K.U.Leuven JCHR: a User-Friendly, Flexible and Efficient CHR System for Java

Peter Van Weert, Tom Schrijvers*, Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium

Abstract. A new integration of Constraint Handling Rules (CHR) and Java is presented: the K.U.Leuven JCHR system. It was designed with three aims in mind: user-friendliness, flexibility and efficiency.
User-friendliness is achieved by providing a high-level syntax that feels familiar to both Java programmers and users of other CHR embeddings, and by full compliance to the refined operational semantics. Flexibility is the result of a well thought-out design, allowing e.g. an easy integration of built-in constraint solvers and variable types. Efficiency is achieved through an optimized compilation to Java and the use of a very efficient constraint store. The design of the K.U.Leuven JCHR system is presented, its implementation discussed and its performance is compared with other CHR systems embedded in Java and Prolog.

1 Introduction

Constraint Handling Rules (CHR) is a high-level, declarative programming language, originally designed for the introduction of user-defined, application-tailored constraint solvers in a given host language. Nowadays CHR is increasingly being used as a general programming language in a wide range of applications – multi-agent systems, type systems and natural language processing, to name some – and for the efficient implementation of classical algorithms.

Since its conception in 1991, CHR systems have been developed for several host languages, with the main emphasis being placed on Prolog implementations. The first full CHR system [16] was developed by Christian Holzbaur in co-operation with Thom Frühwirth. This system is included in SICStus Prolog and Yap and is considered the reference implementation. In the past five years CHR systems have been written for HAL and Haskell. The most recent and advanced CHR system for Prolog is the K.U.Leuven CHR system [21,20], originally hosted by hProlog, but meanwhile ported to two major open-source Prolog systems: SWI-Prolog and XSB.

Another programming language that saw the light in 1991 is Java. This platform-independent, object-oriented language has become very popular over the years. There is a growing interest to extend Java with constraint handling

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

support for implementing applications like planning and scheduling. For this purpose several specific constraint solvers have been implemented in Java [8,25,24]. These so-called black-box solvers require substantial effort in program development and the resulting programs are hard to maintain, modify and extend. A user-friendly and efficient CHR system for Java would considerably simplify the design of new application-tailored constraint solvers for Java, and it would also enable the introduction of the numerous solvers already written in CHR into Java.

Two (prototype) CHR systems already existed in Java at the moment our research started. The first, JCHR,[4] is reasonably user-friendly, but lacks efficiency as we will demonstrate in Section 4. The second, DJCHR [27], is more efficient, but fails to satisfy the user-friendliness requirement. Neither system truly integrates the CHR language into Java, but rather introduces a form of logic programming (LP) on top of Java. Interaction with constraints other than equality is not possible, nor does the design allow for an easy incorporation of new optimizations. The goal we set ourselves in designing and implementing a new system is therefore threefold:

1. **User-friendliness** The high-level syntax used should feel familiar to both CHR and Java adepts. The host language has to be Java (and not "LP in Java"), so one should be able to use regular Java-variables and arbitrary Java-expressions. Also, we must not lose track of the declarative nature of CHR and Constraint Programming (CP): programming in terms of CHR rules and constraints (both built-in and user-defined) should be fully supported. Moreover, to allow an easy porting of existing solvers, support for the same familiar semantics is at least as important as a similar syntax.
2. **Flexibility** Adding of and experimenting with new built-in solvers, built-in constraints, static program analyses and optimizations should be possible. Therefore a well thought-out design is indispensable.
3. **Efficiency** The CHR constraint handlers should be efficient, preferably competitive with their Prolog counterparts. For this goal one can rely on the wide experience acquired with the existing Prolog and HAL systems: several techniques and optimizations can be tailored to the new host language.

Overview We assume the reader to be acquainted with Java, Constraint (Logic) Programming [7,12] and the syntax and semantics of Constraint Handling Rules [11]. Some familiarity with the implementation of CHR in Prolog systems [16] is also helpful. In Section 2 we start with a brief review of some general principles, after which we provide an overview of the two pre-existing Java implementations and their respective shortcomings. Next, Section 3 discusses the K.U.Leuven JCHR System in detail. The performance of the K.U.Leuven JCHR System is compared with that of several other systems in Section 4. Finally, Section 5 concludes.

This paper is based on work done in the context of the master thesis of the first author [26] (in Dutch), in which a more comprehensive exposition can be found.

2 Implementation of CHR: Related Work

2.1 General Principles

Following the well-known reference implementation by Christian Holzbaur [16], a typical CHR system consists of two parts:

- A *compiler* that compiles CHR rules into code of the host language. For a survey of the compilation scheme used in the reference implementation we refer the reader to [20].
- A *runtime (environment)* containing generic code that is reused by all CHR programs. The most important component of the runtime is the constraint store, providing efficient lookup of partner constraints and reactivation of user-defined constraints.
  Because the genericity of the runtime code has its price, namely runtime overhead, its role is more and more taken over by specialized code generated by optimizing compilers.

