reqT.org – A Scala DSL for Constraint-based Requirement Engineering using JaCoP

Björn Regnell

Lund University
Sweden

The 12th Workshop of the Network of Sweden-based researchers and practitioners of Constraint programming, 2013
Sub-disciplines of Requirements Engineering (RE):

- **Elicitation**: generating candidate reqts and context knowledge
- **Specification**: documenting candidate reqts
- **Validation**: checking that the (documented) reqts are good enough
- **Prioritization**: assessing candidate reqts based benefit, cost, risk, urgency, ...
- **Selection**: deciding which reqts to implement when, under constraints of estimated stakeholder priorities, return-on-investment, inter-dependencies, resource constraints, timing issues, ...
Release Planning in Software Development

[ Ruhe et al.]

Björn Regnell, Lund University

reqT.org – Constraint-based Requirement Engineering
Why Constraint Solving in Requirements Engineering?

Some potential benefits of CSP in RE:

- Flexible specification of decision problems
  - Prioritization
  - Release Planning
- Interactive exploration of the solution space
- Out-of-the-box optimization support

Some challenges:

- How to integrate CSP with RE technology and make it user friendly in the domain?
- How to model CSP problems at the right abstraction level given great uncertainties?
A reqT model includes a sequence of graph parts <Entity> <Edge> <NodeSet> separated by comma and wrapped inside a Model( )

```scala
var myRequirements = Model(
  Feature("f1") has (Spec("A good spec."), Status(SPECIFIED)),
  Feature("f1") requires (Feature("f2"), Feature("f3")),
  Stakeholder("s1") assigns(Prio(1)) to Feature("f2")
)
```

Download: http://reqT.org
Source code: https://github.com/reqT/reqT
reqT models are graph structures with Entities & Attributes (nodes) and Relations (edges)

```rust
var myRequirements = Model(
    Feature("f1") has (Spec("A good spec."), Status(SPECIFIED)),
    Feature("f1") requires (Feature("f2"), Feature("f3")),
    Stakeholder("s1") assigns(Prio(1)) to Feature("f2")
)
```
Overview of the reqT metamodel
reqT models can be hierarchical with recursive submodels in a tree structure

```javascript
var myReqs = Model(
    Feature("nice") has Spec("this is a nice feature"),
    Feature("cool") has Spec("this is a cool feature"),
    Stakeholder("Anna") has Submodel(
        Feature("nice") has Prio(1),
        Feature("cool") has Prio(2)
    ),
    Stakeholder("Martin") has Submodel(
        Feature("nice") has Prio(2),
        Feature("cool") has Prio(1)
    )
)
```
reqT can reference values of attribute in deeply nested submodel structures using the ! operator

Feature("f")!Prio == Ref[Int](Vector(Feature(f)),Prio)
Stakeholder("a")!Feature("g")!Benefit ==
  Ref[Int](Vector(Stakeholder(a),Feature(g)),Benefit)

val m = Model(
  Feature("f") has Prio(1),
  Stakeholder("a") has Submodel(
    Feature("g") has Benefit(2),
    Resource("x") has Submodel(
      Feature("h") has Cost(3)
    )
  )
)

m(Feature("f")!Prio) == 1
m(Stakeholder("a")!Feature("g")!Benefit) == 2
m(Stakeholder("a")!Resource("x")!Feature("h")!Cost) == 3
reqT includes a Scala-embedded DSL for CSP that wraps the JaCoP solver

- The DSL uses Scala immutable case classes
- The search is set up using JaCoP when calling solve
- The search is controlled by parameters to solve
- Search results can be accessed through a
  scala.collection.immutable.Map

result.lastSolution(Var("x"))
Constraint-based **Priority Ranking** example:
5 features ranked from 1 to 5

reqT:

```plaintext
val n = 5
var f = vars(n, "f")
val Result(conclusion, nSol, sol, , , ) =
  Constraints(
    f::[0 until n],
    AllDifferent(f),
    f(0) #> f(1),
    f(1) #> f(2),
    f(2) #< f(3),
    forall(0 until n) { f(4) #>= f(_) }
  ).solve(Satisfy)
```

MiniZinc:

```plaintext
int: n = 5;
array[1..n] of var 1..n: f;
constraint
alldifferent(f);
constraint f[1] > f[2];
constraint f[2] > f[3];
constraint f[3] < f[4];
constraint
  forall ( i in 1..n)
    ( f[5] >= f[i] );
solve satisfy;
```
reqT includes a Scala-embedded DSL for constraints over integer variables

