Towards a Generic Trace for Rule Based Constraint Reasoning

Armando Goncalves\textsuperscript{1,2}, Marcos Aurelio\textsuperscript{3}, Pierre Deransart\textsuperscript{2} and Jacques Robin\textsuperscript{1}

\textsuperscript{1} Universidade Federal de Pernambuco, Recife, Brazil
\textsuperscript{2} INRIA, Centre de Paris-Rocquencourt, France
\textsuperscript{3} Université de Paris 6, France

Abstract. CHR is a very versatile programming language that allows programmers to declaratively specify constraint solvers. An important part of the development of such solvers is in their testing and debugging phases. Current CHR implementations support those phases by offering tracing facilities with limited information. In this paper, we propose a new trace for CHR which contains enough information to analyze any aspects of CHR execution at some general abstract level. This approach is based on the idea of generic trace. Such a trace is formally defined as an extension of the \(\omega^\vee\) semantics. It is currently prototyped in a SWI Prolog based CHR implementation.

1 Introduction

CHR (Constraint Handling Rules)\textsuperscript{[9]} is a uniquely versatile and semantically well-founded programming language. It allows programmers to specify constraint solvers in a very declarative way. An important part of the development of such solvers is in their testing and debugging phases. Current CHR implementations support those phases by offering tracing facilities with limited information.

In this paper, we propose a new trace for CHR which contains enough information, including source code ones, to analyze any aspects of CHR\(^\vee\) execution at some abstract level, general enough to cover several implementations and source level analysis. Although the idea of formal specification based tracer is not new (see for example \textsuperscript{[13]}), the main novelty leads in the generic aspect of the trace. Most of the existing implementations of CHR like in \textsuperscript{[11,12,19]} include a tracer with specific CHR ports, but without formal specification, nor consideration with regards with different kind of usages than debugging.

The notion of generic trace has been informally introduced and used for defining portable CLP(FD) tracer and portable applications \textsuperscript{[11,13]}. We propose here to use this approach to specify a tracer for rule based inference engine like CHR\(^\vee\). A generic trace has three main characteristics: it is “high level” in the sense that it is independent from particular implementations of CHR, it has a specified semantics (Observational Semantics), it can be used to implement debugging tools or applications. In \textsuperscript{[17]} it is shown that it can be adapted for software component based programming.
We present a generic trace for CHR based on its refined operational semantics \( \omega_r \), and describe a first prototype developed for SWI-Prolog CHR engine. The implementation consists of combining the original trace of the SWI engine with source code information to get generic trace events, and then, allowing the user to filter these events using a SQL-based language.

This paper is organized as follows. Section 2 gives a short introduction to generic traces. Section 3 presents CHR, its \( \omega_r \) semantics and the observational semantics, OS-CHR, defining the generic trace. Section 4 the CHR-SWI-Prolog based prototype. Section 5 presents the experimentation. Discussion and conclusions are in the two last sections.

2 Generic Trace, Observational Semantics, and Subtrace

The concept of generic trace has been first introduced in [14], formally defined in [5, 6], and a first application to CHR presented in [17]. A generic trace is a trace with a specification based on a partial operational semantics applicable to a family of processes. We give here its main characteristics and the way to specify a generic trace.

2.1 Preliminaries

A trace consists of an initial state \( s_0 \) followed by an ordered finite or infinite sequence of trace events, denoted \( < s_0, \tau > \). \( T \) is a set of traces (finite or infinite). A prefix (finite, of size \( t \)) of a trace \( T = < s_0, \tau_n > \) (finite or infinite, here of size \( n \geq t \)) is a partial trace \( U_t = < s_0, \tau_t > \) which corresponds to the \( t \) first events of \( T \), with an initial state at the beginning. \( T \) may contain any prefixes of its elements.

A trace can be decomposed into segments containing trace events only, except prefixes which start with a state. An associative operator of concatenation will be used to denote sequences concatenations (denoted \( ++ \)). The neutral element is \( [] \) (empty sequence). A segment (or prefix) of size 0 is either an empty sequence or a state.

Traces are used to represent the evolution of systems by describing the evolution of their state. A state of the system is described by a given finite set of parameters and a state corresponds to a set of values of parameters. Such states will be said virtual as they correspond to states of the observed system, but they are not actually traced. We will thus distinguish between actual and virtual traces.