A CHR implementation has to follow the semantics of the language. Besides the declarative semantics [11] (which lets a CHR program easily be understood without knowing how it is executed), several versions of operational semantics can be found throughout the literature. The theoretical operational semantics [2] is highly non-deterministic and therefore still too high-level to fully describe the behavior of most current CHR implementations. The refined operational semantics, formally defined in [10] and implemented in all major CHR systems, remedies this shortcoming by establishing a partial order of execution and treating CHR-constraints as procedure calls.

2.2 JCHR (JaCK)

The Java Constraint Kit (JaCK) [4] is a library for constraint programming (CP) in Java. It is composed of:

JCHR: Java Constraint Handling Rules (JCHR) [19] is the CHR dialect with matching compiler and runtime we will focus on shortly.

VisualCHR: An interactive tool visualizing the execution of JCHR [5].

JASE: CHR is often not capable of solving non-trivial CP problems unless combined with search. Java Abstract Search Engine (JASE) [17] is a framework designed to work in tandem with JCHR. As soon as JCHR is no longer able to do any work, JASE kicks in and makes a choice about which direction a solution is to be searched in. When JCHR detects an inconsistency JASE backtracks to the most recent choice-point. JASE is considered abstract because it allows for the specification of the search-strategy.

The JCHR syntax, illustrated in Fig. 1, is a combination of CHR and Java [19]

> [...] intended to be very similar to Java, in order to provide an intuitive programming.

```
handler leq {
    class java.lang.Integer;

    constraint leq(java.lang.Integer, java.lang.Integer);

    rules {
        variable java.lang.Integer X, Y, Z;

        if (X = Y)   { leq(X, Y) }  <=> { true }      reflexivity;
        { leq(X, Y) && leq(Y, X) }  <=> { X = Y }     antisymmetry;
        { leq(X, Y) &\& leq(X, Y) } <=> { true }      idempotence;
        { leq(X, Y) && leq(Y, Z) }  ==> { leq(X, Z) } transitivity;
    }
}
```

Fig. 1. A leq-handler in JaCK's JCHR

In pursuit of this goal the authors have made some changes to the original CHR syntax that, in our opinion, are unnecessary or even counterintuitive. We will elaborate further on these and other syntactic elements when we contrast them with our approach in Section 3. The variables used by JaCK are exclusively logical variables. The most important conceptual difference with a Prolog system is that they are typed (meaning they can only be bound to variables or values of the same[1] type), necessitating variable declarations. Likewise arguments of user-defined constraints are also typed, leading to extended constraint-declarations (cf. Fig. 1 for an example). The only built-in constraint known (and possible!) is equality, which appears to be implemented with an unoptimized version of the union-find algorithm [22,13]. Besides user-defined constraints, equality and logical variables, a very limited amount of Java-like expressions can be used. Some observations:

- In JCHR one cannot use a variable as implicit argument (as in X.someMethod()), since [19]

  > [...] the programmer is forced to use methods, which [take the variables as parameters and] return the result of their computation. [...] It is a question of design, whether to provide this functionality or not. [JCHR] does not implement this property of Java variables in JCHR variables, in order to stay closer to the properties of logical variables.

- Some simple syntactic sugar is used for the following literals: integer-, float- and string-objects. For example a "1" always gets translated into "new Integer(1)". The undesirable result is that primitive values can *never* be used.

The so-called JCHR-Compiler is actually merely a tool that translates a JCHR program to high-level, unoptimized Java-code. As [3] states:

[1] Unfortunately in the current implementation: *exactly* the same type...

> [...] it is possible to create rules directly in Java. But the compiler is a useful tool, if CHR is already known from Prolog.

Note that this last statement is questionable, given the numerous syntactical changes that were introduced. Later on this intermediate, "compiled" form is interpreted by the runtime (called the JCHRCore-Evaluator). The fact that virtually all code is generic results in high runtime penalties, which can be avoided when compiling to specialized, lower-level code. The only real effort in making the execution more efficient seems to be the use of path indexing [23] for partner constraint lookup. Furthermore, the interpreter does not follow the refined operational semantics. Even worse maybe is that the implemented semantics is not even completely in compliance to the theoretical operational semantics. The former shortcoming leads to the observation that programming can become rather impractical or counterintuitive, whilst the latter actually means that strictly speeking JCHR isn't even a form of CHR.