Some key parts of the implementation in Scala:

\[ \text{Var("x")} \mathbin{\#==} \text{Var("y")} \quad //\quad \text{req.XeqY[String]} = \text{XeqY(Var(x),Var(y))} \]

case class Var[+T](ref: T)

implicit def refToVar[T](r: Ref[T]): Var[Ref[T]] = Var(r)

case class Interval(min: Int, max: Int)

implicit def rangeToInterval(r: Range): Interval = Interval(r.min, r.max)

def vars[T <: AnyRef](vs: T *): Vector[Var[T]] = vs.map(Var(_)).toVector

def vars(n: Int, prefix: String): Vector[Var[String]] =
  (for (i <- 0 until n) yield Var(s"$prefix$i"')).toVector

def forAll[T](xs: Seq[T])(f: T => Constr[_]) = And(xs.map(f(_)).toVector)
def solve[T](
    objective: Objective = Satisfy,
    timeOutOption: Option[Long] = None,
    solutionLimitOption: Option[Int] = None,
    valueSelection: ValueSelection = jacop.IndomainRandom,
    variableSelection: VariableSelection = jacop.InputOrder,
    assignOption: Option[Seq[Var[T]]] = None
): Result[T]

sealed trait Objective
case object Satisfy extends Objective
case object CountAll extends Objective
case object FindAll extends Objective
sealed trait Optimize[+T] extends Objective {
    def cost: Var[T]
}
  case class Minimize[+T](cost: Var[T]) extends Optimize[T]
  case class Maximize[+T](cost: Var[T]) extends Optimize[T]

Björn Regnell, Lund University
reqT.org – Constraint-based Requirement Engineering
case class Result[T](
  conclusion: Conclusion,
  solutionCount: Int = 0,
  lastSolution: Map[Var[T], Int] = Map[Var[T], Int](),
  interruptOption: Option[SearchInterrupt] = None,
  solutionsOption: Option[Solutions[T]] = None
)

sealed trait Conclusion
case object SolutionFound extends Conclusion
case object SolutionNotFound extends Conclusion
case object InconsistencyFound extends Conclusion
case class SearchFailed(msg: String) extends Conclusion

class Solutions[T](
  //the only non-case class (jacop mutability propagates)
  val jacopDomains: Array[Array[JaCoP.core.Domain]],
  val jacopVariables: Array[_ <: JaCoP.core.Var],
  val nSolutions: Int,
  val lastSolution: Map[Var[T], Int]) {
  def solutionMap(s: Int): Map[Var[T], Int] = ...
  def printSolutions: Unit = ...
  ...
}

Entities can have a Constraints attribute containing a sequence of constraints.

```javascript
var myReqs = Model(
    Feature("nice") has Spec("this is a nice feature"),
    Feature("cool") has Spec("this is a cool feature"),
    Stakeholder("Anna") has Constraints(
        (Feature("nice")!Prio) #< 10,
        (Feature("nice")!Prio) #>= 1,
        (Feature("cool")!Prio)::{2 to 7}
    ),
    Stakeholder("Martin") has Constraints(
        (Feature("nice")!Prio) #< 3,
        (Feature("nice")!Prio) #!= 1,
        (Feature("cool")!Prio)::{5 to 10}
    )
);
```
val m = Model(
  Stakeholder("kalle") has (Prio(10), Submodel(
    Feature("F1") has Benefit(20),
    Feature("F2") has Benefit(20),
    Feature("F3") has Benefit(20)
  )),
  Stakeholder("stina") has (Prio(20), Submodel(
    Feature("F1") has Benefit(5),
    Feature("F2") has Benefit(15),
    Feature("F3") has Benefit(35)
  )),
  Resource("developer") has Submodel(
    Release("a") has Capacity(100),
    Release("b") has Capacity(100),
    Feature("F1") has Cost(10),
    Feature("F2") has Cost(70),
    Feature("F3") has Cost(20)
  ),
  Resource("tester") has Submodel(
    Release("a") has Capacity(100),
    Release("b") has Capacity(100),
    Feature("F1") has Cost(40),
    Feature("F2") has Cost(10),
    Feature("F3") has Cost(50)
  ),
  Release("a") has Order(1),
  Release("b") has Order(2)
)
reqT Release Planning: Vectors of Input Entities to prepare imposed constraints

```scala
val features = (m.flatten / Feature).sourceVector
val releases = (m / Release).sourceVector
val resources = (m / Resource).sourceVector
val stakeholders = (m / Stakeholder).sourceVector

val constraints = ??? // to be defined
val utility = ??? // to be defined

val (m2, r) = Model().impose(constraints).solve(Maximize(utility))
```
The XeqC case class constraint, that can be constructed by the \( \#== \) infix operator on \( \text{Var} \), is used to make a sequence of constraints that grounds integer variables to release planning input data from a reqT Model.

```
def assignValuesFromModel(m: Model) =
    Constraints(
        stakeholders.map(s => Var(s!Prio) \#== m(s!Prio)) ++
        releases.map(r => Var(r!Order) \#== m(r!Order)) ++
        (for (s <- stakeholders; f <- features) yield
            Var(s!f!Benefit) \#== m(s!f!Benefit)) ++
        (for (res <- resources; f <- features) yield
            Var(res!f!Cost) \#== m(res!f!Cost)) ++
        (for (res <- resources; rel <- releases) yield
            Var(res!rel!Capacity) \#== m(res!rel!Capacity))
    )
```
All Features shall have an Order integer attribute to model that it can be allocated to some Release (corresponding to the Order attribute of that Release).

```
features.map(f => (f!Order)::[1 to releases.size])
```
For all stakeholders $s$ and all features $f$: 
$\text{Var}(\text{benefit}(s,f))$ is the benefit of the feature according to that stakeholder multiplied with the priority of the stakeholder.

```
for (s <- stakeholders; f <- features) yield
    XmulYeqZ(
        Var(s!f!Benefit), Var(s!Prio), Var(s"benefit($s,$f")")
    )
```
For all features f, for all stakeholders s:
Var(benefit(f)) is the sum of all stakeholders’ benefits of that f:

