CHR Meets MapReduce

Amr Osman[1], Amira Zaki[2], and Slim Abdennadher[1]

[1] Faculty of Computer Science and Engineering, German University in Cairo, Egypt
{amr.salaheldin, slim.abdennadher}@guc.edu.eg
[2] Faculty of Engineering and Computer Sciences, Ulm University, Germany
amira.zaki@uni-ulm.de

Abstract. The robustness, power and abstract syntax of declarative constraint programming has spawned many new research directions [1]. As the industry moves towards more robust, parallel, complex distributed systems and cloud computing for on-demand computation and dynamic scaling of applications, the need to have more intuitive programming languages that provide high-level abstraction from the underlying hardware grows. MapReduce was designed to be one of such programming paradigms. However, it still suffers some drawbacks and it requires developers to have certain skills to efficiently model their problems in the MapReduce paradigm. We argue that CHR could be the next game-changing step in the future cloud computing and we propose a novel work-in-progress for parallelizing execution of CHR programs in MapReduce data-intensive clusters that process BigData.

Keywords: Constraint Handling Rules, MapReduce, Cloud Computing, BigData, Parallel, Concurrent, Stream Computing

1 Introduction

Constraint Handling Rules (CHR) is a declarative logical programming language that relies on a set of constraints to model a specific problem in the form of a program. While different operational semantics and numerous implementations of CHR exist, only a few approaches in the literature attempt to provide parallel and concurrent execution for CHR programs and none of the implementations target multi-node clusters that consume BigData as in cloud computing environments. The power of logic programming in general combined with the ease of declarative constraints could be the next key-enabler to reduce programming complexity in data-intensive clouds and facilitate computations on highly dimensional BigData as in the areas of data mining and Cloud Analytics.

Our work comprises exploiting concurrent execution of CHR programs by firing different constraints in parallel and effectively map the execution pipeline of a CHR program from a sequential way into a MapReduce distributed fashion where different threads per worker per cluster collaborate to run the constraint rules with respect to the input query in order to yield the final result. In a future full of constraint-rich programs and many queries per program streaming into the

service-oriented and data-intensive cloud, traditional sequential processing can no longer scale and maintain a high performance throughput. Another stumbling challenge is avoiding having the constraint store as a single point of failure and storing the constraints in a distributed manner without affecting the consistency of the knowledge base which is updated according to the derivation rules or the behavior of the program.

The rest of the paper is structured as follows: In the next section, we state some preliminary overview about how CHR works and the concept of MapReduce. In section 3, we highlight and review some state-of-the-art approaches in the literature that attempt to provide concurrent CHR implementations on different hardware architectures and platform environments. In the fourth Section, we introduce our proposal to execute CHR concurrently on multi-node clusters and discuss how we suggest to solve various problems that arise from that. Finally, we summarize our approach, discuss its current limitations, lessons learned and our future work suggestions.

2 Background

Before diving into relevant approaches, concurrent CHR and multi-node cluster platforms, we first need to define CHR and provide an overview on the structure of a CHR program, its syntax and how it executes according to operational semantics. Moreover, it is essential to limit our scope to which specific areas of cloud computing we target and how MapReduce generally operates.

2.1 CHR

CHR [2] is a language aimed at developing constraint solvers declaratively by specifying a set of rules which are applied exhaustively to match some input goal constraints and accordingly, remove or add constraints to a multi-set called the constraint store. Unification is used to match any two given constraints with the same arity together. Rule types follow one of the following forms:

```
% rule is optionally tagged by a name before the '@' symbol.

simplification @ HeadReplaced <=> Guard | Body
propagation    @ HeadPersisted ==> Guard | Body
simpagation    @ HeadPersisted \ HeadReplaced <=> Guard | Body
```

The above is also subject to the following specifications:

- Rule heads and goals must consist of only CHR constraints. These, are just FOL predicate symbols that might or might not be unifiable.
- Guards consist of only built-in constraints which are basically a restricted set of functions.
- The Body may be a mix of both: built-in and CHR constraints.

