A General Implementation Framework for Tabled CLP*

Pablo Chico de Guzmán[1], Manuel Carro[1,2],
Manuel V. Hermenegildo[1,2], and Peter Stuckey[3,4]
{pablo.chico,manuel.carro,manuel.hermenegildo}@imdea.org
{mcarro,herme}@fi.upm.es pstuckey@unimelb.edu.au

[1] IMDEA Software Institute, Spain
[2] School of Computer Science, Univ. Politécnica de Madrid, Spain
[3] NICTA Victoria Research Laboratory, Australia
[4] Department of Computing and Information Systems, U. of Melbourne, Australia

Abstract. This paper describes a framework to combine tabling evaluation and constraint logic programming (TCLP). While this combination has been studied previously from a theoretical point of view and some implementations exist, they either suffer from a lack of efficiency, flexibility, or generality, or have inherent limitations with respect to the programs they can execute to completion (either with success or failure). Our framework addresses these issues directly, including the ability to check for answer / call entailment, which allows it to terminate in more cases than other approaches. The proposed framework is experimentally compared with existing solutions in order to provide evidence of the mentioned advantages.
Keywords: Constraint Logic Programming, Tabling, Implementation, Performance.

1 Introduction

Tabling [1, 2] is an execution strategy for logic programs that records calls and their answers in order to reuse them in future calls. Tabling overcomes several limitations of the SLD resolution strategy: it can avoid some infinite failures and improve efficiency in programs which repeat computations. It can also be extended to evaluate programs with stratified negation [3] and has been successfully applied in many diverse contexts which include deductive databases, program analysis, semantic Web reasoning, and model checking.

Constraint Logic Programming (CLP) [4] is a natural extension of Logic Programming (LP) which has attracted much attention. CLP languages apply efficient, incremental constraint solving techniques which blend seamlessly with the characteristics of logical variables and which increase the expressive power and declarativeness of LP.

The interest in combining tabling and CLP stems from the fact that, similarly to the LP case, CLP systems can also benefit from the power of tabling: it can enhance their declarativeness and expressiveness and in many cases also their efficiency. Some of the areas which can benefit from TCLP are:

* Work partially funded by EU projects IST-215483 *S-Cube* and FET IST-231620 *HATS*, MICINN projects TIN-2008-05624 *DOVES*, and CAM project S2009TIC-1465 *PROMETIDOS*. Pablo Chico was also funded by a MICINN FPU scholarship.

Constraint Databases [5] are databases where assignments to atomic values are generalized to constraints applied to variables. This allows for more compact representations and increases expressiveness. Database evaluation in principle proceeds bottom-up, which ensures termination in this context. However, in order to speed up query processing and spend fewer resources, top-down evaluation is also applied, where tabling can be used to avoid loops. In this setting, TCLP is necessary to capture the semantics of the constraint database [6].

Timed automata [7, 8] are used to represent and verify the behavior of real-time systems. Checking reachability (to verify safety properties) requires accumulating and solving constraints (with CLP) and testing for loops (with tabling). TCLP needs constraint projection and entailment to optimize loop detection and, in some cases, to actually detect infinite loops and non-reachable states.

Abstract interpretation requires a fixpoint procedure, often implemented using memo tables and dependency tracking [9] which are very similar to a tabling procedure [10]: repeated calls have to be checked and accumulated information is reused. Some sophisticated abstract domains, such as the *Octagon Domain* [11], have a direct representation as numerical constraints. Therefore, and in principle, the implementation of abstract interpreters can take advantage of TCLP.

The theoretical basis [6, 12] (and an initial experimental evaluation) of TCLP were laid out in the framework of bottom-up evaluation of Datalog systems, where soundness, completeness, and termination properties were established. While that work does not cover the full case of logic programming (due to, e.g., the restrictions on non-interpreted functions), it does show that the constraint domain needs to offer projection and entailment checking operations in order to ensure completeness w.r.t. the declarative semantics. However, existing TCLP frameworks and implementations lack a complete treatment of constraint projection and / or entailment. The novelty of our proposal is that *we present a complete implementation framework for TCLP, independent from the constraint solver, which can use either precise or approximate projection and entailment, possibly with optimizations*.

We have validated the flexibility, generality, and efficiency of our framework by implementing two examples: difference constraints [13] and disequality constraints. We have also evaluated the performance of our framework w.r.t. existing similar implementations and w.r.t. tools to check timed automata properties.

2 Tabling Background

We assume some familiarity with tabling and CLP. For a more complete introduction to these topics, we refer the reader to [3, 4] and their references.

2.1 Tabled Evaluation

Figure 1 presents a program and a query which does not terminate under SLD evaluation, as the call to $t/1$ in the clause body is a variant of the initial query. However, the answer $X = a$ is clearly in the model of the program. Tabled evaluation would suspend the execution when such a variant is found

```
t(b):− t(Y).
t(a).
?− t(X).
```

Fig. 1. Looping, finite model.

(hence the name of *variant tabling*), switch to the second clause, generate a solution for the initial call (X = a), and use this solution to resume the call to t(Y) and succeed with X = b. The first call to t(X) is called the *generator* and subsequent variant calls to t(X) are termed the *consumers*.

