where the number of constraints on a variable can be much larger than the number of different bindings for that variable. Consider for example a finite domain constraint solver with constraint `domain/2`, where the first argument is a variable and the second argument the set of its possible values. If the variable can be bound to at most n values it can take as many as 2^n different `domain/2` constraints, one for each subset of values. Thus many different tables would be needed to cover every possible call pattern.

Varying degrees of abstraction are possible and may depend on the particular constraint system or application. Full constraint abstraction, i.e. the removal of all constraints from the call, is generally more suitable for CHR for the following reasons:

- CHR rules do not require constraints to be on variables. They can be on ground terms or atoms as well. It is not straightforward to define abstraction for ground terms as these are not necessarily passed in as arguments but can just as well be created inside the call. Hence there is no explicit link with the call environment, while such a link is needed for call abstraction. As such, only no abstraction or full constraint abstraction seem suitable for CHR.
- Full constraint abstraction is preferable when the previously mentioned table blow-up is likely.
- As mentioned in the previous section, variant checking of constraint stores can have exponential complexity.

Moreover, it may be quite costly for certain constraint domains to sort out what constraints should be passed in to the call or abstracted away, involving transitive closure computations of reachability through constraints. Hence often full abstraction is cheaper than partial abstraction.

For CHR full abstraction requires the execution of the tabled predicate with an empty constraint store. If the call environment constraint store were used, interaction with new constraints would violate the assumption of full abstraction.

The code below shows how a predicate `p/1` that requires tabling:

```
:- table p/1.
p(X) :- ...
```

is transformed into two predicates, where the first one is called, takes care of the abstraction, calls the second predicate and afterwards combines the answer with the previously abstracted away constraints.

```
p(X) :-
        current_chr_store(CallStore)
        set_empty_chr_store,
        tabled_p(X1,AnswerStore),
        set_chr_store(CallStore),
        insert_answer_store(AnswerStore),
        X1 = X.

:- table tabled_p/2.
tabled_p(X,S_A) :- ...
```

The further implementation of `tabled_p` will be discussed in the next section. The given answer constraints are merged into the current global CHR constraint store by the predicate `insert_answer_store/1`. Given the naive representation discussed in the previous section, this boils down to calling a list of goals to impose the constraints.

The above transformation is not involved at all and can be easily automated, hiding the internals of the CHR-tabling integration from the user. All the user needs to supply is the arguments that are CHR constraint variables. We propose the following declaration:

```
:- table_chr f(_,chr) with Options.
```

meaning that the predicate `f/2` should be tabled, its first argument is an ordinary Prolog variable and its second argument is a CHR constraint variable of which all the constraints are abstracted away.

3.3 Answer Projection

Often one wants to project the answer constraint store on the non-local variables of the call. The usual motivation is that constraints on local variables are meaningless outside of the call. The constraint system should be complete so that no unsatisfiable constraints can be lost through projection.

For tabling there is an additional and perhaps even more pressing motivation for projection: a predicate with an infinite number of different answers may be turned into one with just a finite number of answers by throwing away the constraints on local and unreachable variables.

In some cases it may suffice to look at the constraints in the store separately and given a set of non-local variables to decide whether to keep the constraint or not. In those cases it may be convenient to exploit the operational semantics of CHR and implement projection as a `project/1` constraint with the list of

variables on which to project as an argument. Simpagation rules can then be used to look at and decide what constraints to remove. A final simplification rule at the end can be used to remove the `project/1` constraint from the store.

The following example shows how to project away all leq/2 constraints that involve arguments not contained in a given set $Vars$:

```
project(Vars) \ leq(X,Y) <=> \+ (member(X,Vars),member(Y,Vars)) | true.
project(Vars)            <=> true.
```

Besides removal of constraints more sophisticated operations such as weakening are possible. E.g. consider a set solver with two constraints: `in/2` that requires an element to be in a set and `nonempty/1` that requires a set to be non-empty. The rules for projection could include the following weakening rule:

```
project(Vars) \ in(Elem,Set) <=>
                 member(Set,Vars), \+ member(Elem,Vars) | nonempty(Set).
```

The predicate `tabled_p` would then look like:

```
tabled_p(X,S_A) :- orig_p(X).
                   project([X]),
                   extract_store_representation(S_A).
```

Here the predicate `extract_store_representation/1` converts from the ordinary global store representation to the naive tabled store representation, discussed in Section 3.1.

This approach is of course not general in the sense that certain constraint domains may need more information than just the variables to project on, such as more intricate knowledge of the contents of the constraint store. In addition it relies on operational semantics and ordering of constraints. However, it is a rather compact and high level notation and as such it might be possible to infer conditions on its usage under which the technique is provably correct.