In Simplification rules, the head constraint is replaced by the body if the guard check passes. This is contrary to propagation rules, where head constraints are kept in addition to the body which is further added to the constraint store. Simpagation rules replace constraints after the '\' operator with the body while keeping constraints before it.

It follows that any CHR rule can be generalized in the simpagation rule notation. If 'HeadPersisted' is the empty set, then it is equivalent to the simplification rule. While if 'HeadReplaced' is empty set then, it is equivalent to the propagation rule. Consequently, any argument proven on a simpagation rule is inherently valid for simplification and simpagation rules.

Various operational semantics have been formalized to govern the constraint transition states and limit the constraint store mutation with respect to the applicability of different rules and unifying goal constraints with other constraints. In brief, abstract semantics addresses the issue of non-termination by preventing running into propagation rule cycles for the same constraints. Additionally, it separates built-in constraints from CHR constraints from goal constraints. The refined semantics were later introduced to deterministically apply rules from top to bottom and unify constraints one by one from left to right. This permits programmers to have more control over the program behavior and re-arrange rules and constraints for efficiency at the expense of limiting execution to be strictly procedural.

2.2 MapReduce

Introduced by Google, MapReduce [3] is a full-stack framework and programming paradigm to abstract distributed processing on large clusters. It relies on a master node to schedule jobs and assign tasks to worker nodes where the execution happens in two phases: A map phase and a reduce phase. During the map phase, nodes that are assigned to map tasks consume chunks of data and emit intermediate results in the form of (Key, Value) pairs. These intermediate results are typically shuffled and then passed to reducer nodes which aggregate the intermediate data grouped by key into a smaller set of results. The framework ensures maximum throughput, fault-tolerance and data locality through different optimization techniques. Since we are only interested in the programming paradigm and the abstraction, we will mathematically formalize MapReduce execution.

MapReduce programming is achieved by introducing two key functions for processing the data: A *map* function and a *reduce* function which execute back to back in the pipeline. We mathematically formalize these two functions as follows: the map function m is represented as $\langle k, v\rangle \overset{m}{\longmapsto} \bigcup_{i=1}^{n} \{\langle k'_i, v'_i\rangle\}$ whereas the reduce function r which further processes the map function's output is represented as $\left\langle k'_i, \overrightarrow{V''}\right\rangle \overset{r}{\longmapsto} \overrightarrow{V'''}$ constrained by:

$$\forall_{\langle k'_i, v'_i\rangle \in m(\langle k,v\rangle)} \exists_{\langle k'_i, v'_j\rangle} \left[\{v'_i, v'_j\} \setminus V'' = \phi\right] \quad (1)$$

$$\forall_{\langle k'_i, v'_i\rangle, \langle k'_k, v'_k\rangle \in m(\langle k,v\rangle)} k'_i \neq k'_k \Rightarrow v'_k \notin V'' \quad (2)$$

$$|V'''| \leq |V''| \quad (3)$$

where k, v indicate an arbitrary input key along with its corresponding value from the input data respectively and V'', V''' are both lists of values in the form: $\{v_1, v_2, v_3, ...v_n\}$. The reduce function is preceded by a grouping function which groups the intermediate output values from $m(\langle k, v\rangle)$ by key. This can be shown in constraints (1) and (2). The reduce function then performs some arbitrary computation and reduces the input set of values V'' into a smaller set V''' as shown by constraint (3).

In terms of CHR, MapReduce can be simulated as follows:

```
mapdone @ map(_,[]) <=> true.
map     @ map(K, [D|DS]) <=> pair(K,D), map(K,DS).

merge   @ pair(K,D1), pair(K,D2) <=> list(K,[D1,D2]).
merge2  @ pair(K,D1), list(K,D2) <=> list(K,[D1|D2]).

% Assume <reduce> sums elements.
reduce  @ list(K,[L|L1]) <=> L2 is L+L1, list(K,L2).
```