2.3 DJCHR

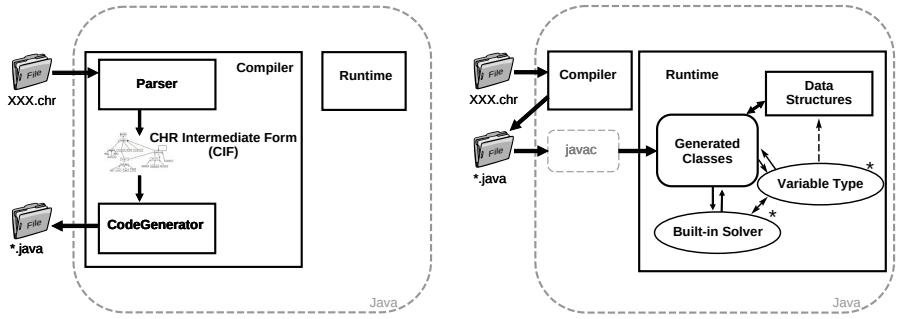
DJCHR aims at introducing a *dynamic* variant of CHR in Java. In practice, many constraint problems change continuously: arbitrary constraints can be added and removed at any time. After each change the solution has to be adapted, rather than solved from scratch. The fact that such applications are typically used online, makes requirements as efficiency and stability all the more important.

A dynamic version of CHR lets all existing CHR handlers be used in dynamically changing environments. Whilst incremental addition of constraints is already characteristic of CHR, to support arbitrary deletions a new incremental operational semantics had to be devised [28].

Syntactically, DJCHR [27] completely follows logic programming: logical variables and compound terms are supported, as are some numerical constants. The only implemented built-in constraint is term equality, using an adaptive unification algorithm. Unfortunately, DJCHR handlers can only be coded directly in Java-code, using expressions similar to the ones JaCK translates its high-level, declarative JCHR programs to. On the other hand, DJCHR *does* compile these Java-coded rules to more efficient, low-level Java-code. The compilation scheme and runtime system are considerably different from other CHR systems because of the dynamic requirements. Based on the accompanying documentation, we presume several optimizations are yet to be made. But given the compiler-design – a single 5000-line class – it looks like this might not always be equally straightforward.

3 The K.U.Leuven JCHR System

As most CHR systems the K.U.Leuven JCHR system consists of two main components: a compiler and a runtime. An overview of the architecture can be found in Figure 2.



(a) The Compiler Subsystem (b) The Runtime Subsystem

Fig. 2. High-level architectural overview of the K.U.Leuven JCHR system.

The compiler subsystem (Fig. 2(a)) takes a K.U.Leuven JCHR source file as input and outputs Java source files. Its kernel consists of a parser generated from an ANTLR-grammar and code generation driven by FreeMarker templates. In between these stages a flexible and complete model of the JCHR program is constructed, called *CHR Intermediate Form* (CIF).

The runtime subsystem (Fig. 2(b)) consists of several generic data-structures (mainly lists and iterators), built-in solvers and variable types. Together with those generated by the compiler subsystem, these generic classes make up an efficient handler capable of solving the user-defined constraints.

So, conceptually, we take the first stage of JaCK – a high-level, declarative syntax with corresponding ANTLR-generated parser – followed by the final stage of DJCHR – full compilation to low-level Java-code. But this is mostly where the analogy ends: we will show that there exists significant differences in both design-goals and their implementation. The concrete aim can succinctly be summarized as follows:

$$\text{K.U.Leuven JCHR} = \text{Java} + \text{CP} + \text{CHR} \tag{1}$$

Hereby it is fundamental that the three terms in right hand side of the equation are equally important. As seen in the previous section, the sum for both other Java-implementations leans more towards "Java + LP + CHR", with a rather inferior role assigned to the integration with Java or even to some important aspects of CHR.

3.1 User-friendliness

The high-level syntax we propose is aimed at an intuitive and familiar look and feel for users of both Java and existing (Prolog-)CHR-systems, allowing a declarative implementation of efficient constraint solvers in Java. Although we have this goal in common with JaCK[2], there are several important distinctions.

[2] Henceforth we will simply use JaCK to refer to the JCHR-portion of this library, to avoid confusion with K.U.Leuven JCHR.

Figure 3 shows the classical leq-handler in K.U.Leuven JCHR syntax, illustrating some but not all aspects mentioned throughout this subsection. More example programs can be found on `http://www.cs.kuleuven.be/~toms/CHR/JCHR/`.

In the remainder of this subsection we will zoom in on several syntactical aspects, grouped under the three terms of Equation 1.

```
import runtime.LogicalVariable;

handler leq<T> {
    solver runtime.EqualitySolver<T>;

    constraint leq(LogicalVariable<T> X,
                   LogicalVariable<T> Y) infix =<;

    rules {
        variable LogicalVariable<T> X, Y, Z;

        reflexivity  @ X =< Y <=> X = Y | true;
        antisymmetry @ X =< Y , Y =< X <=> X = Y;
        idempotence  @ X =< Y \ X =< Y <=> true;
        transitivity @ X =< Y , Y =< Z ==> X =< Z;
    }
}
```

Fig. 3. A leq-handler in K.U.Leuven JCHR

CHR The core of a CHR handler, the declaration of the rules itself, is superficially analogous to the original CHR syntax. Contrary to what is the case in JaCK, this means that the (optional) name precedes a rule and the guard follows the rule-operator. Also, the brackets and the "if (...)"-construction have been removed. The reason is that these changes (which were introduced by JaCK to make the syntax more Java-like), in our opinion, only hinder the rapid porting of existing handlers. Sometimes they even feel counterintuitive, e.g. when placing constraints, i.e. the guard, on variables only introduced afterwards. Only the possibility to separate conjunctions with double ampersand ('&&') and to end a rule with a semicolon (';') is kept, as an alternative to the common Prolog-like notation (',' and '.' respectively). The ugly '&\&' in simpagation rules though has been discarded, in favor of '\'.