- the actual traces \( T^w \) are a way to observe the evolution of a system by generating traces. The events of an actual trace have the form \( e = (a) \) where \( a \) is an actual state described by a set of attributes values. An actual states is described by a finite set of attributes. Actual traces corresponds to sequences of events produced by a tracer of an observed system. They usually encode virtual states changes in a synthetic manner.
the virtual traces \( T^v \) corresponds to the sequence of the virtual states such that for each transition in the system between two virtual states, it corresponds an actual trace event. The virtual trace events have the form \( e = (r, s) \) where \( r \) is a type of action associated with a state transition and \( s \), called virtual state, the new state reached by the transition and described by a set of parameters. Virtual traces correspond to sequences of virtual states of the observed system which produced the actual trace, together with the kind of action which produced the virtual state transition.

The correspondence between both kinds of traces is specified by two functions \( E : T^v \rightarrow T^w \) and \( I : T^w \rightarrow T^v \), respectively the extraction and the reconstruction function, as illustrated by the figure 1.

![Fig. 1. Faithfulness Property](image)

The idea is that the actual generated trace contains as much information as possible in such way that the virtual trace can be reconstructed from the actual one. In other words, the extraction is done without loss of information. Such a property of the traces is called faithfulness and, if we denote \( Id_v \) (resp. \( Id_w \)) the identity between virtual traces (resp. actual traces), it states that \( E \circ I = Id_v \) (composition) or \( E = I^{-1} \), and \( I \circ E = Id_w \) (or \( I = E^{-1} \)).

2.2 Components in Trace Design

When designing a trace, several components must be taken into consideration. They are depicted in the Figure 2.

![Fig. 2. Components in Trace Design](image)

1. The observed process whose behavior is modeled by a virtual trace (sequence of successive virtual states) \( T^v \).
2. An extractor component which encodes the virtual trace into the actual one \( T^w \). This component corresponds in practice to the tracer formalized by the extraction function \( E \).

3. The driver which realizes the actual trace filtering according to some trace query. In this paper we limit its role to select a subtrace of the so called full trace.

4. The rebuilder which may reconstruct from a full or partial actual trace a full or partial virtual trace. This is possible only if there is no loss of information (faithfulness property). The rebuilder is formalized by the reconstruction function \( I \).

5. The analyzer, which corresponds to some debugging tool or particular application, working with the full trace or a partial one.

Notice that in practice the three first components may be intricate in the sense that for a given query the driver may select directly a subset of the virtual trace, thus avoiding to extract and encode a full actual trace before selecting a subtrace.

In this paper we focus on three components (observed process, extractor and rebuilder) and a property. Their description consists in a faithful observational semantics.

2.3 Observational Semantics

The evolution of a system defined by its virtual traces and the production of the corresponding actual trace can be described by a so called Observational Semantics as follows. More general definitions can be found in [6].

2.4 Characteristics of a Generic Trace

The idea of generic trace meets the needs of independent trace specification and portability. It is intended to specify a process or an algorithm by its observed behavior, i.e. the trace of abstracted operations that it is expected to implement. The level of description must be general enough to include family of processes, and the level of granularity must be sufficiently refined to be used by a family of applications. This may be the case for example for applications such as monitoring, debugging, visualization tools, or any application using the generic trace.

Definition 1 (Generic Trace (GT)).

Given a family of processes \( p \in P \), each of them equipped to produce traces \( T_p \), a set of traces \( T_g \) is generic if, for each process \( p \) in the family, there exists a derivation \( D_p \) of its traces which is a parametric subtrace of \( T_g \), that is:

\[
\forall p \in P, \exists T \text{ such that } D_{rvD_p}(T_p, T) \wedge Sub_P(T_g, T).
\]

Three questions are then worth posing:
How to ensure that the trace produced by some process is compliant with the GT?

Can the GT be used in application development, with the guarantee that the application will work with any compliant process?

Can the GT be extended to handle more processes in such a way that existing applications will still work?

Here are some possible answers.

Compliance to the Generic Trace

A trace of a process is compliant w.r.t. the GT if it satisfies the definition \[ \exists T', Drv_p(T_p, T') \land Sub_p(T_g, T'). \]

Building tools with the Generic Trace

The interest of a generic trace is that it facilitates the development of tools that can be used with all compliant processes. The development is made considering that the tool uses at least a sub-GT covering sufficiently many processes. Thus it is possible to adapt the tool to the process \( p \) by applying to the trace generated by the process (without any modification) the derivation \( D_p \) to get a GT. This can be done at the level of the process (process can use any tool) or at the level of the tool (tool can be run with this particular process).

