CHR for XSB

Tom Schrijvers David S. Warren Bart Demoen
Dept. of Computer Science, K.U.Leuven, Belgium
Dept. of Computer Science, State University of New York at Stony Brook, USA
{toms,bmd}@cs.kuleuven.ac.be warren@xsb.com

Abstract

XSB is a highly declarative programming system consisting of Prolog extended with tabled resolution. It is useful for many tasks, some of which require constraint solving. Thus flexible and high level support for constraint systems is required. Constraint Handling Rules is exactly such a high level language embedded in Prolog for writing application tailored constraint solvers.

In this paper we present the integration of a CHR system in the XSB system and especially our findings on how to integrate CHR with tabled resolution, such as how to deal with issues as call abstraction of constraints, constraint store merging, answer store projection and constraint store representations for tabling.

We illustrate the power of the XSB-CHR combination with two examples in the field of model checking. It is indeed possible to quickly write application specific constraint solvers, experiment with them and achieve a reasonable performance and high readability. The combination of XSB's goal-driven fixpoint execution model with CHR's committed choice bottom-up approach has proven not only feasible, but considerably useful as well.

1 Introduction

XSB (see [14]) is a Prolog system 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 and diagnosis and many more applications benefit from tabled resolution. We refer the reader to [1] for a coverage of XSB's SLG execution strategy.

The use of constraint solvers in XSB has been a quite laborious and inconvenient endeavor up to now. Initially XSB provided no builtin support at all for dealing with constraints. Hence XSB programmers resorted to interfacing with foreign language libraries or implementation of constraint solvers in XSB itself with close coupling of constraint solver and application as a consequence. The initial feasibility study of a real time model checking system used a meta interpreter written in XSB to deal with constraints (see [11]). The full system implementation then reports of interfacing XSB with the POLINE polyhedra based constraint solver library and passing around handles to the constraint store in the XSB program (see [7]). A later version of this real time model checking application switched to using distance bound matrices implemented in XSB itself (see [12]). This shows that there is certainly a demand for constraints in the XSB setting, but that a satisfactory solution with sufficient ease of use and a reasonable implementation has not been found so far.

In an attempt to amend some of the constraint problems in XSB, it has been extended with attributed variables (see [2]). Attributed variables is a Prolog language feature that is particularly suited for constraint solver implementation as it allows efficient association of data with variables and user hooks on variable binding. Unfortunately this feature has not caught on in XSB as a basis

for constraint systems because it is a particularly low level feature that still requires considerable scheduling considerations by the constraint solver programmer. However, the work on attributed variables in XSB is not lost, as attributed variables are indeed a powerful implementation tool for constraint systems: efficient compilation of CHR to Prolog relies heavily on it (see [10]).

It is precisely these key features of CHR that are missing in XSB: Constraint Handling Rules, or CHR for short, is a high level language designed for writing application-oriented constraint systems (see [8]). We refer the reader to [9] for a survey of syntax, semantics, theoretical and practical work on CHR. In this paper we will introduce CHR with a quick informal review of syntax and semantics using a simple example (see Section 1.1).

Section 2 will present a general overview of the hProlog CHR system as well as some of its more interesting implementation details. Section 3 will then address the core matter of this paper, the integration of the CHR system in tabled execution. Subsequently, Section 4 briefly illustrates the power of the resulting CHR-XSB system with two model checking applications, one of which is the earlier mentioned real time model checking application. Finally, Section 5 concludes and suggests possible future work.

1.1 CHR by Example

The set of constraint handling rules below defines a less-than-or-equal constraint (leq/2) over numbers. The rules illustrate several syntactical features of CHR.

$$\begin{array}{l} \text{X leq X} \iff \text{true.} \\ \text{X leq Y} \iff \text{number(X), number(Y)} \mid \text{X} =< \text{Y.} \\ \text{X leq Y, Y leq X} \iff \text{X} = \text{Y.} \\ \text{X leq Y} \setminus \text{X leq Y} \iff \text{true.} \\ \text{X leq Y, Y leq Z} \implies \text{X leq Z.} \end{array}$$

The first, second and third rule are simplification rules, indicated by the double arrow. To the left of the arrow is the head of a rule. A simplification rule has the meaning that the constraints in the head can be simplified to the Prolog goal in the body, `true` for the first rule. Variables in constraints are never bound to each other or to terms in the head of a rule; only equality tests are used. The meaning of this first rule should be obvious: the leq relation is reflexive, and hence `X leq X` is trivially satisfied and bears no information.