The program and query in Figure 2 would also loop under SLD. Under tabled execution, the first answer X = a to the query t(X) (which comes from the second clause) would "feed" the first clause to produce the answer X = f(a). This answer feeds the first clause again to produce X = f(f(a)), and so on. The model in this case is infinite, but a tabling strategy able to return answers one by one (e.g., batched or swapping [14]) would eventually generate any given answer.

```
t(f(Y)):− t(Y).
t(a).
?− t(X).
```

Fig. 2. Looping, infinite model.

2.2 Global Table

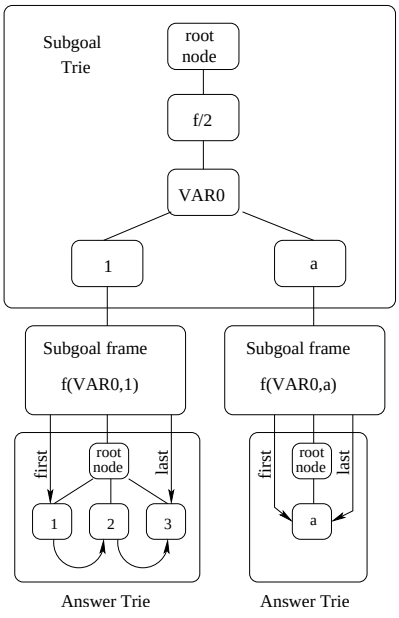

Fig. 3. Call and answers in a trie.

From the previous example, it is clear that checking for repeated call patterns and retrieving answers to previous calls is a key functionality in a tabled system. This is the role of the *global table*, a critical part of a tabling system, both in terms of efficiency and functionality.

Typical implementations for global tables use a two-level *trie* structure [15]. The first level, the *subgoal trie*, stores the call patterns; each leaf node corresponds to a different generator, represented by the generator *subgoal frame*, which stores, among other things, a pointer to the *answer trie*, where the answers obtained for every generator call are inserted. Each leaf node corresponds to a different answer to the generator call and contains the answer substitution for the original free variables of the generator. Figure 3 shows an example with two generators: f(X,1) and f(Y,a).[1] The answers for the former are f(1,1), f(2,1) and f(3,1), and the only answer for the latter is f(a,a).

2.3 Tabling Program Transformation

Our mechanism to implement tabling relies on a program transformation which uses the following primitives: lookup_call/2, is_generator/1, new_answer/1, consume_answer/1, and complete/1. We will briefly explain these operations by translating the user code in Figure 4 into tabling-ready code, shown in Figure 5.

lookup_call/2 locates the tabled call pattern corresponding to its first argument in the global table. The first occurrence of a call pattern is labeled as a *generator* and a new entry is inserted for it. Later calls are labeled as *consumers*.

[1] Trie nodes, from top to bottom, correspond to term arguments read from left to right.

```
:- table t/1.

t(X):- body(X).
```

Fig. 4. A sample code.

```
t(X):-
    lookup_call(t(X),SF),
    test_type_call(t(X),SF).
test_type_call(t(X),SF):-
    is_generator(SF),
    call(tabled_t(X,SF)).
test_type_call(_,SF):-
    consume_answer(SF).

tabled_t(X,SF):-
    body(X),
    new_answer(SF).
tabled_t(_,SF):-
    complete(SF).
```

Fig. 5. Classical tabled program transformation.

SF is unified with the subgoal frame of the current tabled call and is used by test_type_call/2 to check the type of SF via is_generator/1. If it is a generator, it resolves against program clauses to compute answers and store them in the table with new_answer/1. If it is a consumer or no more answers can be generated, it calls consume_answer/1 to read answers from the global table. consume_answer/1 suspends when no more answers are available and the corresponding generator did not complete yet (i.e., it can still generate more solutions).

complete/1 is invoked by generators after executing all their clauses. It decides if a generator can be completed (because it does not depend on previous generators), in which case any frozen memory of suspended consumers can be reclaimed. It also checks if all the consumers under the generator execution tree have consumed all their available answers. If a consumer has pending answers, its execution is resumed. The most efficient approaches are based on *stack freezing* and *trail management* [3, 16]: when a consumer suspends, its memory is protected from backtracking and its trail entries (both the trailed variable and its associated value) are stored, as these bindings will be undone on backtracking and will have to be reinstalled again before resuming a consumer.

2.4 Interaction Between Tabling and CLP

Variant tabling is not enough to ensure termination for some programs and queries. The query in Figure 6 has the answer $\mathsf{X} = \mathsf{a}$, but it loops under variant tabling: the initial query t(X) produces the call t(f(X)), which in turn produces the call t(f(f(X))), and so on. Every call is not a variant of the call which caused it. However, the second call (t(f(X))) is *subsumed* by the first one (t(X)), so any answer to the former can be obtained by further specializing some answer to the latter. Execution can suspend at this point and try the second clause, which succeeds with the single answer $\mathsf{X} = \mathsf{a}$. This is termed *subsumption tabling* [17] and is very useful in some programs.

```
t(X):− t(f(X)).
t(a).
?− t(X).
```

Fig. 6. Looping, incomplete under *variant* tabling.

In TCLP, and in order to retain similar completeness properties when using constraints, we will need to use operations of the constraint domain to detect both when a more particular call can consume answers from a more general one (*call subsumption*) and when to discard some answer because we already found a more general one (*answer subsumption*).