The `project/1` call could easily be added by the automatic transformation of the tabled predicate if the user supplies the `projection` option:

```
:- table_chr p(chr) with [projection].
```

3.4 Entailment checking and other answer combinations

In some cases some of the answers computed for a tabled predicate are redundant and so need not be saved. Indeed there are cases in which for any tabled call only one answer needs to be maintained. Consider for example that the answer `p(a,X)` is already in the table of predicate `p/2`. Now a new answer, `p(a,b)` is found. This new answer is redundant as it is covered by the more general `p(a,X)` that is already in the table. Hence it is logically valid to not record this answer in the table, but to simply discard it. This does not affect the soundness or completeness of the procedure.

The idea of this answer subsumption technique is to reduce the number of answers in the table by replacing two (or more) answers by a single answer. Logically, the single new answer has to be equivalent to the disjunction of the replaced answers. For ordinary tabled Prolog, each answer H_i can be seen as a Herbrand constraint $Answer = Term$, e.g. $Answer = p(a, X)$. Now, for any two of these Herbrand constraints H_0 and H_1 the following two properties hold:

1. If the disjunction is equivalent to another constraint, that constraint is equivalent to one of the two constraints.
$$\exists H : H = H_0 \vee H_1 \Longleftrightarrow \exists i \in \{0,1\} : H = H_i$$
2. If the conjunction is equivalent to one of the two constraints, the disjunction is equivalent to the other.
$$\exists i \in \{0,1\} : H_0 \wedge H_1 = H_j \Longleftrightarrow H_0 \vee H_1 = H_{j-1}$$

These two properties suggest a possible strategy to compute the equivalent single answer of two answers: check whether the conjunction of two answers is equivalent to one of the two, then the other is the single equivalent answer. Otherwise, there is no single equivalent answer.

We can include CHR constraints in the same logically sound idea of answer subsumption. This path length computation will serve as an illustration:

```
dist(A,B,D) :- edge(A,B,D1), leq(D1,D).
dist(A,B,D) :- dist(A,C,D1), edge(C,B,D2), leq(D1 + D2, D).
```

Suppose appropriate rules for the `leq/2` constraint in the above program, where `leq` means less-than-or-equal. The semantics are that `dist(A,B,D)` holds if there is a path from `A` to `B` of length less than or equal to `D`. In other words, `D` is an upper bound on the length of a path from `A` to `B`.

If there is answer `dist(n1,n2,D) :- leq(d1, D)` already in the table and a new answer `dist(n1,n2,D) :- leq(d2, D)`, where `d1 =< d2`, is found, then this new answer is redundant. Hence it can be discarded. Again this does not affect the soundness, since logically the same answers are covered.

Operationally, the same strategy as proposed for ordinary Prolog can be used to reduce two answer constraint stores s_0 and s_1 to a single answer store s. At the end of the tabled predicate we merge a previous answer store s_0 with a new answer store s_1. After merging the store will be simplified and propagated to s by the available CHR rules. This combines the two answers into a new one. This mechanism can be used to check entailment of one of both answers by the other: if the combined answer store s is equal to one of the two, then that answer entails the other: $s_0 \wedge s_1 = s_i (i \in \{0,1\}) \Longrightarrow s_{1-i} = s_0 \vee s_1$.

The predicate `insert_answer_store/1`, mentioned in Section 3.2, can be used for the conjunction of two constraint stores. We assume that one store is the current global CHR constraint store.

When the above two answers of the `dist/3` predicate are merged, the following rule `leq/2` rule will simplify the constraint store to retain the more general answer: `leq(X,D1) \ leq(X,D2) <=> D1 =< D2 | true.`

Note that the `dist/3` program would normally generate an infinite number of answers for a cyclic graph, logically correct but not terminating. However, if it is tabled with answer subsumption, it does terminate for non-negative weights. Not only does it terminate, it only produces one answer, namely `dist(n1,n2,D) :- leq(d,D)` with `d` the length of the shortest path. Indeed, the predicate only returns the optimal answer.

The above strategy is a sound approach to finding a single constraint store that is equivalent to two others. However, it is not complete: a single constraint store may be equivalent to the disjunction of two others, while it is not equivalent to one of the two. This is because, the first property for the Herbrand constraints

does not hold for all constraint solvers, e.g. $leq(X,Y) \vee leq(Y,X) = true$. Nevertheless it is a rather convenient strategy, since it does not require any knowledge on the particularities of the used constraint solver. That makes it a good choice for the default strategy for CHR answer subsumption. Better strategies may be supplied for particular constraint solvers.