Java When looking at the conjuncts, arguments and variables we see that the K.U.Leuven JCHR language leans more towards Java than JaCK does.

Java variables The introductory text [12] reminds us of the following:

> The key aspect in the CLP(X) scheme is to provide the user with more expressiveness and flexibility concerning the primitive objects the language can manipulate. Clearly the user wants to design his application using concepts that are as close as possible to his domain of discourse, e.g. he wants to use sets, boolean expressions, integers, rationals or reals, instead of coding everything as uninterpreted structures, i.e. finite trees [Herbrand terms], as is advocated in logic programming.

Since using real-world concepts is one of the cornerstones of object-oriented programming, an artificial limitation to logical variables (as is the case in DJCHR) would be a step back. The typed logical variables used by JaCK are a step in the right direction, but are still too restrictive and counterintuitive for a Java-programmer. The K.U.Leuven JCHR system therefore basically uses fully-fledged Java-variables, which in principle can have any valid Java-type, including primitive types and generic types (new in Java 1.5). Thanks to the fully supported generic types [6] we are e.g. able to use `LogicalVariable<T>` as a variable-type, substituting `T` for the type we want our logical variable to have. These variables are completely equivalent with the typed logical variables of JaCK. Important to note however is that conceptually this type in itself is no different from any other Java-type. Note that this is just one of the many possibilities: other interesting options include finite-domain variables or even complete Herbrand terms (like the ones used in DJCHR or Prolog), although some syntactic sugar might be in order here.

Java expressions These Java variables can be used in a wide range of Java-expressions, as a conjunct in guard or body or as an argument. A key difference with JaCK is that Java variables *can* be used as implicit argument.

Generic handlers To ease the transition from untyped Prolog to strongly typed Java we have introduced generic handlers. Without this feature we would e.g. have to implement a different leq-handler (cf. Fig. 3) for each variable-type required. To the best of our knowledge the K.U.Leuven JCHR system is the first typed CHR-system that adequately deals with polymorphic handlers this way. Furthermore this feature makes a handler-declaration completely analogous to a Java-class, allowing an arbitrary number of formal type parameters, each having any number of upper-bounds. For more information on these concepts and some constraints we refer the reader to e.g. [6].

Other Java-related syntactical aspects

- Since our variables are typed, they also have to be declared. We've used the same keyword as JaCK, i.e. `variable`, for this purpose. Note nevertheless that we are not declaring logical variables, as is the case in JaCK, but Java-variables!
- Whereas Prolog systems declare constraints using the keyword `constraints` followed by a comma-separated list, JaCK uses the `constraint` keyword for this, declaring each constraint separately. We support both options. As is

the case of JaCK, the arguments of user-defined constraints are also typed, naturally leading to a declaration analogous to a method-signature in Java. To strengthen this analogy, but also to improve readability of debugging information and the generated code, one can also give these arguments a name in K.U.Leuven JCHR.
- An infix-notation can be used for constraints, as illustrated in Fig. 3.
- JaCK uses obligatory `class`-declarations, while still requiring to use the fully qualified name everywhere. K.U.Leuven JCHR on the other hand uses `import`s completely analogous to Java imports[3], i.e. the use is only an optional tool, but once a type is imported it is no longer necessary to use the fully qualified name. To strengthen the analogy with a Java-class even further, but also because one has to be able to use imported classes in a generic handler-declaration, these imports have been moved in front of the handler-block.

CP Besides the use of Java variables and expressions, programming in terms of constraints is fully supported. Inspired by the HAL CHR-system [9] arbitrary built-in constraints and solvers can be used: in no way do we restrict the system to equality. For this the K.U.Leuven JCHR system considers equality over logical variables (cf. JaCK) or Herbrand terms (cf. DJCHR and Prolog) "just another constraint".

A final, declarative feature offered by the K.U.Leuven JCHR system is *coercion*. For Java-programmers: this can be seen as an extension of the new auto-boxing/unboxing feature in Java 1.5. Thanks to automatic coercion one can for example simply type "`3`" instead of the cumbersome expression "`new LogicalVariable<Integer>(Integer.valueOf(3))`", or vice versa use an expression whose type is `LogicalVariable<Float>` in places where a `Float` or `float` is expected. In other words, coercion vastly improves over the simple syntactic sugar used by JaCK. Thanks to this feature the programmer seldom has to be concerned with typing (e.g. when porting from untyped Prolog) and can thus concentrate on what really matters.

3.2 Flexibility

The next important criterion is the flexibility of a system, encompassing its (re)usability, adaptability and extensibility. In general this implies adopting a well thought-out design, using among other things suitable design-patterns [14] (cf. also Fig. 2), but several other aspects are equally important. We review some in this section.