The fact that the GT has a formal specification makes it possible to realize a prototype (executable specification) which shall be itself a new compliant process. It is thus possible to use such a prototype to develop and test tools. This development method guarantees that any tool made on the top of the GT will be able to work with any compliant processes.

Generic Trace Extensions

As long as an extension of the GT preserves the fact that a process is compliant w.r.t. a subtrace of the extended GT, they still are compliant w.r.t. the
extended GT. It is sufficient to ensure that any GT extension preserves the parametric subtraces. This guarantees that the compliant processes will continue to be usable by tools using the original GT.

2.5 Generic Trace Specification

By definition, if the observational semantics of a generic trace is faithful, a parametric subtrace of a virtual trace is a subtrace of the corresponding actual trace, from which the original virtual subtrace can be reconstructed. This is illustrated by figure 3. A query applied to the actual trace selects a partial actual trace in such a way that the resulting partial trace can be transformed in a partial virtual trace (the one from which the partial actual trace could be extracted). A practical consequence is that the definition of a generic trace should be given by a faithful observational semantics.

In practice, the generic trace specification will consists of an operational semantics corresponding to some abstract level of process observation, instrumented to produce an actual trace. The level of description (granularity of the events) should be chosen in such a way that this abstract operational semantics can be abstracted from each particular semantics of each process of the family. Symmetrically, it is requested that the abstract operational semantics can be “implemented” in each process of the family.

The faithfulness property of the observational semantics guarantees that the generic actual trace preserves the whole information concerning the process behavior, which can be deduced from the observation level corresponding to the given operational semantics.

3 Generic Trace for CHR

In this section we introduce the generic trace proposed for CHR. It is based on the refined Theoretical Operational Semantics for CHR, as defined in [3].

Such semantic is declarative enough to cover most of the CHR implementation. It is the case for ECLiPSe Prolog [2], SWI-Prolog [12] and GNU-Prolog [8] whose operational semantics can be viewed as a refinement of (versely can be viewed as an abstraction of the semantics of these implementations).

3.1 Operational Semantics

We define CT as the constraint theory which defines the semantic of the built-in constraints and thus models the internal solver which is in charge of handling them. We assume it supports at least the equality built-in. We use \([H|T]\) to indicate the first \((H)\) and the remaining \((T)\) terms in a list or stack, \(\_\) for pushing elements into stack, \(++\) for sequence concatenation and \([\_\_]\) for empty sequences. We use the notation \({a_0, \ldots, a_n}\) for both bags and sets. Bags are sets which allow repeats. We use \(\cup\) for set union and \(\uplus\) for bag union, and \{"\} to represent both the empty bag and the empty set. The identified constraints
have the form $c\#i$, where $c$ is a user-defined constraint and $i$ a natural number. They differentiate among copies of the same constraint in a bag. We also assume the functions $\text{chr}(c\#i) = c$ and $\text{id}(c\#i) = i$.

An execution state $\mathcal{E}$ is a tuple $(A, S, B, T)_n$, where

- $A$ is the execution stack;
- $S$ is the UDCS (User Defined Constraint Store), a bag of identified user-defined constraints;
- $B$ is the BICS (Built-in Constraint Store), a conjunction of constraints;
- $T$ is the Propagation History, a set of sequences for each recording the identities of the user-defined constraints which fired a rule;
- $n$ is the next free natural used to number an identified constraint.

Current alternatives are denoted as ordered sequence of execution states, $\mathcal{L} = \langle \mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n \rangle$ where $\mathcal{E}_1$ is the active execution state and $\langle \mathcal{E}_2, \ldots, \mathcal{E}_n \rangle$ the remaining alternatives.