The second rule shows that a rule can be extended by a guard, after the arrow and before the vertical bar. In this case the guard is `number(X), number(Y)`. The body of the rule is only executed for constraints that match the head and satisfy the guard. The guard can be any Prolog goal that does not bind variables of the head. Rule two replaces the constraint with a simple Prolog inequality check if the arguments are bound to numbers.

The third rule illustrates that the head of a rule can contain a conjunction of multiple constraints. It formulates the antisymmetry property of the leq constraint.

The fifth rule with the $\implies$ is a propagation rule. The body of the rule is executed once for every matching combination of constraints in the head, not removing the head constraints.

The fourth rule is a "simpagation" rule. It has the same meaning as a simplification rule where the constraints before the backslash would be posed again in the body. However it is more efficient in that it never removes those head constraints and does not unnecessarily trigger rules in that way. In the leq constraint definition its role is to declare the set semantics of the constraint, i.e. the number of copies of a constraint is not important and hence it is more efficient to keep only one.

Operationally, when a constraint is posed the rules are tried in order. For a multi-headed rule, the additional constraints are looked for in the constraint store. The outcome is that the posed constraint is either simplified away or reaches the end of the rules and gets suspended in the constraint store. Suspended it can either be used as an additional constraint for a multi-headed rule or wait until it gets triggered. Triggering of a suspended constraint occurs when any variable in the constraint gets bound. The constraint then tries to match all rules in order again.

2 The hProlog CHR System

Initially the CHR system described in this paper was written for the hProlog system. hProlog is based on dProlog (see [6]) and intended as an alternative backend to HAL (see [5]) 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 [4]).

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 and Yap. The precompiler is intended as a basis for experimentation with optimized compilation of CHR rules, both through 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.

The main advantage of the hProlog implementation is that the dynamic nature of the attributed variables in hProlog allows to move more functionality from the compiled rules to the runtime system.

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 implementation of attributed variables: it did not support attributes in different modules. Moreover, the actual binding of attributed variables was being delayed to the interrupt handler where it was left up to the programmer. 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 them more closely to the hProlog behavior. This of course facilitated the porting of the CHR system considerably.

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 [2], where a general framework for constraint solvers written with attributed variables for XSB is described. 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 Constraint Handling Rules should provide a more convenient

level of programming constraint solvers, hiding execution details whenever possible, than TCLP with attributed variables.

In [2] 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 how to implement these operations with respect to his solver implementation.

In the following we formulate these operations in terms of the CHR implementation and provide a higher level CHR interface for answer projection. In this manner the solver programmer is not confronted with the underlying CHR implementational intricacies.

3.1 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 ordinary non-constraint 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 amount of 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 list of its possible values. If the variable can be bound to at most n values it can take as much as 2^n different `domain/2` constraints, one for each subset of values.

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 reason:

- CHR rules do not require constraints to be on variables. This means that constraints 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.

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.

For CHR full abstraction requires the execution of the tabled predicate with a fresh 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) :-
        get_global_store(S_E),
        set_empty_store,
        tabled_p(X1,S_A),
        merge_stores(S_E,S_A,S_E1),
        set_global_store(S_E1),
        X1 = X.

