Conjure Revisited: Towards Automated Constraint Modelling

Ozgur Akgun[1],
Alan M. Frisch[2], Brahim Hnich[3], Chris Jefferson[1], Ian Miguel[1]

[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

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

ESSENCE by example

- 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

```
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 ESSENCE specifications to multiple ESSENCE′ models
- Compilation process needs to be easily modifiable

The task

- Compile ESSENCE specifications to multiple 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*

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

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

such that
    x subseteq s,
    forall i : x . k(i)
```

Example

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

such that
    x_occr subseteq s,
    forall i : x_occr . k(i)
```

Example

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

such that
    x_expl subseteq s,
    forall i : x_expl . k(i)
```

Example

```
given lb,ub,n: int
given s: set of int(lb..ub)
find  x_?: set (size n) of int(lb..ub)

such that
    x_occr subseteq s,
    forall i : x_expl . k(i)
```

Example

```
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)
```

Example

```
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
```

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_expl`
    - a cardinality constraint for `x_occr`

Example, structural constraints added

```
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

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

such that
    forall i : int(lb..ub) . x_occr[i] <= s_occr[i],
    forall i : int(1..n)   . k(x_expl[i]),
    forall i : int(1..n)   . x_occr[x_expl[i]] = 1,
    forall i : int(lb..ub) . (
        x_occr[i] => exists j : int(1..n) . x_expl[j] = i
    ),
    alldiff(x_expl),
   sum i : int(lb..ub) . x_occr[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*.

Rule representation

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

```
A * B -> A + A, if B is 2
A / B -> A    , if B is 1
```

Rule representation

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

```
A * B -> A + A, if B is 2
A / B -> A    , if B is 1
```

- This is a partial mapping.

Rule representation

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

```
A * B -> A + A, if B is 2
A / B -> A    , if B is 1
```

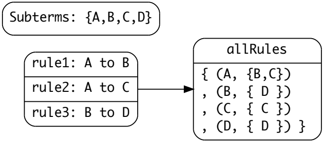
- This is a partial mapping.
- Can combine multiple such rules, into a one-to-many mapping.

Rule representation

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

```
A * B -> A + A, if B is 2
A / B -> A    , if B is 1
```

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



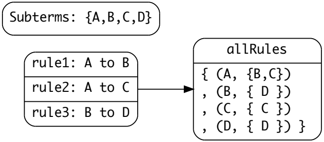
```
Subterms: {A,B,C,D}

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 * B -> A + A, if B is 2
A / B -> A    , if B 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 }) }
```

- Handle just one rule.

Some example rules

```
essence_expression ⇝ equivalent_expression
    guards: properties that essence_expression must satisfy
    declarations: newly created variables and
                  local aliases for expressions
```

Example rules: `ruleSetEq`

```
a = b ⇝ a subseteq b /\ b subseteq a
    guards: a ∼ set of τ,
            b ∼ set of τ
```

Example rules: `ruleSetSubsetEq`

```
a subseteq b ⇝ forall i : a . i elem b
    guards: a ∼ set of τ,
            b ∼ set of τ
```

Example rules: `ruleSetElem`

```
e elem s ⇝ exists i : s . i = e
    guards: e ∼ τ,
            s ∼ set of τ
```

Example rules: `ruleSetQuan`

```
forall i : (a union b) . k ⇝ forall i : a . k /\ forall i : b . k
exists i : (a union b) . k ⇝ exists i : a . k \/ exists i : b . k
forall i : (a intersect b) . k ⇝ forall i : a ( i elem b => k )
forall i : (a intersect b) . k ⇝ forall i : b ( i elem a => k )
exists i : (a intersect b) . k ⇝ exists i : a ( i elem b /\ k )
exists i : (a intersect b) . k ⇝ exists i : b ( i elem a /\ k )
```

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 union b) subseteq c`

Matching expressions, not constraints

- Consider: `(a union b) subseteq c`
- Flattened: `aux subseteq c /\ aux = a union b`

Matching expressions, not constraints

- Consider: `(a union b) subseteq c`
- Flattened: `aux subseteq c /\ aux = a union b`
- We could have: `a subseteq c /\ 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

```
given lb,ub,n,m,k : int
find t : set (size n) of int(lb..ub)
find A : set (size n) set (size m) of int(lb..ub)

such that
    forall s : A .
        (max(s) - max(t) = k) => (k elem s)
```

`@`: the *bubble* attaching operator

```
max(s) ⇝ max_s @ bubble
    guards: s ∼ set of int
    declarations: max_s  : int
                  bubble = (max_s elem s) /\ (forall i : s . i <= max_s)
```

Matching expressions, not constraints