Running the above code with the goal *map(1,[2,3,6,10]),map(2,[4,5,7,11])* under the refined semantics yields the answer *list(2,27),list(1,21)*. The execution flow towards this final result is visualized in Fig. 1. Note that the key value might change in between by the map function itself and that the intermediate values are typically interleaved and not ordered by any means. For simplicity, we let the goal constraint input the initial key values. The constraint `map/2` indicates that a map task with a Key K is to be executed on an input data set `[D|DS]`. The map rule then divides the data in the form of (key,value) pairs as in the constraint `pair/2`. In an intermediate step, the pairs are aggregated and grouped by key forming a list per key as in `list/2`. Finally the intermediate lists are then reduced into a smaller list as in the 'reduce' rule which sums the elements of the second argument.

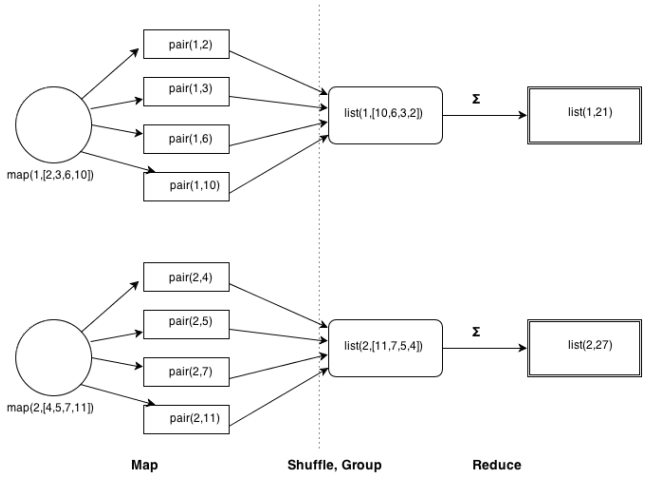

Fig. 1. MapReduce workflow

TwitterStorm [4] is a real-time, MapReduce-inspired, and java-based distributed stream computing framework. Unlike other frameworks which are focused on batch processing and suffer a pipeline stall between the map phase and the reduce phase, it is focused on consuming and propagating data in the form of immutable tuples which are utilized upon arrival. TwitterStorm adds another abstraction layer on top of MapReduce by following a DAG workflow definition. It is based on two simple graph components that are scaled and executed in parallel by all workers: A Spout and a Bolt. Spouts are simply data sources that emit tuples in real-time while Bolts are data sinks that consume input tuples from various streams and emit some new tuples into their corresponding intermediate output streams. This process is demonstrated in Fig. 2.

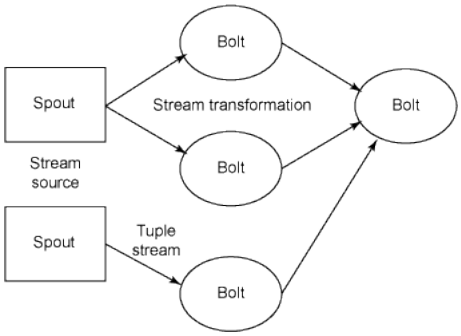

Fig. 2. TwitterStorm DAG

3 Related Work

Seeking answers to our main challenges, we hereby review the literature in hope to collect and dissect solutions to the following:

1. What are the possible means to have a shared constraint store in a distributed environment?
2. How to fully exploit concurrency and parallelism when executing CHR programs in real-time.
3. How to resolve consistency issues and maintain correctness given that rules will fire out of order and goal constraints might get picked at random.
4. Which approach is ideal and analogous to the MapReduce design pattern?

Previous work [5] introduced a variant of concurrent abstract semantics that allows for multiple parallel removals of constraints from the constraint store. Such issue is a limitation of the abstract semantics under the rules of concurrency and a shared constraint store. Their approach specifically targets a class of constraint objects that are multiplicity independent. In other words duplicating a specific constraint, has no effect on the correctness of the final result. By duplicating these objects, parallel rules can then safely remove the same constraint by removing one of replicated constraints each.

