Conjure Revisited: Towards Automated Constraint Modelling

Ozgur Akgun¹,
Alan M. Frisch², Brahim Hnich³, Chris Jefferson¹, Ian Miguel¹

¹ School of Computer Science, University of St Andrews, UK
² Artificial Intelligence Group, Dept. of Computer Science, University of York, UK
³ Department of Computer Engineering, Izmir University of Economics, Turkey

ModRef 2010
The 9th International Workshop on Constraint Modelling and Reformulation
Introduction
- What is?
- ESSENCE
- ESSENCE by example
- ESSENCE'

Approach
- The task
- The pipeline
- Non-deterministic Rewriting
- Some rules
- Matching expressions, not constraints

Conclusion
- Coverage and limitations
- Conclusion and future work
What is?

- **ESSENCE**: a high level problem specification language
What is?

- **ESSENCE**: a high level problem specification language
- **CONJURE**: a tool to generate multiple CSP models given a problem specification
What is?

- **ESSENCE**: a high level problem specification language
- **CONJURE**: a tool to generate multiple CSP models given a problem specification
- **ESSENCE’**: a solver independent, problem class level CSP modelling language
**What is?**

- **ESSENCE**: a high level problem specification language
- **CONJURE**: a tool to generate multiple CSP models given a problem specification
- **ESSENCE'**: a solver independent, problem class level CSP modelling language
What is?

- **ESSENCE**: a high level problem specification language
- **CONJURE**: a tool to generate multiple CSP models given a problem specification
- **ESSENCE’**: a solver independent, problem class level CSP modelling language
ESSENCE

- A high level problem specification language
ESSENCE

- A high level problem specification language
- Supports many type constructors that allow problems to be specified in natural ways
  - boolean, integer, enumeration, unnamed types,
  - set, multi-set, function, relation, tuple,
  - and arbitrary nestings of these type constructors
ESSENCE

- A high level problem specification language
- Supports many type constructors that allow problems to be specified in natural ways
  - boolean, integer, enumeration, unnamed types,
  - set, multi-set, function, relation, tuple,
  - and arbitrary nestings of these type constructors
- No CSP modelling decisions involved
Problem

- given \( n \) distinct items, with associated weights and values
- select a set out of these items maximising total value
- such that the total weight is not more than that of you can carry
**ESSENCE by example**

```plaintext
given item : enum
given w : function item → int(0..)
given v : function item → int(0..)
given cap : int(0..)

find x : set of item

maximising sum i : x . v(i)
such that sum i : x . w(i) ≤ cap
```
Essence'

- Almost a subset of Essence
Essence’

- Almost a subset of Essence
- Operates on problem class level
Essence'

- Almost a subset of Essence
- Operates on problem class level
- Supports boolean and integer decision variables, and multi-dimensional matrices
**ESSENCE'**

- Almost a subset of **ESSENCE**
- Operates on problem class level
- Supports boolean and integer decision variables, and multi-dimensional matrices
- Supports several global constraints, in addition to common arithmetic and logical ones
Essence'

- Almost a subset of Essence
- Operates on problem class level
- Supports boolean and integer decision variables, and multi-dimensional matrices
- Supports several global constraints, in addition to common arithmetic and logical ones
- Tailor compiles efficient CSP models to multiple target solvers
Essence'

- Almost a subset of Essence
- Operates on problem class level
- Supports boolean and integer decision variables, and multi-dimensional matrices
- Supports several global constraints, in addition to common arithmetic and logical ones
- Tailor compiles efficient CSP models to multiple target solvers
  - Minion
  - Gecode
  - FlatZinc
### The task

- Compile **ESSENCE** specifications to multiple **ESSENCE’** models
The task

- Compile \texttt{ESSENCE} specifications to multiple \texttt{ESSENCE}' models
- Compilation process needs to be easily modifiable
The task

- Compile \texttt{ESSENCE} specifications to multiple \texttt{ESSENCE}' models
- Compilation process needs to be easily modifiable
  - A term rewriting infrastructure supported by a set of rewrite rules
The pipeline

- Parsing
The pipeline

- Parsing
- Type checking
The pipeline

- Parsing
- Type checking
- Validating the input
The pipeline

- Parsing
- Type checking
- Validating the input
- Representations phase
The pipeline