CP As seen in the previous subsection one can in principle use arbitrary high-level built-in constraints. This necessitates a means to inform the compiler about the Java-expressions they can be translated to. The K.U.Leuven JCHR system

[3] Package imports and static imports are not yet implemented at the time of writing.

implicitly knows about the the special class `Comparator`-solvers and most frequently used constraints in Java, including (dis)equality-constraints over primitive variables and `Comparable` objects. For other constraints their respective solvers have to be disclosed to the system via a `solver`-declaration (as illustrated in Fig. 3). Furthermore we have chosen to use annotations, the new meta-data facility of Java 1.5, to declare which method(s) have to be used for which constraint. An example of an annotated solver-interface can be seen in Figure 4. Using extensive reflection and some heuristics the K.U.Leuven JCHR compiler is herewith capable of transforming constraint-statements into their corresponding Java-expressions (possibly coercing the arguments as well).

```
import annotations.*;

@CHRconstraints({
    @CHRconstraint(identifier = "eq", arity = 2, infix = "=")
})
public interface EqualitySolver<T> {
    @CHRtells("eq")
    public void tellEqual(LogicalVariable<T> X, T val);
    @CHRtells("eq")
    public void tellEqual(T val, LogicalVariable<T> X);
    @CHRtells("eq")
    public void tellEqual(LogicalVariable<T> X, LogicalVariable<T> Y);

    @CHRasks("eq")
    public boolean askEqual(LogicalVariable<T> X, T val);
    @CHRasks("eq")
    public boolean askEqual(T val, LogicalVariable<T> X);
    @CHRasks("eq")
    public boolean askEqual(LogicalVariable<T> X, LogicalVariable<T> Y);
}
```

Fig. 4. An annotated solver-interface.

Java Another feature reviewed in the previous subsection is the use of arbitrary variable types. Again the system is implicitly fully informed about the most frequent types, including all eight primitive types and their wrappers and `String`s, while for other types the necessary information (needed e.g. for coercion and initialization) can easily be explicitly declared using annotations.

CHR The K.U.Leuven JCHR system uses ANTLR [18] (parser-generator) and FreeMarker [1] (template-engine used for code-generation), which enjoy advantages very similar to declarative programming, like readability and adaptability.

The FreeMarker-templates especially have already proven their flexibility when introducing several optimizations to the compilation scheme.

The flexible intermediate form, named CIF (cf. Fig. 2(a)), introduced between parsing and code-generation is well-suited for static program analysis. Furthermore a framework has been designed to allow for an easy introduction of indices for the constraint store.

With this last subsection we are clearly already on the verge of the final requirement we will consider, namely efficiency.

3.3 Efficiency

In this subsection we will review some implemented efficiency-improvements in the successive stages of the system, ranging from the analysis of a CHR-program, over the corresponding generated code, to its actual execution. From the previous subsection we already know that it should be easy to add several other improvements. In the next section then we will put the current implementation to the test as we compare its performance with that of several other systems.

Program analysis One type of static program analysis has been implemented, which we have called scheduling. Implemented schedulers use techniques like early guard scheduling and the introduction of lookups through the appropriate indices to speed up the searching of partner constraints. This approach is designed to be easily extended towards e.g. join ordering [15].

Compiled code The compilation scheme used, though originally based on a typical Prolog-schemes [20,16], has been fully tailored to the new host language. Equivalents of all simple optimizations listed in [20] are implemented, many for the first time in a Java system. Because Java is an imperative language, the continuation-based control flow has been made significantly more sequential. Exactly this is also what we exploited by introducing a new optimization called explicit backjumping (in contrast to the implicit backtracking used in Prolog systems), possibly capable of avoiding trashing, one of the well-know drawbacks of backtracking.

As with other more mature systems, the K.U.Leuven JCHR system has seen a gradual movement from generic runtime-code towards more efficient code generated by the compiler. The constraint store for instance consists in the current version for the most part of specialized generated code.

Indices After the K.U.Leuven CHR system, the K.U.Leuven JCHR system is to the best of our knowledge the second CHR implementation allowing hash-table indices for the constraint store, speeding up lookups based on equality constraints in the guard (or implicit constraints originating from using the same variable multiple times in a head). In addition, for the first time hash-indices can be used for variables whose hash-value might change over time (in Prolog this corresponds to non-ground terms).

Built-in solvers The only non-trivial built-in constraint solver already implemented for the system handles equality constraints over logical variables (cf. Fig. 4). Its implementation is based on the union-find algorithm, optimized with path compression and union-by-rank [22,13].

4 Experimental Evaluation

In this section we compare the performance of five different CHR-systems: two Prolog-systems, the SICStus reference implementation (for short: **SICStus**) and the K.U.Leuven CHR system as ported to SWI-Prolog (**SWI**) and all three existing Java systems (**DJCHR**, **JaCK** and of course the **K.U.Leuven JCHR** system). The benchmarks used in this section can be found on `http://www.cs.kuleuven.be/~toms/CHR/JCHR/`. We were unable to port some benchmarks to JaCK and/or DJCHR, partially due to the differences in semantics.