The initial configuration is represented by $\mathcal{E}_0 = [(A, \{\}, \text{true}, \{\})]$. The top of execution stack $A$ is a constraint that will be processed and its initial value is determined by the initial goal. The transitions are applied non-deterministically until no transition is applicable. These transitions are defined as follows:

<table>
<thead>
<tr>
<th>Transition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solve-Wake</strong></td>
<td>$\langle[c#i]A, S, B, T \rangle_n \cup \mathcal{L} \mapsto \langle\text{wakeup}(S, e, B) + A, S, c \land B, T \rangle_n \cup \mathcal{L}$, where $c$ is built-in and $\text{wakeup}(S, e, B)$ is a function that implements the $\text{wake-up policy}$</td>
</tr>
<tr>
<td><strong>Activate</strong></td>
<td>$\langle[c#n : 1]A, {c#n} \cup S, B, T \rangle_{n+1} \cup \mathcal{L}$, where $c$ is user-defined constraint</td>
</tr>
<tr>
<td><strong>Reactivate</strong></td>
<td>$\langle[c#i]A, S, B, T \rangle_n \cup \mathcal{L} \mapsto \langle[c#n : 1]A, S, B, T \rangle_n \cup \mathcal{L}$, where $c$ is user-defined constraint</td>
</tr>
<tr>
<td><strong>Apply</strong></td>
<td>$\langle(A, H_1 \cup H_2 \cup S, B, T \rangle_n \cup \mathcal{L} \mapsto \langle(C + A, H_1 \cup S, c \land B, T' \rangle_n \cup \mathcal{L}$ where exists a rule $r \land H_1 \land H_2 \equiv g(C)$ and a matching substitution $e$, such that $\text{chr}(H_1) = e(H_1)$, $\text{chr}(H_2) = e(H_2)$ and $CT \models B \models \exists(e \land g)$ and the entry ${(r, \text{id}(H_1) + \text{id}(H_2))} \notin T$ and $T' = T \cup {(r, \text{id}(H_1) + \text{id}(H_2))}$.</td>
</tr>
<tr>
<td><strong>Drop</strong></td>
<td>$\langle[c#i : j]A, S, B, T \rangle_n \cup \mathcal{L} \mapsto \langle(A, S, B, T \rangle_n \cup \mathcal{L}$, where there is no occurrence $j$ for $c$ in the program.</td>
</tr>
<tr>
<td><strong>Default</strong></td>
<td>$\langle[c#i : j]A, S, B, T \rangle_n \cup \mathcal{L} \mapsto \langle[c#i : j + 1]A, S, B, T \rangle_n \cup \mathcal{L}$, if no other transition is possible in the current state.</td>
</tr>
<tr>
<td><strong>Split</strong></td>
<td>$\langle[c_1 \lor \ldots \lor c_m]A, S, B, T \rangle_n \cup \mathcal{L} \mapsto \langle[\sigma_1, \ldots, \sigma_n] \rangle$, where $\sigma_i = \langle[c_i]A, S, B, T \rangle_n$, for $1 \leq i \leq m$. This transition implements depth-first, other search strategies can be implemented by easily changing this definition.</td>
</tr>
<tr>
<td><strong>Fail</strong></td>
<td>$\mathcal{E} \cup \mathcal{L} \mapsto \mathcal{L}$, This transition is called automatically if $\mathcal{E}$ is a failed state. By definition a failed state occurs when the Built-in store is false.</td>
</tr>
</tbody>
</table>

### 3.2 Generic Trace

We introduce here informally the generic trace of CHR. Each transition in the $\omega^\gamma$ semantics should generate an actual trace event.
- **Wake a built-in constraint (BIC) is solicited**
  This event has 4 attributes and will be written as: \([\text{Wake}, c, \text{wakeup}(S, c, B), n]\). It will happen when a solve transitions is performed.

- **ActivateRDC activate a Rule defined constraint (RDC)**
  This event has 3 attributes and will be written as: \([\text{ActivateRDC}, c, n]\). It will happen when a solve transitions is performed, getting the RDC \(c\) from the top of the execution stack and activating it.

- **ReactivateRDC Activate a Rule defined constraint with justification**
  This event has 4 attributes, including: a RDC and a Wake event, which is the justification of the most recent Reactivate transition. It will be written: \([\text{Reactivate}, c\#i : j, [\text{Wake}, b, n], m]\).

- **TryRule attempt to apply a Rule**
  This event has 7 attributes, including: rule name, the active constraint, the constraints that match the \(\text{keep}\), the constraint that matches the \(\text{remove}\), the guard. It will be written:
  \([\text{TryRule}, \text{ruleName}, \text{activeConstraint}, \text{keep}, \text{remove}, \text{guard}, n]\),
  where \(\text{activeConstraint}\) has the form \(c\#i : j\).