- Parsing
- Type checking
- Validating the input
- Representations phase
- Auto-Channelling phase
The pipeline

- Parsing
- Type checking
- Validating the input
- Representations phase
- Auto-Channelling phase
- Adding structural constraints
The pipeline

- Parsing
- Type checking
- Validating the input
- Representations phase
- Auto-Channelling phase
- Adding structural constraints
- Expression rewriting
The pipeline

- Parsing
- Type checking
- Validating the input
- Representations phase
- Auto-Channelling phase
- Adding structural constraints
- Expression rewriting
- Presentation
The pipeline: Representations phase

- One of the two kinds of modelling decisions
  - Selecting the viewpoint

Ozgur Akgun, et al. | CONJURE Revisited: Towards Automated Constraint Modelling
The pipeline: Representations phase

- One of the two kinds of modelling decisions
  - Selecting the viewpoint
- Select a representation for every parameter and decision variable
The pipeline: Representations phase

- One of the two kinds of modelling decisions
  - Selecting the viewpoint
- Select a representation for every parameter and decision variable
- Possible to represent a variable in multiple ways
  - if it appears in more than one constraint
2 representations for sets

- Given: set (size 2) of int(1..3)
2 representations for sets

- Given: set (size 2) of int(1..3)
  - Explicit representation:
    matrix indexed by [int(1..2)] of int(1..3)
2 representations for sets

- Given: set (size 2) of int(1..3)
  - Explicit representation:
    matrix indexed by [int(1..2)] of int(1..3)
  - Occurrence representation:
    matrix indexed by [int(1..3)] of bool
Example

\textbf{given } lb, ub, n: \texttt{int} \\
\textbf{given } s: \texttt{set of int}(lb..ub) \\
\textbf{find } x: \texttt{set (size n) of int}(lb..ub) \\

\textbf{such that} \\
\quad x \texttt{ subseteq } s, \\
\quad \texttt{forall } i: x. k(i)
Example

\begin{verbatim}
given lb, ub, n: int
given s: set of int(lb..ub)
find x_occurrences: set (size n) of int(lb..ub)

such that
  x_occurrences \subseteq s,
  forall i: x_occurrences . k(i)
\end{verbatim}
Example

\textbf{given} \ lb, ub, n: \texttt{int} \\
\textbf{given} \ s: \texttt{set of int}(lb..ub) \\
\textbf{find} \ x_{\text{expl}}: \texttt{set (size n) of int}(lb..ub) \\

\textbf{such that} \\
\quad x_{\text{expl}} \ \texttt{subseteqq} \ s, \\
\quad \textbf{forall} \ i : x_{\text{expl}} \ . \ k(i)
Example

\[
\text{given } \texttt{lb, ub, n: int} \\
\text{given } \texttt{s: set of int(lb..ub)} \\
\text{find } \texttt{x?: set (size n) of int(lb..ub)} \\
\]

\[
\text{such that} \quad \texttt{x\_occr subseteq s}, \\
\text{forall } i : \texttt{x\_expl . k(i)}
\]
Example

\begin{verbatim}
given lb, ub, n : int
given s : set of int(lb..ub)
find x_expl : set (size n) of int(lb..ub)
find x_occ : set (size n) of int(lb..ub)

such that
  x_occ \subseteq s,
  \forall i : x_expl . k(i)
\end{verbatim}
Example

given \( lb, ub, n : \text{int} \)
given \( s : \text{set of int}(lb..ub) \)
find \( x\_\text{expl} : \text{set (size n) of int}(lb..ub) \)
find \( x\_\text{occr} : \text{set (size n) of int}(lb..ub) \)

such that
\[
    x\_\text{occr} \subseteqeq s, \\
    \text{forall } i : x\_\text{expl} . \ k(i), \\
    x\_\text{occr} = x\_\text{expl}
\]
The pipeline: Auto-Channelling phase

More than one representation for a decision variable

=>

pairwise equality constraints!
The pipeline: Adding structural constraints

- Now, representations for decision variables are known
The pipeline: Adding structural constraints

- Now, representations for decision variables are known
- “Structural constraints” are added to the model
The pipeline: Adding structural constraints