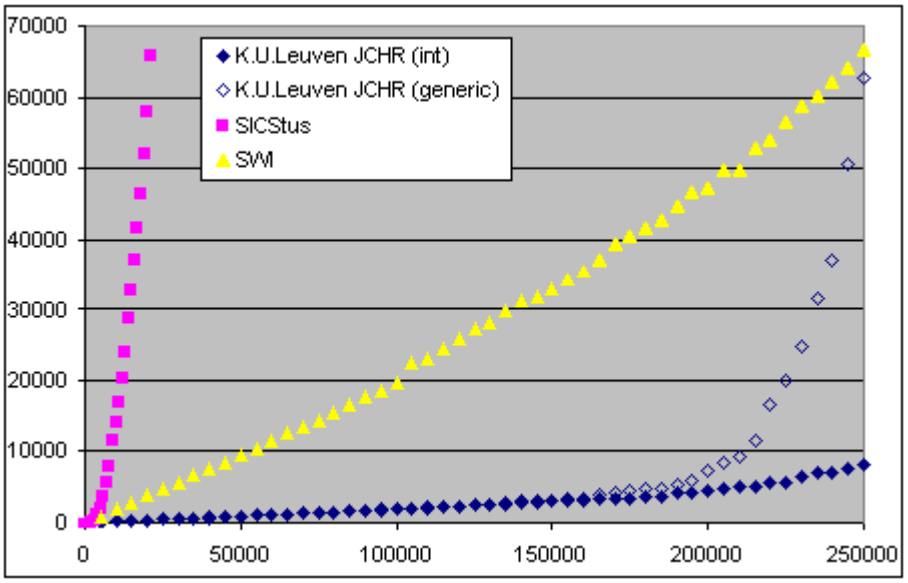


Fig. 5. Results for the union-benchmark.

Figure 5 shows the results of the union-benchmark. **SWI** and **K.U.Leuven JCHR**, both featuring a constraint store indexed with a hash-table, show an optimal, quasi-linear behavior. **SICStus** on the other hand, lacking the necessary constant time operations on the constraint store, has a complexity that appears to be quadratic or worse. For **K.U.Leuven JCHR** the figure contains benchmark-results for both a version of the union-find handler using primitive

integers and a generic version, using String objects to be precise. The performance difference between the two is negligible, until the latter causes garbage collection to become a performance-bottleneck.

In the Fibonacci-benchmark (Fig. 6(a)) the three systems considered behave similarly. The same holds for this trio in the next benchmark, performing the well-known "Sieve of Eratosthenes" algorithm (Fig. 6(b)). Furthermore this benchmark also clearly illustrates the lack of efficiency in **JaCK**. While the other systems remain within a quarter of a second, **JaCK** rapidly takes more than a minute and a half. This performance gap is also noticeable in the two remaining benchmarks: the leq- (Fig. 6(c)) and bool-benchmark (Fig. 6(d): see the top-left corner!).

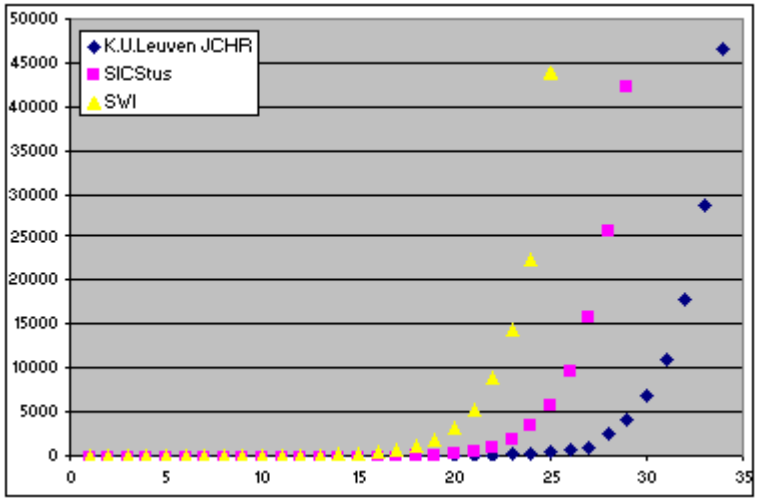
Memory seems the only limiting factor in most benchmarks, with the apparent exception of the fib-benchmark[4]. In all but this last benchmark and the highly recursive primes-benchmark (where it is a stack-overflow that occurs) **K.U.Leuven JCHR** eventually runs out of heap-space. The same occurs to **DJCHR** sometimes even a lot sooner, like e.g. in the bool-benchmark. Memory-difficulties are not restricted to the Java systems: **SWI** too eventually overflows a stack in the fib-, leq- and bool-benchmark. In this respect **SICStus** clearly outperforms the other systems.