In the case of TCLP, subsumption is generalized to *constraint entailment*. A set of constraints C_1 is entailed by another set of constraints C_2 in the domain

D if $D \models C_2 \rightarrow C_1$. This would make it possible to determine that the program:

p(X) :- Y < X, p(Y).

finishes under the query $\{X \leq 10\}$ p(X).[2] Moreover, entailment checking can avoid redundant computations.

Another required constraint operation is *projection*. The projection of constraint C onto variables V is a constraint C' over variables V such that $D \models \exists \overline{x}.C \leftrightarrow C'$ where $\overline{x} = vars(C) - V$. The projection makes it possible to get rid of irrelevant constraints of a tabled call or a found answer. This is particularly important for programs where otherwise an infinite number of answers with ever growing answer constraint stores could be generated. Consider:

p(X) :- X1 = X + 1, X2 = X1 − 1, p(X2).

This would generate an infinite set of answers to p(X) unless the constraints are projected onto X, which will make it clear that the answers are identical.

The sometimes very high computational cost of entailment and projection has to be taken into account when deciding whether to implement them or execute under TCLP. The constraint domains we implemented and describe in Section 4 have comparatively inexpensive entailment and projection operations. An alternative to performing entailment/projection operations is *call abstraction*, where the constraint store associated to a tabled call is not taken into account to execute it. Call abstraction can unfortunately lead to arbitrarily larger computations and impact termination properties.

3 A General Framework for TCLP

We now present our TCLP implementation framework, which tries to address the following main challenges: designing a global table which is parametric on the constraint system; devising a new program transformation for TCLP programs which can take advantage of entailment; and managing consumer suspension and resumption when using general, perhaps external constraint solvers in which updates cannot be directly recorded / undone by the Prolog trail. We also point to some implementation alternatives for the primitives used by the constraint solver to communicate with our TCLP framework.

3.1 Constraint Global Table

For reasons which will become clear in short, our enhanced global table needs to distinguish between (normal) Prolog variables and variables which take part in constraints. In our implementation, the TCLP framework uses attributed variables [18] (which we will term *AVs*) for the constrained variables. *AVs* are a mechanism to customize the behavior of the unification and are used, among other things, to develop (in Prolog) constraint solvers for Prolog: the attributes can be used to keep the constraints themselves, or used to point to a constraint store managed by an external constraint solver. We will denote the normal Prolog variables as *VAR*. Figure 7 shows a snapshot of a constraint global table which will help us understand the TCLP program transformation.

[2] The constraint store at call time is shown between curly brackets.

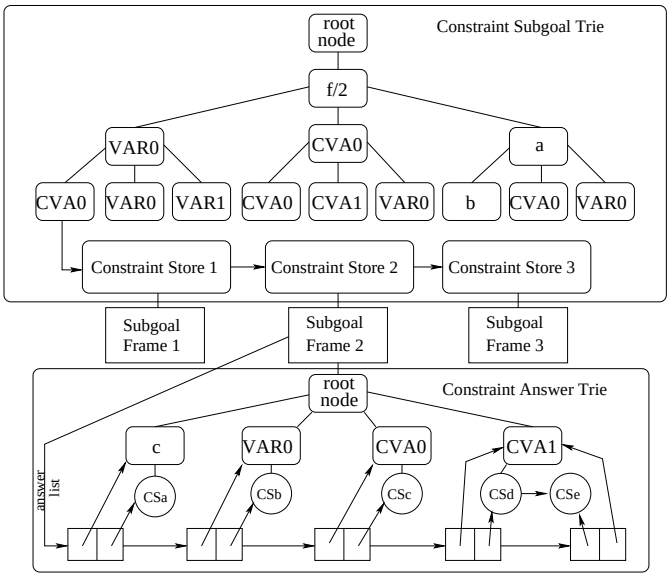

Fig. 7. Constraint Global Table

Since we still want to filter / classify calls according to their data structures (i.e., Herbrand terms), call patterns are distinguished at the level of the *constraint subgoal trie*, which is an extension of the subgoal trie to differentiate between *VARs* and *AVs*. Figure 7 shows nine different Herbrand call patterns: f(X,A), f(X,X), f(X,Y), f(A,A), f(A,B), f(A,X), f(a,b), f(a,A), and f(a,X). [3]

Every leaf node points to the *call constraint store set*, a general structure (here, a list) which can maintain separate constraint stores (representing different generators) for the same Herbrand call pattern. Call constraint store sets are managed by the constraint solver and the global table merely keeps pointers to them. Here, f(X,A) has three different constraint stores, corresponding to the different generators: $\{A < 0\}$ f(X,A), $\{3 < A < 5\}$ f(X,A) and $\{7 < A\}$ f(X,A). Each generator is associated to its corresponding subgoal frame.

The *constraint answer trie* follows a similar idea, classifying the different answer patterns of a generator at the Herbrand level. Every leaf node of the constraint answer trie points to the *answer constraint store set*, which maintains constraint stores of possibly different answers for every Herbrand answer pattern. Figure 7 shows five different answers for the generator: $\{3 < A < 5\}$ f(X,A), $\{3 < A < 4\}$ f(c,A), $\{3 < A < 4\}$ f(X,A), $\{3 < A < 4\}$ f(A,A), $\{3 < A < 4, 0 < B\}$ f(B,A), and $\{3 < A < 4, B < 0\}$ f(B,A).