Another paper [6] proposed a variation of the refined semantics that exploits multi-core concurrency using Software Transactional Memory (STM) on top of Haskell. The implementation fully relies on transactions and atomic locks to resolve conflicts of re-writing multiple constraints where a failed transaction can be restarted at any time. It was concluded that even with the somewhat naive implementation, a great speedup was achieved. In a more recent paper [7], the same approach was used and perfected with more optimizations for goal-based parallelism [8] which is argued to scale better than rule-based parallelism for such scenario. The paper makes compelling justifications for different optimization techniques including unique constraint lookup plans to minimize thread conflicts, constraint indexing, breadth-first and depth-first goal execution and lazy goal storage. Thanks to such optimizations, the paper achieves even better speedups for different CHR applications.

General Purpose Graphics Processing Units (GPGPUs) are ideal platforms for massively parallel multi-core stream computing. An ongoing preliminary work [9] demonstrated a method to translate CHR rules and constraints into C++ structures which are then used and computed in parallel during kernel calls on a GPGPU with the shared constraint store inside the device memory. We could later adapt a similar method to port a simple-enough CHR implementation into a MapReduce cluster. Other optimization techniques as aforementioned in previous research could also be incorporated such as constraint indexing and caching utilizing the device shared memory per Symmetric Multi-Processor (SM) on CUDA architectures to make best use of data locality.

AngelicCHR [10] sheds light on a revolutionary approach to bring parallel execution using the abstract semantics. Instead of being committed-choice when

applying CHR rules, it suggests exploring all possible choices and directions according to a proposed formalization of angelic semantics. Moreover, it follows a goal-based approach and visualizes all derivation paths in the form of a derivation network. However, no actual implementation details were provided. It is trivial that under such modification, constraint stores isolation per path are necessary. i.e. no sharing. This could demand more storage. Nevertheless, derivation nets are seemingly the perfect match for MapReduce. The next Section will highlight some key reasons why.

4 Proposed Approach

For maximum concurrency, we base our work on the abstract operational semantics as formalized in [2]. The non-determinism in applying the rules as well as selecting the goal constraints allows for interleaving execution in contrast to the refined semantics which has limited monotonicity where strict rule scheduling and prioritization is enforced by firing rules top-down and matching constraints from left to right. However under a shared constraint store, triggering multiple rules might alter how a certain program behaves and falsify results due to removal of a constraint that could have been matched by some other rule. Similarly, deciding which goal constraint to execute first will have an impact on which of the rules will get fired and the propagation history per state.

Example 1. Consider the following simple CHR program and the goal constraint 'a':

```
a <=> b,c.
a ==> d.
```

Clearly if the first rule is executed first, then a will be removed from the constraint store and the result constraints 'b' & 'c' will be returned since there are no possible rules to match them. On the other hand if the second rule is executed first then, both a and d will be added to the constraint store. Since the propagation rule was marked in the propagation history of the current state, we will avoid executing it for the same constraint 'a' again. Now, the 1st rule will fire next replacing 'a' with 'b' & 'c' in the constraint store yielding 'b', 'c' and 'd' as a final result.

Example 2. Consider the CHR program below when prompted with the goal constraints: 'a, b' and assuming rules will fire top-down:

```
a <=> true.
a,b <=> x.
```

If we match 'a' first, then it will be removed from the constraint store and 'b' will be returned as an answer. However if we matched 'b' first then, it will match nothing. Consequently we will try to match 'a' whilst having 'b' in the store which will trigger the second rule adding 'x' to the constraint store and removing both: 'a' & 'b' and therefore, emitting the final result 'x'.

Note 1. In both examples under the refined operational semantics, the second rule will never fire due to the strict order of execution. It will always be the case that 'a' will be removed from the constraint store and replaced by the first rule's body.