Time-wise, **K.U.Leuven JCHR** outperforms most often the other systems. There are several diverse reasons for that (cf. also Section 3.3), but some general observations can be made. We have already noted that the performance gap with **JaCK** can largely be explained by the fact that **JaCK** does not really compile CHR. The difference with **DJCHR** might be partially caused by the extent of book-keeping needed for the dynamic requirements. In fairness **K.U.Leuven JCHR** also does not yet have to keep the book-keeping needed for backtracking, but we are confident this can be added efficiently. An important factor contributing to the performance of **K.U.Leuven JCHR** is undoubtedly the just-in-time compilation in a mature Java-runtime, which can account for up to a an order of magnitude.

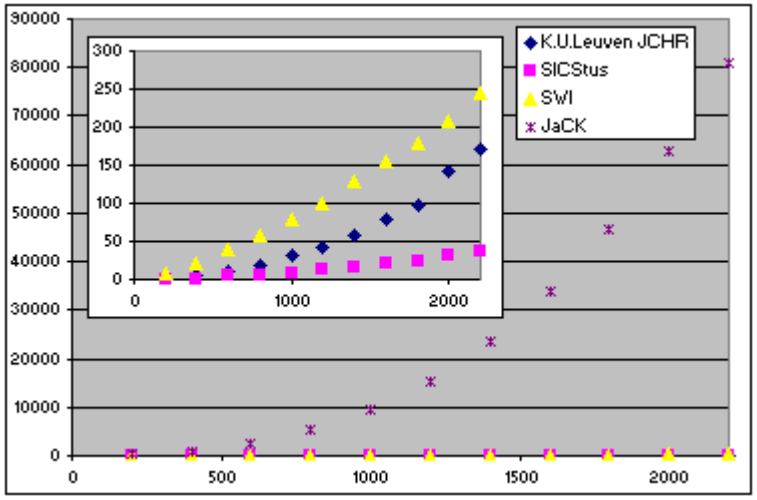
5 Conclusion

Our contribution is the K.U.Leuven JCHR system, a user-friendly, flexible and efficient CHR system in Java. The proposed high-level, declarative syntax and the full compliance to the refined operational semantics offer an intuitive look and feel, and easy porting of already existing CHR code. The flexible design allows for an integration of arbitrary static program analyses, compilation optimizations, hash-indices and constraint solvers, as demonstrated by features already implemented. We have shown how to apply the optimized compilation techniques of CHR for Prolog to the object-oriented programming paradigm and that this leads to superior efficiency.

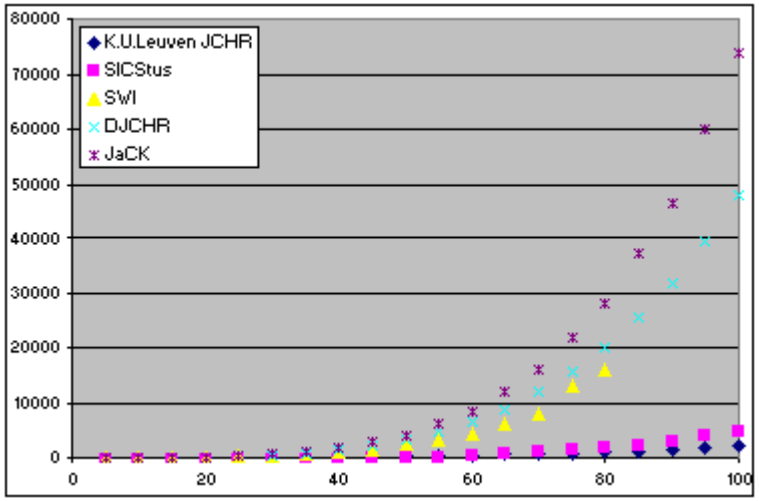
[4] However when one uses a simpagation rule instead of a simplification rule for the first rule (cf. the website for the program), memory once again becomes the only limiting factor instead of the execution-time!



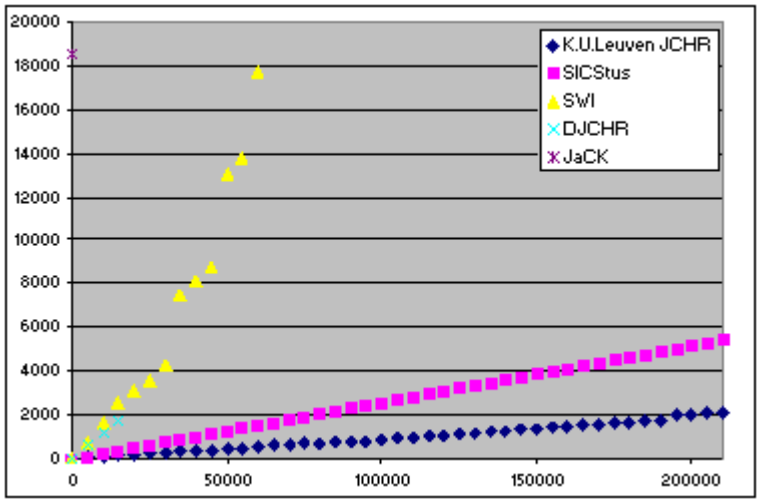
(a) fib



(b) primes



(c) leq



(d) bool

Fig. 6. Benchmark results.