3.2 TCLP Program Transformation

The program transformation for TCLP programs uses some operations provided by the tabling engine (lookup_call/2, prune_generators/1, is_generator/1, consume_answer/2, lookup_answer/2, prune_answers/1, new_answer/3 and complete/1) and some operations which must be provided by the constraint solver (solver_lookup_call/4, solver_recover_CS/1, solver_consume_answer/1

[3] A and B are *AVs*, and X and Y are *VARs*.

```
t(X):-
    lookup_call(t(X),Calls_S),
    solver_lookup_call(Calls_S,SF,CS_o,P_G),
    prune_generators(P_G),
    test_type_call(t(X),SF,CS_o).
test_type_call(t(X),SF,CS_o) :-
    is_generator(SF),
    call(tabled_t(X,SF,CS_o)).
test_type_call(t(X),_,CS_o) :-
    solver_recover_CS(CS_o)
    consume_answer(SF,A_CS),
    solver_consume_answer(A_CS).
```

```
tabled_t(X,SF,CS_o):-
    body(X),
    lookup_answer(SF,A_S),
    solver_lookup_answer(A_S,CS_o,A_CS,P_A),
    prune_answers(P_A),
    new_answer(SF,A_S,A_CS).
tabled_t(_,SF,_):- complete(SF).
```

Fig. 8. The program in Figure 7 after being transformed for constraint tabled evaluation.

and solver_lookup_answer/4. The new program transformation for the code in Figure 4 is illustrated in Figure 8, and explained below.

lookup_call/2 behaves as its LP version and returns $Calls_S$, the leaf node of the constraint subgoal trie which represents the Herbrand pattern of the current tabled call. solver_lookup_call/4 uses $Calls_S$ to look for a previous call whose associated constraint store CS is more general than the store CS_o associated to the current call G. CS_o is projected onto $Vars(G)$ to give CS_p, and entailment checking is used to search for a CS more general than CS_p. If it is found, SF is unified with the subgoal frame associated to CS. Otherwise, CS_p is added to the call constraint store set $Calls_S$ and SF is unified with a new subgoal frame associated to CS_p. CS_p can be made the active store (which in some cases can speed up the execution of the generator — see Section 3.4), in which case CS_o is returned to be used by solver_recover_CS/1 to reinstall it in order to continue the execution. P_G may be unified with a list of existing generators entailed by the new tabled call (whose constraint stores would be removed from $Calls_S$), which can be pruned and transformed into consumers of the new tabled call by prune_generators/1, similarly to *retroactive subsumption tabling* [19].

test_type_call/3 is a modified version of test_type_call/2 to include CS_o as an argument, which will be used by both solver_lookup_answer/4 (see later) and solver_recover_CS/1. consume_answer/2 is a modified version of consume_answer/1 which unifies A_{CS} with the constraint store of the answer being consumed. solver_consume_answer/1 uses A_{CS} to conjoin the current constraint store with the answer constraints.

lookup_answer/2 behaves as its LP version and returns A_S, the leaf node of the constraint answer trie which represents the Herbrand pattern of the new answer. solver_lookup_answer/4 is the answer version counterpart of solver_lookup_call/4. It uses A_S to look for a previous answer whose associated constraint store CS is more general than the current constraint store. The current constraint store is projected onto the set of variables which are visible by the rest of the execution to give A_{CS}, and entailment checking is used to search for a CS more general than A_{CS}. If it is found, solver_lookup_answer/4 fails. Otherwise, A_{CS} is added to the answer constraint store set A_S. CS_o, which is the constraint store of the current tabled call, can be used to avoid storing redundant constraints between the tabled call and the current answer. For example, if CS_o is $\{0 < X < 10\}$ and

A_{CS} is $\{0 < X < 5\}$ we could store only $\{X < 5\}$ as the new answer constraint store, since $\{0 < X\}$ is already a constraint in the generator. P_A may be unified with a list of existing answers entailed by the new one (whose constraint stores would be removed from A_S), which will be removed from the answer list of SF by prune_answers/1. If they are currently being consumed, the current execution path is canceled. Finally, new_answer/3 inserts the pair $\langle A_S, A_{CS} \rangle$ into the answer list of SF to be later returned by consume_answer/2.

3.3 Consumer Suspension/Resumption

Suspension-based tabling needs to navigate the execution tree in order to suspend and resume consumer executions. Usual suspension-based tabling uses stack freezing and trail management to this end. However, the TCLP framework hides the way the constraint solver keeps the constraint store and therefore cannot rely on updates being trailed. We work around this by having the constraint solver define a forward_trail/2 predicate whose arguments represent actions which specify how to undo constraint addition (i.e., how to backtrack) and how to reinsert previously removed constraints, needed to proceed towards consumer resumption. Every time there is a change in the constraint store, a forward_trail/2 term is pushed onto the trail stack (using the `$undo/1` predicate) so that it is invoked on backtracking. By defining forward_trail/2 to execute its second argument when called, the action to undo updates to the constraint store will be executed on backtracking. Later on, when recovering a consumer execution state, the TCLP framework will find a forward_trail/2 term in the trail and its first argument will be called to, stepwise, recover the state of the constraint store at the time of suspension.