- Now, representations for decision variables are known
- “Structural constraints” are added to the model
  - an alldiff constraint for \(x_{\text{expl}}\)
  - a cardinality constraint for \(x_{\text{occr}}\)
Example, structural constraints added

```plaintext
given lb, ub, n: int
given s: set of int(lb..ub)
find x_expl: set (size n) of int(lb..ub)
find x_occr: set (size n) of int(lb..ub)

such that
  x_occr subseteq s,
  forall i : x_expl . k(i),
  x_occr = x_expl,
  { alldiff on x_expl's refinement },
  { cardinality on x_occr's refinement }
```

Example, final

```plaintext
given lb, ub, n: int
given s_occrr: matrix indexed by [int(lb..ub)] of bool
find x_expl: matrix indexed by [int(1..n)] of int(lb..ub)
find x_occrr: matrix indexed by [int(lb..ub)] of bool

such that
forall i: int(lb..ub). x_occrr[i] <= s_occrr[i],
forall i: int(1..n). k(x_expl[i]),
forall i: int(1..n). x_occrr[x_expl[i]] = 1,
forall i: int(lb..ub). ( x_occrr[i] => exists j: int(1..n). x_expl[j] = i ),
all diff(x_expl),
sum i: int(lb..ub). x_occrr[i] = n
```

Non-deterministic Rewriting

- Given a set of rewrite rules and a starting term, apply the rules repeatedly.
Non-deterministic Rewriting

- Given a set of rewrite rules and a starting term, apply the rules repeatedly.
  - normal form
Non-deterministic Rewriting

- Given a set of rewrite rules and a starting term, apply the rules repeatedly.
  - *normal form*
- More than one rule can match a term.
Non-deterministic Rewriting

- Given a set of rewrite rules and a starting term, apply the rules repeatedly.
  - normal form
- More than one rule can match a term.
  - Select one at random?
Non-deterministic Rewriting

- Given a set of rewrite rules and a starting term, apply the rules repeatedly.
  - normal form
- More than one rule can match a term.
  - Select one at random?
  - Apply all matching rules? (produces a list of terms)
Rule representation

- Rules are represented as *directed equations* with *guards*. 

---

Ozgur Akgun, *et al.*  |  **CONJURE** Revisited: Towards Automated Constraint Modelling
Rule representation

- Rules are represented as *directed equations with guards*.

\[
\begin{align*}
A \times B & \rightarrow A + A, \text{ if } B \text{ is } 2 \\
A \div B & \rightarrow A, \text{ if } B \text{ is } 1
\end{align*}
\]
Rule representation

- Rules are represented as *directed equations with guards.*
  
  \[
  A \times B \rightarrow A + A, \text{ if } B \text{ is } 2 \\
  A \div B \rightarrow A, \text{ if } B \text{ is } 1
  \]

- This is a partial mapping.
Rule representation

- Rules are represented as *directed equations with guards*.
  
  \[
  A \times B \rightarrow A + A, \text{ if } B \text{ is } 2
  
  A \div B \rightarrow A, \text{ if } B \text{ is } 1
  \]

- This is a partial mapping.

- Can combine multiple such rules, into a one-to-many mapping.

---

Ozgur Akgun, et al.

**CONJURE Revisited: Towards Automated Constraint Modelling**
Rule representation

- Rules are represented as *directed equations with guards*.
  
  \[
  A \times B \rightarrow A + A, \text{ if } B \text{ is } 2 \\
  A / B \rightarrow A, \text{ if } B \text{ is } 1
  \]

- This is a partial mapping.

- Can combine multiple such rules, into a one-to-many mapping.

Subterms: \{A,B,C,D\}

| rule1: A to B |
| rule2: A to C |
| rule3: B to D |

allRules
\{ (A, \{B,C\}), (B, \{D\}), (C, \{C\}), (D, \{D\}) \}
Rule representation

- Rules are represented as *directed equations with guards*.
  - \( A \times B \rightarrow A + A, \) if \( B \) is 2
  - \( A / B \rightarrow A \), if \( B \) is 1

- This is a partial mapping.

- Can combine multiple such rules, into a one-to-many mapping.