For some applications one can combine answers with answer generalization which does not preserve the logical correctness. An example in regular Prolog would be to have two answers `p(a,b)` and `p(a,c)` and to replace the two of them with one answer `p(a,X)`. This guarantees (for positive programs) that no answers are lost, but it may introduce extraneous answers. A similar technique is possible with constrained answers. While this approach is logically unsound, it may be acceptable for some applications if the overall correctness of the program is not affected. An example is the use of the least upper bound operator to combine answers in the tabled abstract interpretation setting of [2].

In summary, two additional options can be supplied to extend the automatic transformation:

- `canonical_form(`*`PredName`*`)` specifies the name of the predicate that should compute the (near) canonical form of the answer constraint store. This canonical form is used to check equivalence of two constraint stores.
- `answer_combination(`*`PredName`*`)` specifies the name of the predicate that should compute the combination of two answers, if they can be combined. The value `default` selects the above mentioned default strategy.

A subsumption-based optimization technique The technique used in the `dist/3` program is to replace the computation of the exact distance of a path with the computation of an upper bound on the distance via constraints. Then, by tabling the predicate and performing answer subsumption, the defining predicate has effectively been turned into an optimizing one, computing the length of the shortest path. It is a straightforward yet powerful optimization technique that can be applied to other defining predicates as well, turning them into optimizing predicates with a minimum of changes. No meta-programming to iterate over all possible answers is required.

3.5 Evaluation of a shipment problem

Problem statement: *There are N packages available for shipping using trucks. Each package has a weight and some constraints on the time to be delivered. Each truck has a maximum load and a destination. Determine whether there is a subset of the packages that can fully load a truck destined for a certain place and all the packages in this subset are delivered on time.*

The problem is solved by the *truckload* program (see [17] for the source). Packages are represented by clauses of `pack/4`, e.g.

```
pack(3,60,chicago,T) :- leq(4,T),leq(T,29).
```

means this is the third package, it weights 60 pounds, is destined for Chicago and has to be delivered between the 4th and the 29th day. The `truckload/4` predicate computes the answer to the problem, e.g. `truckload(30,100,chicago,T)`

computes whether a subset of the packages numbered 1 to 30 exists to fill up a truck with a maximum load of 100 pounds destined for Chicago. The time constraints are captured in the bound on the constraint variable `T`. There may be multiple answers to this query, if multiple subsets that satisfy it exist.

We have run the program in four different modes:

- Firstly, the program is run as is without tabling.
- Secondly, to avoid the recomputation of subproblems in recursive calls the `truckload/4` predicate is tabled with:

```
:- table_chr truckload(_,_,_,chr) with [representation(naive)].
```

- In a third variant the answer store is canonicalized by simple sorting such that permutations are detected to be identical answers:

```
:- table_chr truckload(_,_,_,chr)
             with [representation(naive),canonical_form(sort)].
```

- Finally, in the fourth variant we apply a custom combinator to the answers: two answers with overlapping time intervals are merged into one answer with the union of the time intervals. For example the disjunction of the following two intervals on the left is equivalent to the interval on the right:

$$(1 \leq T \leq 3) \quad \vee \quad (2 \leq T \leq 4) \qquad \Longleftrightarrow \qquad (1 \leq T \leq 4)$$

  This variant is declared as, with `interval_union/3` the custom answer combinator:

```
:- table_chr truckload(_,_,_,chr)
   with [representation(naive),answer_combination(interval_union)].
```

Table 3. Results for the `truckload` program

	no tabling	tabling								
		plain			sorted			combinator		
load	runtime	runtime	space	answers	runtime	space	answers	runtime	space	answers
100	0	100	286	324	100	286	324	100	279	283
200	160	461	979	2,082	461	956	2,069	451	904	1,686
300	2,461	1,039	1,799	4,721	1,041	1,723	4,665	971	1,584	3,543
400	12,400	1,500	2,308	5,801	1.510	2,202	5,751	1,351	2,054	4,449
500	$>$ 5 min.	1,541	2,449	4,972	1,541	2,365	4,935	1,451	2,267	4,017

Table 3 contains the results of running the program in the four different modes for different maximum loads. Runtime is in milliseconds and has been obtained on the same machine as in Section 2.1. For the modes with tabling the space usage, in kilobytes, of the tables and number of unique answers have been recorded as well.

It is clear from the results that tabling does have an overhead for small loads, but that it scales much better. Both the modes with the canonical form and the answer combination have a slight space advantage over plain tabling which increases with the total number of answers. There is hardly any runtime effect for the canonical form, whereas the answer combination mode is faster with increasing load.

In summary, tabling can be useful for certain programs with CHR constraints to considerably improve scalability. Canonicalization of the answer store and answer combination can have a favorable impact on both runtime and table space depending on the particular problem.