3.4 Improvements to Constraint Domain Operations

The implementation of the constraint solver API can offer additional functionality besides what is strictly necessary to use tabled constraints.

A constraint store S is a conjunction of simple constraints. We define the result of *merging* a set of constraint stores, S_i, as a new constraint store M such that $M \Leftrightarrow \bigvee_i S_i$ where $\bigvee_i S_i$ is true for and only for valuations of variables which make at least one S_i true.

solver_lookup_answer/4 can benefit from *answer merging*, which replaces two or more answer constraint stores by the result of merging them. This can save memory in the answer constraint store set and also execution time, since consumers will consume one, more general answer, instead of two or more.

On the other hand, solver_lookup_call/4 can take advantage from *call merging*, which replaces two or more generators by a new one whose associated constraint store is the result of merging the constraint stores of the original generators. This is sound (in the absence of pruning operators), as any answer to the more general call will also be an answer to the more concrete ones. The advantage is that answers can be reused in more cases, but it can also recompute execution paths of the merged generators. For example, the generators $G_1 \equiv \{0 < X < 3\}$ p(X) and $G_2 \equiv \{X \geq 3\}$ p(X) can be merged and replaced by $G_3 \equiv \{0 < X\}$ p(X). A future consumer $\{X > 2\}$ p(X) can reuse answers from G_3 but it could not have

reused answers from either G_1 or G_2. Call merging is a special case of call abstraction which is based on the execution behavior. lookup_constraint_call/4 can also apply standard call abstraction to create a call more general than the actual one, based on some constraint domain heuristics.

As we presented earlier, the state of the constraint store of a resumed consumer can be recreated using forward_trail/2. Another alternative is *cloning*: copying the constraint store when consumers suspend to make reinstalling that state cheaper. This can be built on top of the forward_trail/2 mechanism: when suspension happens, the constraint store is copied and forward_trail/2 is installed on the trail (via $undo/1) to switch between the current and the consumer (copied) constraint store. The rest of the forward_trail/2 calls can be left in the stack with null actions, or just not be pushed. This can make it possible to find a balance between copying and trailing [20].

Finally, solver_lookup_call/4 projects the active constraint store onto the variables of the current (generator) goal in order to perform entailment checking. A new constraint store can be generated with this projection and made the active one in order to speed up the execution of the goal, at the cost of creating a new constraint store. As an alternative, solver_lookup_call/4 may avoid making the projected constraint store active, so that the goal is executed under the non-projected one. The impact on performance depends on the particular program, and deciding the best option in every case is left for future work.

4 Sample Implementations

4.1 Equality and Disequality Constraints

The constraint domain $D_{\neq}$ allows constraints of the form $X = a$, $X = Y$, $X \neq a$ and $X \neq Y$. Under the assumption that there is an infinite domain of constants, managing this constraint domain is easy. The normal Prolog mechanisms can handle the equality relationships by implementing them with unification, and therefore the solver representation only has to keep track of the disequalities. Hence a constraint store C is simply a set (which represents a conjunction) of disequalities. These constraints simply suspend until both sides of the disequality are ground and the solver then fails if they are identical.

The projection and entailment operations are simple. Projecting C onto V is simply $\{d \mid d \in C, vars(d) \subseteq V\}$, i.e., keeping only disequalities that involve variables in V. Entailment is defined by $D_{\neq} \models C_1 \rightarrow C_2$ iff $C_1 \supseteq C_2$.

Merging in this domain is also easy, since the constraints are so weak. Suppose that we have two constraint stores $C \cup \{X \neq a\}$ and $C \cup \{X \neq b\}$. Any solution θ of C where X takes some value c is clearly a solution of $(C \wedge X \neq a) \vee (C \wedge X \neq b)$, since either $c \neq a$ or $c \neq b$. Hence we can merge the two constraint stores to obtain C. As a consequence, two constraint stores over the same variables can always be merged and the call/answer constraint store set always has one constraint store.

This constraint solver is fully implemented using the attributes of the constrained variables and thus the functionality of the forward_trail/2 predicate is not needed (the standard trail takes care of constraint store changes).

4.2 Difference Constraints

Difference constraints $D_{\leq}$ are of the form $X - Y \leq d$. This is an important class of constraints that are useful for scheduling problems and temporal reasoning. They include the simple bounds constraints $X \leq d$ and $X \geq d$ as special cases by using a distinguished variable V_0 which represents the value 0: $X \leq d \Leftrightarrow X - V_0 \leq d$ and $X \geq d \Leftrightarrow V_0 - X \leq -d$.

Solving difference constraints is based on shortest path algorithms. Each constraint $X - Y \leq d$ represents an edge from X to Y of length d. The system is satisfiable if there are no negative cycles. This can be checked using the Bellman-Ford single-source shortest path algorithm. An incremental solver for these problems is also possible [13]. While satisfiability only requires a single-source shortest path algorithm, for entailment and projection we will need information on all pairs of shortest paths. Hence we make use of the Floyd-Warshall algorithm to compute them.