Our main approach is thusly centered around utilizing TwitterStorm and leveraging its real-time distributed stream processing capabilities in order to apply rules on different goal constraints in parallel. This should also be done transparently without imposing any restrictions on the given CHR programs. In Section 3, we provided a high-level review of AngelicCHR. It follows that it lacks implementation and technical aspects of parallel strategies despite the concrete theoretical formalism of Angelic semantics. Precisely, this gap is where our effort is articulated. The concept of derivation nets suits the MapReduce execution strategy in general and TwitterStorm in specific due to the fact that the different derivation paths are shown and that the dependencies between different rule constraints are highlighted in the arcs of the network. Such dependencies easily model the separation between map tasks and reduce tasks respectively. Along the same line of thought, the lack of dependencies is a true indicator of the degree of parallelism of each task.

However, TwitterStorm lacks CHR implementation. One possible way is to follow the footsteps of [9] by simplifying rules in terms of data structures and Java code. Another way is to make use of JCHR [11] in a distributed fashion and ensure that different versions consisting of different constraint permutations are given to each worker to fully explore all paths and then, aggregate the results back during the reduce phase.

Before we start discussing our technical approach, it is worth mentioning that there are many ways to support a shared constraint store in a distributed cloud environment such as using distributed NoSQL, No Single Point of failure in-memory databases which ensure that computation is tightly coupled with the data storage to make best use of locality and avoid the overhead of sending data cross-nodes over the network. Also by relying on in-memory storage, we eliminate another layer of latency during accessing the hard disks. However as we argued before, AngelicCHR relies on isolated constraint stores to fully explore all computation paths. Thus, it is not necessary to have a shared constraint store.

To describe a TwitterStorm topology, we need to define two necessary implementations: A Bolt and a Spout. There could be one or more Bolt implementations depending on the different roles of each one and its position inside the DAG. In CHR, how these two components work can be simulated using the following rules:

```
% A Spout keeps generating new tuples from input data
Spout @ data(X,Y,Z,L,M,N) <=> tuple(X,Z,M),tuple(Y,L,N).

% A Bolt performs arbitary computation:
%   consumes and emits new tuples
% PS: tuples are immutable.
Bolt  @ tuple(X,Y,Z) ==> X1 is X+Y, newtuple(X1,Y,Z).
Bolt2 @ tuple(X,Y,Z) <=> Y1 is Y+Z, newtuple2(X,Y1,Z).
Bolt3 @ newtuple(X,Y,Z), newtuple2(L,M,N) <=> R is X+Y+Z+L+M+N,
                                              newtuple3(R).
```

The above minimalistic code when executed under the refined semantics simulates the TwitterStorm topology in Fig. 3. (Except for the fact that each Bolt and Spout will be executed by thousands of threads randomly distributed on working nodes in a cluster). We use the constraints `newtuple/3`, `newtuple2/3` and `newtuple3/1` to indicate that such tuples belong to different streams directed at or coming out of different Bolts such that each Bolt can be attached to one ore more streams. We can therefore confirm the following points:

- Tuples are immutable. i.e. their values cannot change but, can be only read.
- Dimensions of tuples throughout the lifecycle of the topology can vary along the stream from one component (Bolt) to another.
- Spouts generate tuple structures from input data while Bolts consume them.
- Different Bolts can consume the same tuples. Or tuple streams can be replicated and distributed according to the definition of the topology. This is why we have rule 2 as a propagation instead of simplification for example.
- Bolts can also aggregate data from different streams. This can be deduced from the last rule.

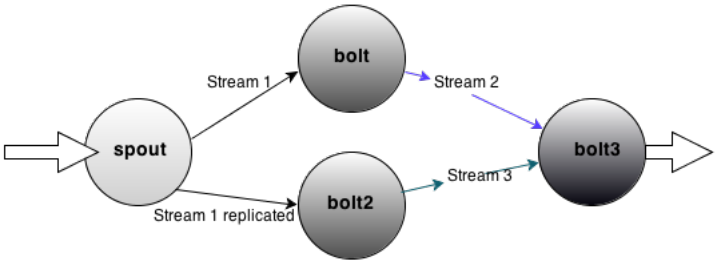

Fig. 3. A general example of a TwitterStorm topology

As discussed earlier, we decided to use JCHR on all nodes which eliminates the need to implement abstract semantics and CHR rules from scratch at first. However, we discovered that it would be much easier to call the Prolog interpreter via a system command from Java. On Linux, we can also pragmatically direct the input query via bash I/O redirection. This leaves us with the task of rules

re-ordering and parallel execution. We only decided to head this route towards a proof of concept implementation and leave the second route of implementing optimized abstract semantics from scratch for our future work. Table 1 provides a TwitterStorm glossary for some of the built-in functions and terms that will be used.