- **ApplyRule apply the rule**
  After trying a rule, if the guard is true, the apply rule event will trigger. It has 7 attributes, including: the last corresponding TryRule event, added RDCs, preserved Head, removed RDCs, added BICs. The last three attributes are obtained from the rule’s body. It will be written:
  \([\text{ApplyRule}, [\text{TryRule}, \ldots], \text{addedRDCs}, \text{keep}, \text{remove}, \text{addedBICs}, n]\]

- **Drop drop a constraint**
  This event has 3 attributes, including a RDC, and will be written:
  \([\text{Drop}, c\#i : j, n]\). It corresponds to the end of the execution of \(c\), where \(c\) was an active constraint.

- **Default numbering incrementation**
  Occurs when virtual states numbering is incremented. It has 3 attributes and will be written: \([\text{Default}, j, n]\), where \(j\) is the new value.

- **Split create a disjunction**
  Occurs when a rule is disjunctive. It has 3 attributes, including the ApplyRule event with the rule whose body has the disjunction. It will be written:
  \([\text{Split}, [\text{ApplyRule}, \ldots], n]\).

- **Fail the rule application fails**
  Occurs when the Built-In store is false. It has 3 attributes, including the ApplyRule with the ultimate rule tested before failure. It will be written:
  \([\text{Fail}, [\text{ApplyRule}, \ldots], n]\). \(n\) is the numbering of the failed state.

All the variables which occur in the initial goal will keep their original name in all their occurrences in the generic trace. The formal definition of trace generation will be given in section 3.3.
3.3 Observational Semantics of CHR\textsuperscript{∨} (OS-CHR\textsuperscript{∨})

We specify the observational semantics of CHR\textsuperscript{∨}, OS-CHR\textsuperscript{∨}, on the top of the operational semantics of section \[\text{OOpSec}\]. The current generated actual trace is denoted \(N\), an ordered sequence of the trace events. The initial configuration will be represent as \(\mathcal{E} = \langle \{A\}, \{\}, \text{true}, \{\{\}\} \rangle \), \(N = \[]\).

<table>
<thead>
<tr>
<th>Solve+Wake</th>
</tr>
</thead>
<tbody>
<tr>
<td>[({c}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Activate</th>
</tr>
</thead>
<tbody>
<tr>
<td>[({c#n \land A}, S, B, T)_{n+1},</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reactivate</th>
</tr>
</thead>
<tbody>
<tr>
<td>[({c#i \land A}, S, B, T)_{n+1},</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Apply.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>[({c#i \land j \land A}, H_1 \lor H_2 \lor S, B, T)_{n+1},</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Apply.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\langle A, H_1 \lor H_2 \lor S, B, T \rangle_{n}, N \mapsto \langle C + A, H_1 \lor S, B, T \rangle_{n}, N++ \langle \text{TryRule}, \text{tryRule}(N), \text{addRDCs}(C), H_1, H_2, \text{addBICs}(C), n \rangle, \text{where CondApp2 (see below)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Drop</th>
</tr>
</thead>
<tbody>
<tr>
<td>[({c#i \land j \land A}, S, B, T)_{n+1},</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>[({c#i \land j \land A}, S, B, T)_{n+1},</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Split</th>
</tr>
</thead>
<tbody>
<tr>
<td>[({c_1 \lor ... \lor c_m \land A}, S, B, T)_{n+1},</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
</tr>
</tbody>
</table>

\(\text{SolveCond}: c\) is built-in, and \(\text{wakeup}(S, c, B)\) defines which CHR constraints of \(S\) that are woken by adding the constraint \(c\) to the built-in store \(B\).

\(\text{CondReac}: \) the function \(\text{wake} : \text{Constraint}, \text{Trace} \mapsto \text{Wake}\) is responsible for selecting the \text{Wake} event that justifies the Reactivate.

\(\text{CondApp1}: \) there exists a rule \(r@H_1' \setminus H_2' \Leftrightarrow g|C\) and a matching substitution \(e\), such that \(\text{chr}(H_1) = e(H_1')\), \(\text{chr}(H_2) = e(H_2')\) and \(\{(r, id(H_1)+id(H_2))\} \notin T\).

\(\text{CondApp2}: \) \(C\) is the body of the rule \(r@H_1' \setminus H_2' \Leftrightarrow g|C\). The \text{tryRule}: \text{Trace} \mapsto \text{TryRule}\) will retrieve the \text{TryRule} event generated by Apply.1. It will search for the event in the trace log, normally the event \text{TryRule} will be one step back. \(\text{addRDCs}: \text{Body} \mapsto \text{Sequence(RDC)}\) will select only the RDCs on the body; the function \(\text{addBICs}: \text{Body} \mapsto \text{Sequence(BIC)}\) will select the BICs on the body. Same conditions of \(\text{Apply.1}\) plus \(CT \models B \supset \exists(e \land g)\) and \(T' = T \cup \{(r, id(H_1)+id(H_2))\}\).