The solver representation is an $n \times n$ matrix of distances A where $A_{X,Y}$ is the shortest distance from X to Y. Satisfiability is checked by running the $O(n^3)$ Floyd-Warshall algorithm and checking that the $A_{X,X}$ entries are all non-negative. Incremental solving simply updates the matrix using a new edge $X - Y \leq d$ and is $O(n^2)$. This matrix is implemented in C and the attributes of the *AVs* are indexes in this matrix. forward_trail/2 is used to undo/redo changes in the matrix dimension or in any matrix cell.

Projecting the constraint store onto a set of variables V is simply extracting from the current matrix A a matrix of distances $A'_{v_1,v_2} = A_{v_1,v_2}$ for all pairs $\{v_1, v_2\} \subseteq V$. Entailment of one store A' by another A, $D_{\leq} \models A \rightarrow A'$, simply checks that $A_{v_1,v_2} \leq A'_{v_1,v_2}, \forall \{v_1, v_2\} \subseteq vars(A)$.

Store merging is also possible. We have implemented a modified version of [21]. We attempt to merge a new answer constraint store A_n with each previous answer constraint store $A_i, 0 \leq i < n$. If $A_i \bigcap A_n \neq \emptyset$, we calculate their convex hull, i.e., the matrix $A_i \biguplus A_n = \bigwedge max(A_{i_{v_j,v_k}}, A_{n_{v_j,v_k}})$. We then subtract all answer constraint stores $A_j, 0 \leq j \leq n$ from $A_i \biguplus A_n$. If the result is unsatisfiable, $A_i \biguplus A_n$ is the new merged answer constraint store and all the previous answer constraint stores which are entailed by $A_i \biguplus A_n$ can be eliminated.

5 Experimental Performance Evaluation

In this section we compare our TCLP framework with other systems. In particular, we compare with TCHR under XSB [22] and UPPAAL [23],[4] Our TCLP framework has been implemented in Ciao Prolog [26], available from `http://ciaohome.org/download_latest.html`.

All the systems were compiled with gcc 4.5.2 and executed on a machine with Ubuntu 11.04 and a 2.7GHz Intel Core i7 processor. The TCLP program transformation imposes some overhead w.r.t. the regular tabling program transformation

[4] Plain XSB [24] (which supports constraint call variant) and [25] were not included in our comparison. The current version of the former is currently not able to execute most of our benchmarks and the latter is not publicly available.

	Ciao SLD	Ciao TCLP	CHR	TCHR
sg_dq_0	—	2 312	—	49 184
sg_dq_10	—	2 408	—	50 638
sg_dq_20	—	2 008	—	51 952
sg_dq_30	—	1 441	—	52 179
sg_dq_40	—	730	—	52 366
sg_dq_50	—	104	—	52 511
path_30	—	7 140	—	129 978
path_25	—	6 680	—	129 876
path_20	—	5 964	—	128 955
path_15	—	4 336	—	129 313
path_10	—	2 396	—	128 994
path_5	—	433	—	129 616
path_0	—	1	—	129 472
truckload_100	254	78	3 041	1 511
truckload_200	12 040	2 096	105 833	26 505
truckload_300	119 815	5 900	$> 15min$	102 901
fibo_10	53	0	2 047	4
b_fibo_89	62	4	2 231	—

Table 1. Time comparison for TCLP frameworks in *ms*.

when there are no constraints in the tabled calls. We have measured this overhead to be, on average, around 10%, using the set of benchmarks in [14]. Of course, whether to use TCLP or normal tabling can be decided by the user.

5.1 Ciao TCLP versus TCHR / XSB

Table 1 shows execution times in ms. for a set of benchmarks.[5] *sg* is the *same generation* benchmark with 50 nodes, and is the only benchmark which uses the $D_{\neq}$ disequality solver. The suffix `dq_N` indicates a query such as `sg(1,X)` where `X` is constrained with `N` disequalities.

The rest of the benchmarks use the $D_{\leq}$ difference constraint solver. *path* implements right-recursive reachability in a (dense) graph with 30 nodes. The suffix indicates the maximum number of nodes in the path (which is forced through a constraint). *truckload* is taken from [22] and is a shipment problem with time constraints and parametrized by the load of the: truck the more load, the larger the search space. *fibo* is the *Fibonacci* problem using a constraint-based recursive definition. *fibo_10* calculates the 10^{th} Fibonacci number and *b_fibo_89* finds the index whose Fibonacci number is 89. Note that the two last benchmarks use the same program; reversibility stems from using constraints.

We compare standard SLD in Ciao and our TCLP implementation in Ciao, and CHR and TCHR in XSB. Our implementation is clearly more efficient than TCHR, partly due to a better, leaner constraint solver and partly to the TCHR overhead in managing tabled constraint calls.

If we ignore these overhead differences, interesting conclusions can be reached. From the examples, TCHR does not appear to benefit from additional constraints: the execution time for the `sg` and `path` benchmarks[6] is largely stable despite the increasing number of constraints imposed on the call variables. This is to be expected as TCHR uses call abstraction and removes these constraints

[5] Available from `http://clip.dia.fi.upm.es/~pchico/tabling/flops2012.tar`.