Table 1. TwitterStorm glossary

Function()/Object	**Description**
nextTuple()	Automatically called once the Spout needs to emit a new tuple
_collector	An output collector object which emits new tuples onto the out stream
emit(..)	Emits strictly one new tuple onto a specific output stream
Values	An ordered collections of values for a given tuple
Fields	An ordered collection of filed names for the corresponding values of a tuple
getIntegerByField()	Gets a specific integer value corresponding to a field name inside a tuple
getStringByField()	Gets a specific String value corresponding to a field name inside a tuple
declareOutputFields()	Called automatically to announce the field names of the values of all output tuples
execute(..)	Called automatically when a Bolt receives an input tuple

Our approach can be summarized using the following algorithm for the Spout:

```
//Assuming we have read the input program and stored the rules.
final String[] rules;
//Input Queries from the user
final String[] queries;
final Random rand = new Random();
int randIndex;
String query;

public void nextTuple() {
    randIndex = rand.nextInt(rules.length);
    query = queries[rand.nextInt(queries.length)];
    final String rule = rules[randIndex];
    _collector.emit(new Values(rules.length, randIndex, rule, query));
}

public void declareOutputFields(OutputFieldsDecelarer d) {
    d.declare(new Fields("expect", "ruleId", "rule", "query")); }
```

Each Bolt is explicitly told how many rules to expect, a rule identifier for each rule streamed, a random input query and the rule string itself. This is all packaged into the tuple being sent which contains a random rule each time. Our Bolt abstract implementation is as follows:

```
ArrayList<Integer> receivedRules = new ArrayList<Integer> ();
ArrayList<String> rules = new ArrayList<String>();

public void execute(Tuple tuple) {
        int expectedCount = tuple.getIntegerByField("expect");
        int ruleId = tuple.getIntegerByField("ruleId");
        String rule = tuple.getStringByField("rule");
        String query = tuple.getStringByField("query");
        if(rules.size() == expectedCount) {
            String fileName = writePlFile(receivedRules);
            String result = execCHR(query, fileName);
            _collector.emit(new Values(result));
        }
        else if(!receivedRules.contains(ruleId)) {
            rules.add(rule);
            receivedRules.add(ruleId);
        }
}

public void declareOutputFields(OutputFieldsDecelarer d) {
    d.declare(new Fields("result"));
}
```

The correctness and exploring all the derivation paths obviously depends on the degree of parallelism specified for the Bolt. Or, how many threads per worker will execute. These specifications are easily tuned during building the DAG topology and submitting it for execution on the cluster using some provided built-in methods. We could later introduce another Spout to filter out the duplicate results in case the random ordering of the rules was the same at some instance of time or, in case of a confluent program. But, it is not necessary.

In brief, the Spout simply streams a random rule from the input program and a random query from the input queries each time. Every Bolt worker collects and stores rules by the order of arrival and when it reaches the expected number of unique rules as in the original program, it starts executing the random query on the current program version and yields out the result. In the above code, the `writePlFile(..)` function writes the received rules in their order to a valid CHR module in the form of a prolog file. The `execCHR(..)` function will call the prolog interpreter on the written pl file and query it with the current received query using the methodology we highlighted earlier.

Despite that our approach allows multiple streaming queries on a given program in parallel and implements a subset of angelic parallel semantics (More specifically, the rule-based angelism property without deep guards or head decomposability), we still have not fully exploited goal-based parallelism and stream-

ing of constraints in a shared store environment. Similarly, allowing multiple programs with multiple queries to execute in parallel. We leave this and other future optimizations for our coming development.