\(\text{CondSplit}: \) where \(\sigma_i = \langle A_i \land A\rangle, S, B, T \rangle_n\), for \(1 \leq i \leq m\), and \(\text{rule}: \text{Trace} \mapsto \text{ApplyRule}\) is a function that will retrieve the cause of the split, a disjunctive rule.

\(\text{CondFail}\): \(n\) is the numbering of the failed state \(\mathcal{E}\), and \(\text{rule}(N)\) is a function that will retrieve the cause of the failure.
4 Prototyping of a generic CHR ∨ Trace Engine using SWI Prolog

A generic CHR ∨ tracer for SWI-Prolog was developed. In Section 5.1, we introduce the SWI Prolog debug output trace produced when executing CHR rule bases. Section 5.2 presents our mapping from the produced trace into OS-CHR ∨

4.1 Running Example

The generic trace will be illustrated on a simple disjunctive graph-coloring problem. The following CHR ∨ rules define a graph coloring solution:

\[
\begin{align*}
\text{node1 @ node(r1,C)} & \implies (C = r ; C = b ; C = g). \\
\text{node2 @ node(r2,C)} & \implies (C = b ; C = g). \\
\text{node3 @ node(r3,C)} & \implies (C = r ; C = b). \\
\text{node4 @ node(r4,C)} & \implies (C = r ; C = b). \\
\text{node5 @ node(r5,C)} & \implies (C = r ; C = g). \\
\text{node6 @ node(r6,C)} & \implies (C = r ; C = g ; C = t). \\
\text{node7 @ node(r7,C)} & \implies (C = r ; C = b). \\
\text{startGraph @ edges} & \implies \text{edge(r1,r2), edge(r1,r3), edge(r1,r4), edge(r1,r7), edge(r2,r6), edge(r3,r7), edge(r4,r5), edge(r4,r7), edge(r5,r6), edge(r5,r7).} \\
\text{wrong @ edge(Ri,Rj), node(Ri,Ci), node(Rj,Cj)} & \implies Ci = Cj \mid \text{false.} \\
\text{l1 @ l([ ],[ ])} & \implies \text{true.} \\
\text{l2 @ l([R|Rs],[C|Cs])} & \implies \text{node(R,C), l(Rs,Cs).}
\end{align*}
\]

This CHR base handles a graph-coloring problem with at most 3 colors where any two nodes connected by a common edge must not have the same color. The constraints node(r1,C) means that the node r1 has color C, startGraph rule defines the edges between the nodes of a graph and wrong assures that two nodes will have different colors. A small part of the trace from the following goal "edges, l([r1,r7,r4,r3,r2,r5,r6],[C1,C7,C4,C3,C2,C5,C6])." is depicted:

\[
\begin{align*}
\text{CHR: (1) Insert: node(r1,G9234) # <384>}
\text{CHR: (2) Call: node(r1,G9234) # <384>}
\text{CHR: (2) Try: node(r1,G9234) # <384> ==> G9234=r;G9234=b;G9234=g.}
\text{CHR: (2) Apply: node(r1,G9234) # <384> ==> G9234=r;G9234=b;G9234=g.}
\ldots
\text{CHR: (2) Insert: node(r7,G9235) # <386>}
\text{CHR: (3) Call: node(r7,G9235) # <386>}
\text{CHR: (3) Try: node(r7,G9235) # <386> ==> G9235=r;G9235=b.}
\text{CHR: (3) Apply: node(r7,G9235) # <386> ==> G9235=r;G9235=b.}
\text{CHR: (4) Wake: node(r7,r) # <386>}
\text{CHR: (4) Try: node(r1,r) # <384>, edge(r1,r7) # <376>, node(r7,r) # <386> ==> r=r \mid \text{false.}}
\text{CHR: (4) Apply: node(r1,r) # <384>, edge(r1,r7) # <376>,}
\end{align*}
\]
This subset of the execution is responsible for trying the value C1 and C7 as red then backtracking because C1 and C7 cannot have the same colors.