The K.U.Leuven JCHR system is available at `http://www.cs.kuleuven.be/~toms/CHR/JCHR/`.

5.1 Future Work

We intend to add support for backtracking to the system. This feature is essential when CHR is to be used for partial constraint solvers that require some form of search. For backtracking the constraint store data structures need to be extended with support for undoing updates. However, the most important challenge is to investigate how ordinary Java method calls should interact with backtracking.

An efficient implementation of CHR in Java opens the door for efficient CHR implementations in other imperative (object-oriented) languages, such as C, C#, Visual Basic, ... The general principles are already outlined in this paper, but the tight integration of CHR with the particular host language should be considered for each language separately.

References

1. The FreeMarker website. `http://www.freemarker.org`.
2. Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *Proceedings of Third International Conference on Principles and Practice of Constraint Programming*, pages 252–266, Schloss Hagenberg, Austria, 1997. Springer LNCS.
3. Slim Abdennadher. *Rule-based Constraint Programming: Theory and Practice*. PhD thesis, Institute of Computer Science, LMU, Munich, 2001.
4. Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JaCK: A Java Constraint Kit. In *Proceedings International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, Kiel, Germany, September 13-15 2001. Society of Logic Programming, University of Kiel; published as Tech Report No. 2017.
5. Slim Abdennadher and Matthias Saft. A Visualization Tool for Constraint Handling Rules. In *Proceedings of 11th Workshop on Logic Programming Environments*, 1th December 2001.
6. Gilad Bracha. *Generics in the Java Programming Language*. Sun Microsystems, Inc., July 2004.
7. Manuel Carro et al. An introductory course on Constraint Logic Programming. `http://clip.dia.fi.upm.es/~vocal/public_info/`, December 1998.
8. H. Chun. Constraint Programming in Java with JSolver. In *Proceedings of the First International Conference and Exhibition on the Practical Application of Constraint Technologies and Logic Programming*, London, 1999.
9. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. Extending arbitrary solvers with Constraint Handling Rules. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 79–90, New York, NY, USA, 2003. ACM Press.
10. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, pages 90–104, Saint-Malo, France, September 2004.

11. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriott, Eds.), Journal of Logic Programming*, 37(1–3):95–138, October 1998.
12. Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. Constraint Logic Programming – an informal introduction. Technical Report ECRC-93-5, ECRC, Munich, Germany, February 1993.
13. Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. In *ACM Computing Surveys (CSUR)*, volume 23, pages 319–344. September 1991.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
15. Christian Holzbaur, María García de la Banda, David Jeffery, and Peter J. Stuckey. Optimizing compilation of constraint handling rules. In *Proceedings of the 17th International Conference on Logic Programming*, pages 74–89, London, UK, 2001. Springer-Verlag.
16. Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules compiler and runtime system. *Applied Artificial Intelligence*, 14(4):369–388, April 2000.
17. Ekkehard Krämer. A generic search engine for a Java Constraint Kit. Master's thesis, Institute of Computer Science, LMU, Munich, Germany, January 15 2001.
18. Terence J. Parr et al. The ANTLR website. `http://www.antlr.org`.
19. Matthias Schmauß. An Implementation of CHR in Java. Master's thesis, Department of Computer Science, University of Munich, Germany, 1999.
20. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, jun 2005.
21. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *Proceedings of the First Workshop on Constraint Handling Rules: Selected Contributions*, Ulm, Germany, May 2004.
22. Tom Schrijvers and Thom Frühwirth. Implementing and analysing union-find in CHR. Report CW 389, K.U.Leuven, Department of Computer Science, July 2004.
23. Mark E. Stickel. The path-indexing method for indexing terms. Technical Report 473, 333 Ravenswood Ave., Menlo Park, CA 94025, 1989.
24. Naoyuki Tamura. Cream: Class library for constraint programming in java. `http://bach.istc.kobe-u.ac.jp/cream/`, 2004–?
25. Paolo Viappiani et al. JCL – the Java Constraints Library. `http://liawww.epfl.ch/JCL/`.
26. Peter Van Weert. Constraint Programming in Java: een gebruiksvriendelijk, flexibel en efficiënt CHR-Systeem voor Java. Master's thesis, K.U.Leuven, May 2005.
27. Armin Wolf. Adaptive constraint handling with CHR in Java. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, number 2239 in Lecture Notes in Computer Science, Paphos, Cyprus, Nov./Dec. 2001. Springer-Verlag.
28. Armin Wolf, Thomas Gruenhagen, and Ulrich Geske. On incremental adaptation of CHR derivations. *Journal of Applied Artificial Intelligence*, Special issue on Constraint Handling Rules, 2000.