:- table tabled_p/2.

tabled_p(X,S_A) :- ...
```

The further implementation of `tabled_p` and `merge_stores` will be discussed in the next Sections.

3.2 Tabled Store Representation and Merging

When a tabled predicate `p` returns, the answer constraint store s_a should be stored in the answer table. When a subsequent call to `p` is made, s_a should be fetched from the table and merged with the calling environment constraint store s_e. On a high level this means that all the constraints in s_a have to be inserted in s_e and triggered in such a fashion that the merged store reaches a consistent final state, i.e. with all the applicable simplifications and propagations.

The implementation of the tabled predicate `tabled_p` mentioned above is revealed here. It has an extra argument, the tabled store representation `S_A`, that is extracted from the global store after the original body of `p`, now moved to the predicate `orig_p`, has been executed.

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

orig_p(X) :- ... /* body of original p */.
```

The representation of the answer store in the table and the highly correlated merging algorithm determine largely the cost of a call to a tabled predicate with constraints. Two different implementations have been explored and it appears the more naive approach is the best. An indication of what programs or predicates are good candidates for tabling with respect to time efficiency, is obtained from this conclusion. Obviously if termination is an issue, tabling is paramount.

Suspension Representation This representation aims at storing the suspended constraints in the answer table in much the same way as they are represented in the constraint store. The constraint store is an updatable term, containing suspended constraints grouped by functor. Each suspended constraint is represented as a suspension term, containing among others 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

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

It is possible to store attributed variables in answer tables (see [3]), but two other issues do pose a problem. Firstly, the tables do not deal with cyclic terms[1]. This can be dealt with by breaking the cycles before storage and resetting them 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.

In addition to the above operations upon retrieval, the suspensions are inserted into the call environment store and then triggered.

By keeping the tabled representation as close as possible to the global store representation we hope to save on execution time during merging: the propagation history of the answer store is retained and hence answer constraints will not propagate a second time.

It turns out that in the programs we have tested so far, preparing constraints for storage in the tables and proper initialization upon retrieval is considerably more costly than repropagation. One has to bear in mind that retrieval of a table consists of creating new terms and attributed variables. Hence preserving a useful structure in the table is hardly better than preserving the data in any other format that allows easy derivation of that structure.

Naive Representation The naive representation aims at keeping the information in the table in as simple a form as possible: for each 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, it is necessary that the CHR rules behave according to set semantics, i.e. the presence of multiple identical constraints should not lead to different answers modulo identical constraints.

In all the applications we have encountered, this approach turns out to be better. The simplicity of storage and retrieval are more important than unnecessary propagation overhead.

All in all the conclusion seems to be that superior efficiency through tabling can only be achieved if a certain amount of simplification or non-constraint related computation occurs inside the tabled

[1] If the cycle point were represented as an attributed variable, then XSB tabling would handle the cyclic terms. However, this representation was deemed inappropriate due to its complexity and expected performance.

predicate and if the cost of creating the tabled constraints is smaller than executing the predicate. Hence just as it makes no sense to table `append/3`, it makes no sense to table the constraint equivalent of `append/3`, a predicate that builds a large constraint store straightforwardly.

3.3 Answer Combination and Entailment Checking

In some cases it is undesirable to have multiple answers for a tabled predicate. While all the answers are valid, they may all be just approximations. In such a case one would like to combine all answers to a single most specific answer.

Using the XSB local strategy for table completion, 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, combining both answers. If this combined answer s is different from s_0, then s_0 is discarded.

The computation of the shortest path serves as a good illustration:

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

Suppose appropriate rules for the leq/2 constraint in the above program, as in Section 1.1. The query `path(x,y,D)` will then find an answer for every single path from `x` to `y`. The answers will only differ in the upper bound on `D`.

If we are only interested in the most specific answer, we can make sure to include the following CHR rule:

$$\text{X leq D1} \setminus \text{X leq D2} \iff \text{D1} =< \text{D2} \mid \text{true}.$$

The same mechanism can be used to check entailment: if the combined answer store s is equal to one of the two, then that answer entails the other.

$$s_0 + s_1 = s_i \wedge i \in \{0, 1\} \Longrightarrow s_i \vdash s_{1-i}$$

Here the symbol + is used to indicate merging of constraint stores and = means equality of constraint stores. Constraint store equality is discussed later, in Section 3.5

3.4 Answer Projection

Often it is necessary 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 a predicate with a finite number of constraints 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 rules and implement projection as a `project/1` constraint with the list of variables to project on as an argument. A series of simpagation rules can then be used to look at and decide on what constraints to remove. A final simplification rule at the end can be used to remove the `project/1` constraint from the store.