Informal definitions of the trace events of SWI-Prolog can be found here. Some problems occur when an analysis of the trace is needed: the try/apply transition has no rule name, it’s very difficult to link the name of the generated var with the name of the variable passed as goal since all vars were renamed and there isn’t an efficient way to query it.

4.2 Understanding SWI-Prolog Trace

SWI-Prolog’s default search strategy implemented is depth-first, the parameter depth indicates the transaction’s actual level in the search tree and id is the constraint’s unique identifier. The trace output fits the following pattern:

CHR: (depth) Instruction: constraint(terms) #<ID>.

Small parts of the trace will be shown and explained.

CHR: (0) Insert: edges # <372>
CHR: (1) Call: edges # <372>

The trace produced by these two ports are responsible for removing a constraint from the goal and insert in execution stack. Notice that in SWI’s trace they always appear together.

CHR: (2) Exit: edge(r1,r2) # <373>

The computation over the active constraint is finished.

CHR: (3) Try: node(r7,_G9235) # <386> ==> _G9235=r;_G9235=b.
CHR: (3) Apply: node(r7,_G9235) # <386> ==> _G9235=r;_G9235=b.

The trace produced by Try and Apply’s port only happens together and It means that a rule was tried and applied respectively.

CHR: (4) Wake: node(r7,r) # <386>

The Wake port is traced when a built-in is solved, in this case the constraint was reactivated because C7 = r.

4.3 Transforming SWI Tracer into OS-CHR

The SWI’s output is not enough to perform a translation to OS-CHR. We do need information about what was the goal passed and access to the source-code. The inputs and outputs of the algorithm is illustrated by figure. The Translator’s algorithm will be explained by example.

Few ports have direct connection with OS-CHR:v: Call and Exit. All others ports will need a computation using the generated SWI trace. The Insert port is ignored because is redundant with the port Call.

CHR: (0) Call: edges # <372> -> [ActivateRDC,[edges,372],372]
CHR: (2) Exit: edge(r1,r2) # <373> ->[Drop,[edge,r1,r2,373],373]

For the tryRule map, we have to look the source code and try to find what is the rule name for that transition, and while generating the trace we keep track of the active constraint.

CHR: (5) Try: node(r1,r) # <384>,edge(r1,r4) # <375>, node(r4,r) # <388> ==> r=r | false. -> [TryRule, ruleName,activeConstraint, [keep,[node,r1,r,384],[edge,r1,r4,375],[node,r4,r,388]], [remove], [guard, [r=r]], 388]

ApplyRule is the most complicated map, we have to link(@) with the tryRule and check if it has a disjunctive body, if it is we have keep track to link correctly with a possible failure status; it can generate a lot of trace event depending on how many constraints were added/removed and possibly a split transition. The link function will recover the real name of the variable, in this case \( G9235 = C7 \)

CHR: (3) Apply: node(r7,\_G9235) # <386> ==> \_G9235=r;\_G9235=b.
-> [ApplyRule, '@TryRule,[addedRDCs],[removeRDCs],[bic,
[link(_G9235) = r],[link(_G9235) = b]],388] ++
[Spli,t ApplyRule,388]

For the Wake port the we have to look to previous values of the trace and determine what BIC solving fired this transition and also a Reactivate event will be produced. In this case \( C7 = r \) was the cause.

CHR: (4) Wake: node(r7,r) # <386> -> [Wake,bicSource,388]++
[Reactivate,[node,r7,r,386],@Wake,388]

The Fail port will produce a Fail event with its cause, a rule that propagates to false.

CHR: (4) Fail: node(r4,r) # <388> -> [Fail, @ApplyRule,388]

The redo port only indicates a Backtrack event, note that in SWI trace often a backtrack is done without any redo appear in the log. Here is an example:
4.4 Trace Querying

The produced generic trace is represented by a sequence of java objects. The language we choose for querying the trace is the SQL for Java Objects (JoSQL), its implementation can be found here\(^5\).

These are some examples of query in JoSQL: (on a trace of example \(^4\))

- SELECT * FROM trace WHERE type = 'ApplyRule' AND (name = 'wrong' OR name = 'node1' OR name = 'node2') Will select the trace of the execution of rules: wrong, node1, node2.

- SELECT * FROM trace WHERE type = 'Split' OR type = 'Fail' Will select all split and fail transition.