[6] All of these benchmarks need tabling because they lead to infinite derivations.

before executing the calls. Our implementation maintains all the constraints, and therefore execution time in these benchmarks decreases as the constraints get tighter because the search space is reduced. In particular, for a node count of 0 in the path benchmark, the search space is empty. Note that for 10 disequalities in the sg benchmark execution time increases. We postulate that it is because the search space does not change very much and we add the overhead of constraint management.

fibo and *truckload* do not need tabling but they benefit from memoing. Here, constraints do not prune the search space very much and TCHR / call abstraction could be viewed as a usable alternative (the differences w.r.t our framework can be mainly attributed to TCHR overheads). On the other hand, *b_fibo* under TCHR does not finish because the recursive call generates an infinite search space when using call abstraction. As a conclusion, call abstraction is acceptable for some problems but there are other cases where constraint entailment is key (to ensure termination, for example).

Although we did not present memory statistics, benchmarks where our framework explores a search space smaller than TCHR have a memory behavior improvement similar to that in execution time.

5.2 Timed Automata Applications

Reachability problems in timed automata (TA) can be expressed with difference constraints. Table 2 shows execution times in ms. for the verification of the Fisher Mutual Exclusion protocol for N processors. We compare with UPPAAL v4.0.13, a specialized, well-known, industry-standard tool widely recognized as the most efficient TA verification tool. We are using it here without extrapolation and memory reduction techniques, to make a fairer comparison.

It is clear that the last version of UPPAAL outperforms our TCLP framework, but we think that our implementation is still usable. The reasons for the difference are obvious: our TCLP implementation is a generic tool and in any case much less mature than UPPAAL, and it is still open to many optimizations, while UPPAAL is a specific tool developed over several years. On the other hand, our TCLP framework is strictly more powerful than UPPAAL. TCLP can perform backward reachability analysis and deal with Timed Modal Mu-Calculus formulas [27], while UPPAAL only performs forward reachability analysis. Also, our TCLP framework could implement a more general constraint domain – e.g. linear constraints – to solve more complex problems (although entailment/projection operations would be less efficient) and it can combine all these characteristics with standard Prolog code.

Finally, Table 3 shows the benefits of answer merging. We verify a synthetic timed automaton where the stores for answer constraints of each TA state can be merged into a more general one (e.g $\{X > 0, X \leq Y, Y < 10\}$ can be merged with $\{Y > 0, Y < X, X < 10\}$ to obtain $\{0 < X < 10,\ 0 < Y < 10\}$). The

	Ciao TCLP	UPPAAL
Fischer 2	0	0
Fischer 3	12	1
Fischer 4	270	44
Fischer 5	10 576	4 514

Table 2. Ciao TPLP vs. UPPAAL.

	size 4	size 5	size 6	size 7
Ciao	14	94	1 948	212 169
Ciao Merging	0	17	112	741

Table 3. Non-Merging vs. Merging.

merging algorithm can be expensive and therefore it may not be advisable to turn it on by default, but it can prune the search space exponentially and give an exponentially better memory usage. It will be interesting, in our view, to explore how to detect the cases where answer merging can bring an advantage.

6 Related Work

Besides the seminal work of [6, 12] there are other proposals [25, 22] notably close in spirit to this paper, but which differ in a number of relevant points.

A framework for tabled constraint solvers in XSB is presented in [25]. It builds around the ability to table calls with attributed variables, and it assumes that the constraint solver is written using attributed variables. The advantage is that, since attributed variables are trailed, the builtin forward trailing mechanism of XSB is automatically reused, instead of having to provide one tailored to the constraint solver. However, we also provide that functionality automatically for solvers written using attributed variables, and in addition we can use external solvers. We note that the latter is very interesting since attributed variables are sometimes cumbersome or underperforming. Additionally, the entailment, projection, and abstraction operations have to be provided by the constraint solver in [25], while we require operations that are more oriented towards tabling and which could be optimized further in the constraint solver. Finally, one of the drawbacks of [25] is related to efficiency: it requires a rigid management of answers (stored as lists) or answer entailment checking with respect to all the previous found answers, even if they have different answer patterns, which can lead to poor performance.

XSB also supports constraint tabled calls by default, but it imposes the restriction of using only variant call checking (including constraints), without entailment checking of calls/answers. As we have seen (Section 2.4), this is impractical in some scenarios.

A general framework for CHR under tabling evaluation is described in [22]. This approach, which brings the flexibility that CHR provides for writing constraint solvers[7] also suffers from rigid management of constraint stores associated to tabled calls / answer sets, efficiency issues rooted on the need to change the representation between CHR and Herbrand terms, and the necessity to perform a non-trivial manual program transformation. Beyond this, the main drawback of this approach is the lack of call entailment checking and the enforcement of total call abstraction. This was a deliberate design decision, but, as we have seen, it brings serious disadvantages. We believe that constrained tabled calls should be supported, and the constraint solver should be able to decide the abstraction level of call abstraction to be provided.

Finally, XMC/rt [28] and XMC/DBM [27] are two tools written in XSB to perform verification of timed automata. We did not compare with them because they are not complete constraint programming systems, but rather applications. Also, we could not find up to date versions to execute and compare with, and extrapolating from previously published performance figures proved unreliable.

[7] Note that our approach allows, in principle, to use solvers written in CHR, specially since the Ciao system includes a CHR implementation.