- Handle just one rule.
Some example rules

\[
\text{essence\_expression} \rightsquigarrow \text{equivalent\_expression}
\]

- **guards:** *properties that essence\_expression must satisfy*
- **declarations:** *newly created variables and local aliases for expressions*
Example rules: \texttt{ruleSetEq}

\[
a = b \Rightarrow a \subseteq b \land b \subseteq a \\
guards: \ a \sim \ \text{set of } \tau, \\
b \sim \ \text{set of } \tau
\]
Example rules: `ruleSetSubsetEq`

\[ a \subseteq b \Rightarrow \text{forall } i : a . i \in b \]

guards: \[ a \sim \text{set of } \tau, \]
\[ b \sim \text{set of } \tau \]
Example rules: ruleSetElem

\[ e \text{ elem } s \leadsto \exists i : s . i = e \]

guards: \[ e \sim \tau, \quad s \sim \text{set of } \tau \]
Example rules: \texttt{ruleSetQuan}

\[
\begin{align*}
\text{forall } i: (a \cup b).k \rightsquigarrow & \text{forall } i: a.k \land \text{forall } i: b.k \\
\text{exists } i: (a \cup b).k \rightsquigarrow & \text{exists } i: a.k \lor \text{exists } i: b.k \\
\text{forall } i: (a \cap b).k \rightsquigarrow & \text{forall } i: a.(i \in b \Rightarrow k) \\
\text{forall } i: (a \cap b).k \rightsquigarrow & \text{forall } i: b.(i \in a \Rightarrow k) \\
\text{exists } i: (a \cap b).k \rightsquigarrow & \text{exists } i: a.(i \in b \land k) \\
\text{exists } i: (a \cap b).k \rightsquigarrow & \text{exists } i: b.(i \in a \land k)
\end{align*}
\]
Matching expressions, not constraints

- Some other tools reason about complete constraints
Matching expressions, not constraints

- Some other tools reason about complete constraints
  - including previous implementations of Conjure
Matching expressions, not constraints

- Some other tools reason about complete constraints
  - including previous implementations of Conjure
- Now, can match with any subexpression, not necessarily complete constraints
Matching expressions, not constraints

- Some other tools reason about complete constraints
  - including previous implementations of Conjure
- Now, can match with any subexpression, not necessarily complete constraints
  - Can work on a non-flat model
Matching expressions, not constraints

- Some other tools reason about complete constraints
  - including previous implementations of Conjure
- Now, can match with any subexpression, not necessarily complete constraints
  - Can work on a non-flat model
  - Able to reason about structure
Matching expressions, not constraints

- Some other tools reason about complete constraints
  - including previous implementations of **Conjure**
- Now, can match with any subexpression, not necessarily complete constraints
  - Can work on a non-flat model
  - Able to reason about structure
  - Do more with fewer rules
Matching expressions, not constraints

- Consider: \((a \cup b) \subseteq c\)
Matching expressions, not constraints

Consider: \[(a \cup b) \subseteq c\]
Flattened: \[\text{aux} \subseteq c \land \text{aux} = a \cup b\]
Matching expressions, not constraints

- Consider: \((a \cup b) \subseteq c\)
- Flattened: \(aux \subseteq c \land aux = a \cup b\)
- We could have: \(a \subseteq c \land b \subseteq c\)
Matching expressions, not constraints

- Flattening = lost information
Matching expressions, not constraints

- Flattening = lost information
- We can still *flatten* things, but only if we want, using our powerful rewrite rules!
Matching expressions, not constraints

- Flattening = lost information
- We can still flatten things, but only if we want, using our powerful rewrite rules!
- A small problem, where to put helper constraints?
Matching expressions, not constraints

\[
\begin{align*}
given & \ \ lb, ub, n, m, k : \text{int} \\
\text{find} & \ \ t : \text{set (size n) of int(lb..ub)} \\
\text{find} & \ \ A : \text{set (size n) set (size m) of int(lb..ub)} \\
\text{such that} \ & \\
\quad \text{forall} & \ \ s : A . \\
\quad \quad \quad & (\max(s) - \max(t) = k) \implies (k \ \text{elem} \ s)
\end{align*}
\]
@: the *bubble* attaching operator

\[
\text{max}(s) \rightsquigarrow \text{max}_s \circ \text{bubble} \\
\text{guards: } s \sim \text{set of int} \\
\text{declarations: } \text{max}_s : \text{int} \\
\quad \text{bubble} = (\text{max}_s \text{ elem } s) \land (\forall i : s . i \leq \text{max}_s)
\]
Matching expressions, not constraints