The predicate `tabled_p` would then look like:

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

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

project(VarSet) \ X leq Y $\Longleftrightarrow$ \+ (member(X,VarSet),member(Y,VarSet)) | true.
project(VarSet) $\Longleftrightarrow$ true.

Besides removal of constraints more sophisticated operations such as weakening are possible. E.g. consider an constraint $\in /2$ for a set constraint solver that constraints an element to be in a list and a *nonempty*/1 constraint that indicates a set should not be empty:

project(VarSet) \ Elem $\in$ Set $\Longleftrightarrow$ member(Set,VarSet), \+ member(Elem,VarSet) | nonempty(Set).

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.

3.5 Constraint Store Equality

The need to check constraint store equality arises at three different locations:

- Partial call abstraction means a subset of the call environment store is passed in. The tabling system then needs to check whether a previous call with the same passed in constraint store appears in a table.
- Entailment checking means we need to check whether a merged store equals one of the initial stores.
- New answer checking means that a new answer store is compared with previous answer stores. This operation performed by the tabling mechanism is needed to avoid multiple copies of the same answer.

We can consider this equality checking with the naive representation of constraints presented previously in mind. Any permutation of this list represents the same constraint store.

When exact variable identity matters, i.e. in the case of entailment checking, the representation can be easily brought in a canonical form based on an arbitrarily chosen ordering of the involved variables, e.g. by sorting of the constraints.

For comparison with call or answer patterns in the tables, exact variable identity is not required. Equality checking needs to be done modulo variable renaming.

Elaboration on heuristics for an algorithm 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.

4 Applications

XSB Prolog, with its tabling, has proven very convenient for model checker implementation. The XMC toolset (see [13]), written on top of XSB, is a witness to that. An important feature previously missing from XSB, a simple way to write application tailored constraint solvers, has now been added with CHR.

In this section we will look at two model checking applications that both use tabling and constraint solving but in different ways. Both systems previously turned to ad hoc implementations of constraint solvers to satisfy their constraint solving needs. It took very little effort to replace these ad hoc solvers with more flexible and higher level CHR solvers.

We will focus on the constraint-related problems only and refer the reader to specialized publications regarding model checking, if more insight into the bigger scope of the applications is desired. However the common approach of the two following model checking applications is based on reachability between states in an automaton or nodes in a graph.

The standard reachability definition in Prolog gives rise to infinite loops for cyclic graphs.

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- edge(X,Z), reach(Z,Y).
```

Fortunately in XSB these infinite loops are avoided by tabling the `reach/2` predicate. Hence this `reach/2` predicate is the main intersection point for tabling and CHR in our two applications.

4.1 Model Checking of Data-Independent Systems

Data-independent systems manipulate data variables over unbounded domains but have a finite number of control locations. Such systems can be modeled as extended finite automata, finite automata with guards on the transitions and variable mapping relations between source and destination locations.

The approach of [15] represents these systems as constraint logic programs: variables are passed along states collecting more and more constraints. Hence the `reach/2` predicate will have to be described as discussed in Section 3 to deal with CHR constraints on the relevant variables.

Of this constraint approach to checking data-independent systems we studied one model that checks for a particular vulnerability in the comsat program. Comsat is a Unix server that notifies users of new mail by printing the first 7 lines to the user's terminal. Earlier versions of this program had a vulnerability that would allow a malicious person to obtain root access on the machine comsat is running on.

The exact property we are looking for here is whether a user can write arbitrary data to the `/etc/passwd` password file. If that is the case then that user can easily set a new password that is known to him for the root user. Two conditions exist under which this is possible:

- The user has write permission on `/etc/passwd`. This is a trivial solution.
- the user has write permission on `/etc/utmp`[2] This file stores user login information, including the user's terminal. By setting the root's terminal to `/etc/passwd` and sending a mail to root, any user can manage to set a new root password.

A simple model of the system that only allows to find the first solution was available to the authors. Nevertheless it contains the typical features of an extended finite automaton, while not

[2]The actual location of this file may vary between different Unix systems.

overly drawing attention to the complexity of the problem at hand, to serve as a good proof of concept for CHR in a tabled context.

Constraints perform three tasks in this application:

- They record the conditions under which state transitions in the model can be taken, and as such specify the conditions under which the system is vulnerable to attack by a user.
- They allow to rule out the uninteresting case that the user is root.
- They enable avoidance of impossible transitions in the model through failure because of unsatisfiable constraints.

The previous implementation of the constraints used an explicit list-based constraint store. From time to time consistency checks, simplification and projection were performed on this store.

It was easily replaced by 2 constraints, `neq/2` for user inequality and `exists/2` for the existence of a file in a file system.