7 Conclusions

We have studied the viability of a new architecture for tabled constraint and implemented solvers with support for tabling evaluation for two different domains. Both solver implementations had an impact on the design of our TCLP framework and contributed to a more general management of the constraint stores. To the best of our knowledge, this is the first implementation of TCLP supporting call entailment checking, which provides reliability, flexibility, and efficiency. In particular, in our TCLP framework call/answer constraint store sets can be managed using data structures other than lists, the suspension/resumption of consumers is based on trail management, and performing entailment checking and user-defined call abstraction is possible. Although some of these characteristics demand more work from the constraint solver programmer (call/answer merging, forward_trail/2,...), the implementation of tabling already provides non-trivial mechanisms such as consumer suspension, consumer resumption, freezing of execution states, or computation pruning. Finally, our system incorporates novel ideas such as answer merging, call merging, answer merged pruning or call merged pruning. A proper use of these features can lead to large performance improvements in TCLP evaluation.

References

1. Tamaki, H., Sato, M.: OLD Resol. with Tabulation. In: ICLP, pp. 84–98. LNCS (1986)
2. Warren, D.S.: Memoing for Logic Programs. CACM **35**(3), pp. 93–111 (1992)
3. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM TOPLAS **20**(3), pp. 586–634 (May 1998)
4. Jaffar, J., Maher, M.: Constraint LP: A Survey. JLP **19/20**, pp. 503–581 (1994)
5. Kanellakis, P.C., Kuper, G.M., Revesz, P.Z.: Constraint Query Languages. J. Comput. Syst. Sci. **51**(1), pp. 26–52 (1995)
6. Toman, D.: Memoing Evaluation for Constraint Extensions of Datalog. Constraints **2**(3/4), pp. 337–359 (1997)
7. Alur, R., Dill, D.L.: A Theory of Timed Automata. TCS **126**, pp. 183–235 (1994)
8. Bengtsson, J., Yi, W.: Timed Automata: Semantics, Algorithms and Tools. In: Lectures on Concurrency and Petri Nets. pp. 87–124. (2003)
9. Hermenegildo, M., Puebla, G., Marriott, K., Stuckey, P.: Incremental Analysis of Constraint Logic Programs. ACM TOPLAS **22**(2), pp. 187–223 (March 2000)
10. Toman, D.: Constraint Databases and Program Analysis Using Abstract Interpretation. In: CDTA. Volume 1191 of LNCS. pp. 246–262. (1997)
11. Miné, A.: The Octagon Abstract Domain. HOSC **19**(1), pp. 31–100 (2006)
12. Codognet, P.: A Tabulation Method for Constraint Logic Programming. In: INAP'95, Tokyo, Japan. (Oct. 1995)
13. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Fully Dynamic Shortest Paths and Negative Cycles Detection on Digraphs with Arbitrary Arc Weights. In: ESA. pp. 320–331. (1998)
14. Chico de G., P., Carro, M., Warren, D.S.: Swapping Evaluation: A Memory-Scalable Solution for Answer-On-Demand Tabling. TPLP **10 (4–6)**, pp. 401–416 (July 2010)
15. Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T., Warren, D.: Efficient Tabling Mechanisms for Logic Programs. In: ICLP. pp. 697–711. (1995)
16. Demoen, B., Sagonas, K.: CHAT: the copy-hybrid approach to tabling. Future Generation Computer Systems **16**, pp. 809–830 (2000)

17. Rao, P., Ramakrishnan, C.R., Ramakrishnan, I.: A Thread in Time Saves Tabling Time. In: JICSLP. MIT Press (1996)
18. Holzbaur, C.: Metastructures vs. Attributed Variables in the Context of Extensible Unification. In: PLILP'92, pp. 260–268. LNCS 631 (August 1992)
19. Cruz, F., Rocha, R.: Retroactive Subsumption-Based Tabled Evaluation of Logic Programs. In: JELIA. Volume 6341 of LNCS., pp. 130–142. Springer (2010)
20. Schulte, C.: Comparing trailing and copying for constraint programming. In: International Conference on Logic Programming. pp. 275–289. (1999)
21. David, A.: Merging DBMs Efficiently. In: 17th Nordic Workshop on Programming Theory, NWPT'05, pp. 54–56. DIKU, University of Copenhagen (2005)
22. Schrijvers, T., Demoen, B., Warren, D.S.: TCHR: a Framework for Tabled CLP. TPLP **8**(4), pp. 491–526 (2008)
23. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. STTT **1**, (1997)
24. Sagonas, K., Swift, T., Warren, D.: The XSB Programming System. In: ILPS - PLD. Number TR #1183, pp. 164–164. U. of Wisconsin (October 1993)
25. Cui, B., Warren, D.S.: A System for Tabled Constraint Logic Programming. In: Computational Logic. pp. 478–492. (2000)
26. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. TPLP **12**(1–2), pp. 219–252 (2012) http://arxiv.org/abs/1102.5497.
27. Pemmasani, G., Ramakrishnan, C.R., Ramakrishnan, I.V.: Efficient Real-Time Model Checking Using Tabled Logic Programming and Constraints. In: ICLP. Volume 2401 of LNCS., pp. 100–114. Springer (2002)
28. Du, X., Ramakrishnan, C.R., Smolka, S.A.: Tabled Resolution + Constraints: A Recipe for Model Checking Real-Time Systems. In: IEEE Real-Time Systems Symposium, pp. 175–184. IEEE Computer Society (2000)