- SELECT addedRDCs, removedRDCs, addedBICs FROM trace WHERE type = 'ApplyRule'

The last query is more general and can by used by any application which need to handle a current state of the constraint store.

5 Experimentation

To evaluate our approach 3 benchmarks were set: 10-Queens, primes and a compiled example of scheduling from CHORD\(^6\), available on its test folder, the reason for choosing a CHORD example was the complexity, more than 100 rules. All results are shown in the following table.

All the experiments were performed on a PC with Pentium Core 2 Duo processor running at 2.4 GHz, with 4 GB of RAM and 1.5GB were reserved to the Java heap. The prolog process and our traces are two different process as described by Langevine\(^15\).

\(^5\) http://josql.sourceforge.net/
\(^6\) http://sourceforge.net/projects/chord/
The following queries were done:

scheduling> g []. %starts chord computation
primes> candidates(8000). %calculates primes upto 8000
10-Queens> solveall(10,N,S). % give all solutions for 10 Queens.
graphColoring> edges, 1([r1,r7,r4,r3,r2,r5,r6],[C1,C7,C4,C3,C2,C5,C6]). %graph with 7 edges

As we can observe the generation of trace events is very time consuming, but adding the feature of querying it as we parse to our definition normally add a negligible amount of time. For the selection of objects, tests executed on JoSQL show that that a list of 1,000,000 generic trace events can be queried in about 1.5s.

6 Discussion

Several aspects of such a generic trace were explored on [17], in particular its relations with component software development, the use of the fluent calculus to prototype traces and the use of object oriented specification methods. The generic trace presented in that work is thus limited to the simple theoretical operational semantics \(\omega \) [11] and therefore is less precise than the one given here.

Our approach of the observational semantics rely to abstract interpretation. The OS is similar to the “Observable Semantics” of Lucas [10] or the partial trace semantics of Cousot [3]. The parameters used to describe the execution states are, as expressed by Lucas, “syntactic objects used to represent the conduct of operational mechanisms”. The traces are abstract representations of CHR\(^\vee\) semantics which allow to take into account the sole details we want to consider as common to different implementations. The (abstraction) relations between a generic trace and the traces of specific implementations of solvers are explored in [5], together with a conformance proof method. Furthermore the generic trace contains a set of details considered as useful in several debugging tasks with several levels of refinement or observation. It could be enriched according to different needs or refined without changing the semantics of the already existing one.

This way to proceed is opposite to the frequently adopted approach as, in particular, in [18], where a set of (visual) debugging tools is defined together

\(^7\) An extensive study about the needs for constraint debugging can be found in [3].
with their input data, which consists of a restricted trace containing the minimal needed information. In our approach, we specify a semantically rich trace which can be used as input data for a potentially larger set of tools. The choice of the data to trace is made on the basis of a high level operational semantics, not on the basis of some specific debugging need. However the generic trace is designed in such a way that most of debugging tools devoted to the analysis of CHR resolution behavior may find in this trace what they need. As a consequence, based on this observational semantics, the work of implementation of the tracer and the work of designing debugging tools can be performed independently.

One may however feel that implementing a full generic trace is too much work demanding or that the resulting tracer performance will be considerably slow down. It has been shown in [15] that a generic approach may have more advantages than drawbacks in the sense that there may be a good trade-off between a very detailed generic trace (based on a more refined operational semantics) and the use of a trace driver able to query efficiently the generic trace, with a significant improvement in portability of debugging tools. We have shown here, that the implementation of the CHR\(^\lor\) generic trace in SWI-Prolog CHR implementation can easily be performed on the top of an existing tracer, resulting in an efficient generic tracer.

7 Conclusion

We have presented a first observational semantics of CHR\(^\lor\), a formal specification of a CHR\(^\lor\) generic tracer, and a first prototype based on a CHR SWI-Prolog implementation. This approach shows that the generic trace can be easily and efficiently implemented on existing CHR\(^\lor\) implementations. The interest of the “generic approach” leads in the portability of analysis tools developed on the basis of this trace and the variety of possible trace based applications.

We do not claim that the CHR\(^\lor\) observational semantics which is presented here is the ultimate one. More refined observational semantics could be considered or inclusion of several levels of refinements (for example combining with Prolog semantics in Prolog based implementations); we just have shown that this approach can be realistic and useful in a great variety of CHR based software development.

Future work will concern more experimentation and improvements of the generic trace, OO based CHR implementation including a generic trace, and generic trace for hybrid constraint solvers.

References