- \( \text{forall } s: A . \)
  \( (\text{max}(s) - \text{max}(t) = k) \Rightarrow (k \text{ elem } s) \)
Matching expressions, not constraints

- For all \( s : A \):
  \[
  (\text{max}(s) - \text{max}(t) = k) \Rightarrow (k \text{ elem } s)
  \]

- For all \( s : A \):
  \[
  ((\text{max}_s @ \text{bubble}_s) - \text{max}(t) = k) \Rightarrow (k \text{ elem } s)
  \]
Matching expressions, not constraints

1. For all $s \in A$.
   $$(\text{max}(s) - \text{max}(t) = k) \Rightarrow (k \text{ elem } s)$$

2. For all $s \in A$.
   $$((\text{max}_s@\text{bubble}_s) - \text{max}(t) = k) \Rightarrow (k \text{ elem } s)$$

3. For all $s \in A$.
   $$((\text{max}_s@\text{bubble}_s) - (\text{max}_t@\text{bubble}_t) = k) \Rightarrow (k \text{ elem } s)$$
Matching expressions, not constraints

- \( \forall s: A . \) \((\text{max}(s) - \text{max}(t) = k) \Rightarrow (k \in s)\)
- \( \forall s: A . \) \(((\text{max}_s@\text{bubble}_s) - \text{max}(t) = k) \Rightarrow (k \in s)\)
- \( \forall s: A . \) \(((\text{max}_s@\text{bubble}_s) - (\text{max}_t@\text{bubble}_t) = k) \Rightarrow (k \in s)\)
- \( \forall s: A . \) \(((\text{max}_s-\text{max}_t) @ (\text{bubble}_s \setminus \text{bubble}_t))=k) \Rightarrow (k \in s)\)
Matching expressions, not constraints

- $\forall s: A \ . \ (\max(s) - \max(t) = k) \Rightarrow (k \text{ elem } s)$
- $\forall s: A \ . \ ((\max_s@\text{bubble}_s) - \max(t) = k) \Rightarrow (k \text{ elem } s)$
- $\forall s: A \ . \ ((\max_s@\text{bubble}_s) - (\max_t@\text{bubble}_t) = k) \Rightarrow (k \text{ elem } s)$
- $\forall s: A \ . \ (((\max_s-\max_t) @ (\text{bubble}_s \setminus \text{bubble}_t))=k) \Rightarrow (k \text{ elem } s)$
- $\forall s: A \ . \ (((\max_s-\max_t=k) @ (\text{bubble}_s \setminus \text{bubble}_t))) \Rightarrow (k \text{ elem } s)$
Matching expressions, not constraints

- $\forall s : A \ . \ (\max(s) - \max(t) = k) \Rightarrow (k \in s)$
- $\forall s : A \ . \ ((\max_s@\text{bubble}_s) - \max(t) = k) \Rightarrow (k \in s)$
- $\forall s : A \ . \ ((\max_s@\text{bubble}_s) - (\max_t@\text{bubble}_t) = k) \Rightarrow (k \in s)$
- $\forall s : A \ . \ (((\max_s - \max_t) \land (\text{bubble}_s \land \text{bubble}_t)) = k) \Rightarrow (k \in s)$
- $\forall s : A \ . \ (((\max_s - \max_t = k) \land (\text{bubble}_s \land \text{bubble}_t))) \Rightarrow (k \in s)$
- $\forall s : A \ . \ (((\max_s - \max_t = k) \Rightarrow (k \in s)) \land \text{bubble}_t \land \forall s : A \ . \ (((\max_s - \max_t = k) \Rightarrow (k \in s)) \land \text{bubble}_s)$
Matching expressions, not constraints

- $\forall s: A . \quad (\max(s) - \max(t) = k) \Rightarrow (k \in s)$
- $\forall s: A . \quad ((\max_s@\text{bubble}_s) - \max(t) = k) \Rightarrow (k \in s)$
- $\forall s: A . \quad ((\max_s@\text{bubble}_s) - (\max_t@\text{bubble}_t) = k) \Rightarrow (k \in s)$
- $\forall s: A . \quad (((\max_s-\max_t) @ (\text{bubble}_s \cap \text{bubble}_t)) = k) \Rightarrow (k \in s)$
- $\forall s: A . \quad (((\max_s-\max_t=k) @ (\text{bubble}_s \cap \text{bubble}_t)) \Rightarrow (k \in s)$
Matching expressions, not constraints