4 Related and Future Work

In this paper we have shown that it is possible to integrate the committed choice bottom-up execution of CHRs with the tabled top-down execution of XSB. In particular the issues related to the consistency of the global CHR store and tables have been established and solutions have been formulated for call abstraction, tabling constraint stores, answer projection, answer combination (e.g. for optimization), and answer entailment checking.

Several ad hoc approaches to using constraints in XSB exist, such as a meta-interpreter [15], interfacing with a solver written in C [9] and explicit constraint store management in Prolog [16]. However, these approaches are quite cumbersome and lack the ease of use and generality of CHR.

The main related work that this paper builds on is [4], which presents a framework for constraint solvers written with attributed variables. Attributed variables are a much cruder tool for writing constraint solvers though. Implementation issues such as constraint store representation and scheduling strategies that are hidden by CHR become the users responsibility when he programs with attributed variables. Also in the tabled setting, the user has to think through all the integration issues of his attributed variables solver. For CHR we have provided generic solutions that work for all CHR constraint solvers and more powerful features can be accessed through parametrized options.

Guo and Gupta propose a technique for dynamic programming with tabling ([12]) that is somewhat similar to the one proposed here. During entailment checking they compare a particular argument in a new answer with the value in the previous answer and keep either one based on whether that argument needs to be minimized or maximized. Their technique is specified for particular numeric arguments whereas ours is for constraint stores. Further investigation of our proposal is certainly necessary to establish the extent of its applicability.

In [18] we briefly discuss two applications of CHR with tabling in the field of model checking. The integration of CHR and XSB has shown to make the implementation of model checking applications with constraints a lot easier. Indeed the next step in the search for applications is to explore more expressive models than are currently viable with traditional approaches.

We still have to look at how to implement partial abstraction and the implications of variant and subsumption based tabling. Partial abstraction and subsumption are closely related. The former transforms a call into a more general call while the latter looks for answers to more general calls, but if none are available still executes the actual call.

Finally, we would like to mention that an XSB release with the presented CHR system will soon be publicly available (see `http://xsb.sf.net`).

References

1. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.
2. M. Codish, B. Demoen, and K. Sagonas. Semantic-based program analysis for logic-based languages using XSB. *International Journal of Software Tools for Technology Transfer*, 2(1):29–45, jan 1998.
3. V. S. Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual*.
4. B. Cui and D. S. Warren. A System for Tabled Constraint Logic Programming. In *Computational Logic*, pages 478–492, 2000.
5. B. Cui and D. S. Warren. Attributed Variables in XSB. In I. Dutra et al., editors, *Electronic Notes in Theoretical Computer Science*, volume 30. Elsevier, 2000.
6. B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, oct 2002.
7. B. Demoen, M. G. de la Banda, W. Harvey, K. Marriott, and P. J. Stuckey. An Overview of HAL. In *Principles and Practice of Constraint Programming*, pages 174–188, 1999.
8. B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd et al., editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *LNAI*, pages 1240–1254, 2000.
9. X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled Resolution + Constraints: A Recipe for Model Checking Real-Time Systems. In *IEEE Real Time Systems Symposium*, Orlando, Florida, November 2000.
10. T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in LNCS, pages 90–107. Springer Verlag, March 1995.
11. T. Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming*, volume 37, October 1998.
12. H.-F. Guo and G. Gupta. Simplifying Dynamic Programming via Tabling. In R. Lopes and M. Ferreira, editors, *Proceedings of CICLOPS 2003. Technical Report DCC-2003-05, DCC - FC & LIACC, University of Porto.*
13. Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. PO Box 1263, SE-164 29 Kista, Sweden, October 2003.
14. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM Press, 1987.
15. M. Mukund, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. Verma. Symbolic Bisimulation using Tabled Constraint Logic Programming. In *International Workshop on Tabulation in Parsing and Deduction*, Vigo, Spain, September 2000.
16. G. Pemmasani, C. R. Ramakrishnan, and I. V. Ramakrishnan. Efficient Model Checking of Real Time Systems Using Tabled Logic Programming and Constraints. In *International Conference on Logic Programming*, LNCS, Copenhagen, Denmark, July 2002. Springer.
17. T. Schrijvers. CHR benchmarks and programs, January 2004. Available at http://www.cs.kuleuven.ac.be/~toms/Research/chr.html.
18. T. Schrijvers, D. S. Warren, and B. Demoen. CHR for XSB. In R. Lopes and M. Ferreira, editors, *Proceedings of CICLOPS 2003. Technical Report DCC-2003-05, DCC - FC & LIACC, University of Porto*, pages 7–20, December 2003.
19. D. S. Warren et al. The XSB Programmer's Manual: version 2.5, vols. 1 and 2, 2001.