```
neq(User,Name) \ neq(User,Name) <=> true.
exists(File,FileSystem) \ exists(File,FileSystem) <=> true.
exists(file(Name,Permissions,Data),FS) <=> lookup(Name,FS,P,D) | Permissions = P, Data = D.

project(User,FS) \ neq(AUser,FS) <=> User \ == AUser | true.
project(User,FS) \ exists(File,AFS) <=> AFS \ == FS | true.

lookup(Name,FS,Permissions,Data) :- member(file(Name,Permissions,Data),FS).
```

The `reach/2` predicate was transformed to do full constraint abstraction and the above described projection onto variables of interest.

With this CHR implementation we have much more confidence in the scheduling of projection, simplification and satisfiability checking. The implementation is much more compact and readable as well. The program runs in less than a millisecond.

4.2 Model Checking of Real Time Systems

The problem looked at here is model checking for timed automata. See [16] for an overview of different techniques.

Timed automata are automata with a finite set of clocks that can take continuous values between 0 and ∞. All clocks advance synchronously. Transitions between states can have upper and lower bound constraints on clock values. In addition, clocks can be reset on a transition. In a state clocks can delay for any amount of time before taking a next transition.

The constraint solving subproblem consists of determining the upper and lower bounds on all clocks for a certain transition given:

- the lower bound of the clocks at the start state of the transition
- the constraints on the clocks for the transition
- the maximum pairwise distances between clocks determined by resets

The previous implementation in the XMC toolkit (see [12]) used distance bound matrices (DBMs) to represent constraint stores. The necessary matrix manipulations were provided to pose constraints, determine the canonical form, reset clocks and delay.

We have replaced the DBM implementation with CHR constraints:

- `lowerbound(N,O,C)` with `N` a number, `O` either $\leq$ or $<$ and C a clock. This expresses the constraint that $n \leq c$ or $n < c$.
- `upperbound(N,O,C)` with `N` a number, `O` either $\geq$ or $>$ and C a clock. This expresses the constraint that $c \leq n$ or $c < n$.
- `diff(N,O,C1,C2)` with `N` a number, `O` either $\geq$ or $>$ and both C1 and C2 clocks. This is used to model the distance constraints. Its meaning is that $n \geq c_1 - c_2$ or $n > c_1 - c_2$.

Clock resets and delays have been implemented as operations on a list representation of the constraint store. For the actual constraint solving constraints on the clocks 24 CHR rules have been implemented, of which the following 3 are a sample :

upperbound(X,$\leq$,N1) \ upperbound(X,$\leq$,N2) $\Longleftrightarrow$ N1 $\leq$ N2 | true.

upperbound(X,$\leq$,N1), lowerbound(X,$\geq$,N2) $\Longrightarrow$ N1 $\geq$ N2.

dist(X,Y,$\leq$,D), upperbound(Y,$\leq$,N) $\Longrightarrow$ M is N + D, upperbound(X,$\leq$,M).

In this application we used tabling on two levels: `reach/2` as well as `edge/2` transitions.

In contrast to the previous application and the general approach discussed in Section 3 no constraint call abstraction is performed. Instead the constraint store is passed around explicitly when not doing any constraint solving. As no auxiliary variables are introduced, no projection is applied either. To take care of clocks delaying in states before taking a transition, upper bound constraints on them are removed in the explicit representation of the constraint store. For the solving of the constraint on an edge transition the explicit constraint store is converted to the usual global CHR constraint store and afterwards the other way around.

Instead of performing constraint abstraction in the tabled predicates, clock name abstraction is performed: i.e. clock names are replaced by variables. This allows answer reuse for identical call constraint stores modulo clock names. This optimization is quite useful in the case of parallelization of multiple identical automata. The clock names will be different for different instances of the automaton, but states and constraints are otherwise identical. The original DBM implementation is not easily capable of this optimization as rows and clocks implicitly correspond with clock names and reordering of rows and columns on other criteria would be non-trivial compared to the near canonicalization performed on the CHR list representation.