-
  ```
  forall s: A .
      (max(s) - max(t) = k) => (k elem s)
  ```

Matching expressions, not constraints

- `forall s: A .`
  ```
      (max(s) - max(t) = k) => (k elem s)
  ```
- `forall s: A .`
  ```
      ((max_s@bubble_s) - max(t) = k) => (k elem s)
  ```

Matching expressions, not constraints

-
  ```
  forall s: A .
      (max(s) - max(t) = k) => (k elem s)
  ```

-
  ```
  forall s: A .
      ((max_s@bubble_s) - max(t) = k) => (k elem s)
  ```

-
  ```
  forall s: A .
      ((max_s@bubble_s) - (max_t@bubble_t) = k) => (k elem s)
  ```

Matching expressions, not constraints

- `forall s: A .`
  `(max(s) - max(t) = k) => (k elem s)`
- `forall s: A .`
  `((max_s@bubble_s) - max(t) = k) => (k elem s)`
- `forall s: A .`
  `((max_s@bubble_s) - (max_t@bubble_t) = k) => (k elem s)`
- `forall s: A .`
  `(((max_s-max_t) @ (bubble_s /\ bubble_t))=k) => (k elem s)`

Matching expressions, not constraints

-
  ```
  forall s: A .
      (max(s) - max(t) = k) => (k elem s)
  ```

-
  ```
  forall s: A .
      ((max_s@bubble_s) - max(t) = k) => (k elem s)
  ```

-
  ```
  forall s: A .
      ((max_s@bubble_s) - (max_t@bubble_t) = k) => (k elem s)
  ```

-
  ```
  forall s: A .
      (((max_s-max_t) @ (bubble_s /\ bubble_t))=k) => (k elem s)
  ```

-
  ```
  forall s: A .
      (((max_s-max_t=k) @ (bubble_s /\ bubble_t))) => (k elem s)
  ```

Matching expressions, not constraints

- `forall s: A .`
  ```
  (max(s) - max(t) = k) => (k elem s)
  ```
- `forall s: A .`
  ```
  ((max_s@bubble_s) - max(t) = k) => (k elem s)
  ```
- `forall s: A .`
  ```
  ((max_s@bubble_s) - (max_t@bubble_t) = k) => (k elem s)
  ```
- `forall s: A .`
  ```
  (((max_s-max_t) @ (bubble_s /\ bubble_t))=k) => (k elem s)
  ```
- `forall s: A .`
  ```
  (((max_s-max_t=k) @ (bubble_s /\ bubble_t))) => (k elem s)
  ```
- `forall s: A .`
  ```
  (((max_s-max_t=k) => (k elem s)) @ (bubble_s /\ bubble_t))
  ```

Matching expressions, not constraints

-
  ```
  forall s: A .
       (max(s) - max(t) = k) => (k elem s)
  ```

-
  ```
  forall s: A .
       ((max_s@bubble_s) - max(t) = k) => (k elem s)
  ```

-
  ```
  forall s: A .
       ((max_s@bubble_s) - (max_t@bubble_t) = k) => (k elem s)
  ```

-
  ```
  forall s: A .
       (((max_s-max_t) @ (bubble_s /\ bubble_t))=k) => (k elem s)
  ```

-
  ```
  forall s: A .
       (((max_s-max_t=k) @ (bubble_s /\ bubble_t))) => (k elem s)
  ```

-
  ```
  forall s: A .
       (((max_s-max_t=k) => (k elem s)) @ (bubble_s /\ bubble_t))
  ```

-
  ```
  forall s: A .
       (((max_s-max_t=k) => (k elem s)) /\ bubble_s /\ bubble_t)
  ```

Matching expressions, not constraints

-
  ```
  forall s: A .
      (max(s) - max(t) = k) => (k elem s)
  ```

-
  ```
  forall s: A .
      ((max_s@bubble_s) - max(t) = k) => (k elem s)
  ```

-
  ```
  forall s: A .
      ((max_s@bubble_s) - (max_t@bubble_t) = k) => (k elem s)
  ```

-
  ```
  forall s: A .
      (((max_s-max_t) @ (bubble_s /\ bubble_t))=k) => (k elem s)
  ```

-
  ```
  forall s: A .
      (((max_s-max_t=k) @ (bubble_s /\ bubble_t))) => (k elem s)
  ```

-
  ```
  forall s: A .
      (((max_s-max_t=k) => (k elem s)) @ (bubble_s /\ bubble_t))
  ```

-
  ```
  forall s: A .
      (((max_s-max_t=k) => (k elem s)) /\ bubble_s /\ bubble_t)
  ```

-
  ```
  bubble_t /\ forall s: A .
      (((max_s-max_t=k) => (k elem s)) /\ 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