- \( \forall s : A . \)
  \[ (\text{max}(s) - \text{max}(t) = k) \Rightarrow (k \in s) \]
- \( \forall s : A . \)
  \[ ((\text{max}_s@\text{bubble}_s) - \text{max}(t) = k) \Rightarrow (k \in s) \]
- \( \forall s : A . \)
  \[ ((\text{max}_s@\text{bubble}_s) - (\text{max}_t@\text{bubble}_t) = k) \Rightarrow (k \in s) \]
- \( \forall s : A . \)
  \[ (((\text{max}_s-\text{max}_t) @ (\text{bubble}_s /\text{bubble}_t))=k) \Rightarrow (k \in s) \]
- \( \forall s : A . \)
  \[ (((\text{max}_s-\text{max}_t=k) @ (\text{bubble}_s /\text{bubble}_t))) \Rightarrow (k \in s) \]
- \( \forall s : A . \)
  \[ (((\text{max}_s-\text{max}_t=k) => (k \in s)) @ (\text{bubble}_s /\text{bubble}_t)) \]
- \( \forall s : A . \)
  \[ (((\text{max}_s-\text{max}_t=k) => (k \in s)) /\text{bubble}_s /\text{bubble}_t) \]
- \( \text{bubble}_t /\forall s : A . \)
  \[ (((\text{max}_s-\text{max}_t=k) => (k \in s)) /\text{bubble}_s) \]
Coverage of **Essence**

- **Essence** has 7 core type constructors
  - matrix, set, mset, partition, tuple, function, relation
Coverage of **ESSENCE**

- **ESSENCE** has 7 core type constructors
  - matrix, set, mset, partition, tuple, function, relation
- Type constructors supported: all except partition
Coverage of **ESSENCE**

- **ESSENCE** has 7 core type constructors
  - matrix, set, mset, partition, tuple, function, relation
- Type constructors supported: all except partition
- Also we haven’t yet implemented support for enumerated and unnamed types
Coverage of **ESSENCE**

- **ESSENCE** has 7 core type constructors
  - matrix, set, mset, partition, tuple, function, relation
- Type constructors supported: all except partition
- Also we haven’t yet implemented support for enumerated and unnamed types
- There are nearly 30 operators defined on these type constructors
Coverage of **ESSENCE**

- **ESSENCE** has 7 core type constructors
  - matrix, set, mset, partition, tuple, function, relation
- Type constructors supported: all except partition
- Also we haven’t yet implemented support for enumerated and unnamed types
- There are nearly 30 operators defined on these type constructors
- Almost all of them implemented
Differences to the prototype implementation

- Broader coverage of ESSENCE
Differences to the prototype implementation

- Broader coverage of ESSENCE
- Representation decisions
Differences to the prototype implementation

- Broader coverage of ESSENCE
- Representation decisions
- Auto-channelling becomes very easy
Differences to the prototype implementation

- Broader coverage of ESSENCE
- Representation decisions
- Auto-channelling becomes very easy
- No flattening
Differences to the prototype implementation

- Broader coverage of **ESSENCE**
- Representation decisions
- Auto-channelling becomes very easy
- No flattening
- Easier rule authoring
Conclusion and future work

- New version of CONJURE with far greater coverage of the ESSENCE language
Conclusion and future work

- New version of *Conjure* with far greater coverage of the *Essence* language
- Immediate future work, covering all of the types and operations in *Essence*
Conclusion and future work

- New version of Conjure with far greater coverage of the Essence language
- Immediate future work, covering all of the types and operations in Essence
- Capture best modelling practices
Conclusion and future work

- New version of Conjure with far greater coverage of the Essence language
- Immediate future work, covering all of the types and operations in Essence
- Capture best modelling practices
- Model selection
Conclusion and future work

- New version of CONJURE with far greater coverage of the ESSENCE language
- Immediate future work, covering all of the types and operations in ESSENCE
- Capture best modelling practices
- Model selection
- Investigate multi-model search techniques