5 Conclusion and Future work

CHR is an inherently-parallel declarative constraint logic programming language. Relative to other languages, its power in terms of lines of code is significantly higher and thus, it is more abstract and high level. As the cloud seeks more abstraction from the hardware and virtualization all the way to the service-level, declarative constraint programming is potentially the key. Different cloud applications could be abstracted using CHR such as service orchestration [12], multi-policy management, business modeling [13][14]. Many approaches have also been proposed for massively BigData-rich parallel computations as in data mining [15], cloud analytics and scheduling [16]. Other areas such as airplanes runway-routing inside airports, railway scheduling and more [17] remain active fields of interest.

We introduced a very early and draft work-in-progress to bring CHR into the data-intensive cloud and parallelize/scale it inside real-time cluster environments. We further modeled the MapReduce programming paradigm and the TwitterStorm [4] workflow using CHR after our mathematical formalization. Our early implementation follows the same footsteps as AngelicCHR [10], supports streaming multi-queries per program and is headed towards a parallel, shared and distributed constraint store implementation inside the cloud relying on abstract semantics. Exploiting the class of CHR programs that enjoy the online property is additionally on our schedule. For future work, we also plan to use optimization techniques discussed in [8] with the aid of distributed in-memory databases for our implementation and add multi-program support. CHR as a service is indeed the next big step in the era of the service-oriented cloud.

References

1. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint handling rules. TPLP **10**(1) (2010) 1–47
2. Frühwirth, T.: Constraint handling rules. Volume 3. Cambridge University Press Cambridge (2009)
3. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1) (2008) 107–113
4. Nathan, M., James, X., Jason, J.: Storm: Distributed real-time computation system, [online]. available: `http://storm-project.net/`. (2012)
5. Raiser, F., Frühwirth, T.: Exhaustive parallel rewriting with multiple removals. In: Proc. of 24th Workshop on (Constraint) Logic Programming. (2010)
6. Lam, E.S., Sulzmann, M.: A concurrent Constraint Handling Rules semantics and its implementation with software transactional memory. In: DAMP '07: Proc. ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming. (January 2007)

7. Sulzmann, M., Lam, E.S.: Parallel execution of multi-set constraint rewrite rules. In: Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming, ACM (2008) 20–31
8. Lam, E.S., Sulzmann, M.: Concurrent goal-based execution of constraint handling rules. Theory and Practice of Logic Programming **11**(6) (2011) 841–879
9. Zaki, A., Frühwirth, T., Geller, I.: Parallel execution of constraint handling rules on a graphical processing unit. In: Proceedings of the 9th International Workshop on Constraint Handling Rules. (2012) 82
10. Martinez, T.: Angelic chr. In: Eighth International Workshop. (2011) 19
11. Van Weert, P., Schrijvers, T., Demoen, B.: Ku leuven jchr: a user-friendly, flexible and efficient chr system for java. In: Proceedings of the 2nd Workshop on Constraint Handling Rules. (2005) 47–62
12. Salomie, I., Chifu, V., Harsa, I., Gherga, M.: Web service composition using fluent calculus. International Journal of Metadata, Semantics and Ontologies **5**(3) (2010) 238–250
13. Dausend, M., Raiser, F.: Model transformation using constraint handling rules as a basis for model interpretation. In: Proceedings of the Eighth International Workshop on Constraint Handling Rules. (2011) 64–78
14. Raiser, F., FrÜhwirth, T.: Analysing graph transformation systems through constraint handling rules. Theory Pract. Log. Program. **11**(1) (January 2011) 65–109
15. De Raedt, L., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM (2008) 204–212
16. Abdennadher, S., Marte, M.: University timetabling using constraint handling rules. In: Proc. JFPLC. Volume 98. (1998) 39–49
17. Langbein, J., Stelzer, R., Frühwirth, T.: A rule-based approach to long-term routing for autonomous sailboats. In: Robotic Sailing 2011, Part V. (2011) 195–204