The preliminary results of this implementation are that performance equal to that of the DBM implementation has been achieved. The code however is much more concise and more confidence in the correctness of the implementation has been achieved as it is much closer to the semantics of the problem domain than the mapping onto DBM matrix manipulations. Moreover, future work on this application will explore high level optimizations enabled by the CHR implementation and semantical extensions of tabled automata such as different kinds of constraints on clocks.

5 Conclusion 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.

Furthermore CHR with tabling has proven a powerful combination: both CHR and tabled XSB relieve the programmer of the complicated underlying scheduling mechanism behind the scenes and put the focus on higher level semantics.

Model checking provides a rich application field. The combination of CHR and XSB extends the conciseness of ordinary model checking systems to those with constraints. Indeed the next step in the search for applications is to explore more expressive models than are currently viable with traditional approaches: the flexible nature of CHR makes it easy to experiment with various types of custom constraints.

Competitive performance has been observed in the model checking domain. The high level nature of the CHR implementation has revealed some optimizations not apparent or feasible to other implementations. These and more general performance aspects will be explored in future work.

As mentioned earlier in Section 3.5, pattern equality testing of constraint stores remains a challenge. The efficiency and accuracy of the used algorithm may have a considerable impact on the overall runtime of particular applications.

Other aspects of tabling constraints that have not been touched in this paper are how to implement partial abstraction and the implications for 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.

It is also worth mentioning that an XSB release with the presented CHR system will soon be publicly available (see `http://xsb.sf.net`).

Acknowledgements

We would like to thank Giridhar Pemmasani and Beata Sarna-Starosta for allowing and assisting us to rework their model checking systems with CHR constraint solvers.

Tom Schrijvers is a research assistant of the Fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

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] B. Cui and D. S. Warren. A System for Tabled Constraint Logic Programming. In *Computational Logic*, pages 478–492, 2000.

[3] B. Cui and D. S. Warren. Attributed Variables in XSB. In I. Dutra, V. S. Costa, G. Gupta, E. Pontelli, M. Carro, and P. Kacsuk, editors, *Electronic Notes in Theoretical Computer Science*, volume 30. Elsevier, 2000.

[4] B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium, oct 2002. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html.

[5] 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.

[6] B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.

[7] 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 (RTSS)*, Orlando, Florida, November 2000.

[8] T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in Lecture Notes in Computer Science, pages 90–107. Springer Verlag, March 1995.

[9] 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.

[10] C. Holzbaur and T. Frühwirth. Compiling Constraint Handling Rules. In *ERCIM/COMPULOG Workshop on Constraints*, CWI, Amsterdam, 1998.

[11] 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 (TAPD)*, Vigo, Spain, September 2000.

[12] 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 (ICLP)*, Lecture Notes in Computer Science, Copenhagen, Denmark, July 2002. Springer.

[13] C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. N. Venkatakrishnan. XMC: A Logic-Programming-Based Verification Toolset. In *Twelfth International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 576–580, Chicago, Illinois, July 2000. Springer.

[14] K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, S. Dawson, and M. Kifer. The XSB Programmer's Manual: version 2.5, vols. 1 and 2, 2001.

[15] B. Sarna-Starosta and C. Ramakrishnan. Constraint-Based Model Checking of Data-Independent Systems. In *5th International Conference on Formal Engineering Methods*, 2003.

[16] S. Yovine. Model Checking Timed Automata. In *European Educational Forum: School on Embedded Systems*, pages 114–152, 1996.