\[ \text{benefit}(f) = \sum_s \text{benefit}(s, f) \]

features.map(f =>
    Sum(
        stakeholders.map(s => Var(s"benefit($s,$f)")),
        Var(s"benefit($f)"
    )
)
for all releases r, for all features f:
if f is allocated to r then benefit(r, f) = benefit(f)
else benefit(r, f) = 0

for (r <- releases; f <- features) yield
  IfThenElse(
    Var(f!Order) #== Var(r!Order),
    Var(s"benefit($r,$f)") #== Var(s"benefit($f)")",
    Var(s"benefit($r,$f)") #== 0
  )
For all releases \( r \), for all features \( f \):

\[
\text{totBenefit}(r) = \sum_f \text{benefit}(r, f)
\]

\begin{verbatim}
for (r <- releases) yield 
  Sum(
    features.map(f => Var(s"benefit($r,$f"))),
    Var(s"totBenefit($r")
  )
\end{verbatim}
For all releases rel, for all features f, for all resources res:
If f is allocated to rel then cost(rel, f, res) is the cost of that feature needed by that resource, else it is zero.

for (rel <- releases; f <- features; res <- resources) yield
  IfThenElse(
    Var(f!Order) #== Var(rel!Order),
    Var(s"cost($rel,$f,$res")") #== Var(res!f!Cost),
    Var(s"cost($rel,$f,$res")") #== 0
  )
For all resources res, for all releases rel, for all features f:

\[
totCost(rel, res) = \sum_f cost(rel, f, res)
\]

for (res <- resources; rel <- releases) yield

Sum(
  features.map(f => Var(s"cost($rel,$f,$res"))),
  Var(s"totCost($rel,$res")

)
For all resources res, for all releases rel:

\[ \text{totCost}(rel, res) \leq \text{availableCapacity}(res, rel) \]

```
for (res <- resources; rel <- releases) yield
    XlteqY(
        Var(s"totCost($rel,$res)"),
        Var(res!rel!Capacity)
    )
```
For all releases \( rel \), for all resources \( res \):

\[
\text{totCost}(rel) = \sum_{res} \text{totCost}(rel, res)
\]

\[
\text{for (rel <- releases) yield Sum(
    resources.map(res => Var(s"totCost($rel,$res"))),
    Var(s"totCost($rel)"))}
\]
val releasePlanningConstraints = Constraints(
  features.map(f => Var(f!Order)::[1 to releases.size]) ++
  (for (s <- stakeholders; f <- features) yield
   XmulYeqZ(Var(s!f!Benefit), Var(s!Prio), Var(s"benefit($s,$f")))++) ++
  features.map(f => Sum(stakeholders.map(s =>
    Var(s"benefit($s,$f")), Var(s"benefit($f")))++) ++
  (for (r <- releases; f <- features) yield
   IfThenElse(Var(f!Order) #== Var(r!Order),
    Var(s"benefit($r,$f") #== Var(s"benefit($f")),
    Var(s"benefit($r,$f") #== 0)) ++
  (for (r <- releases) yield
   Sum(features.map(f => Var(s"benefit($r,$f")), Var(s"totBenefit($r")))++) ++
  (for (rel <- releases; f <- features; res <- resources) yield
   IfThenElse(Var(f!Order) #== Var(rel!Order),
    Var(s"cost($rel,$f,$res") #== Var(res!f!Cost),
    Var(s"cost($rel,$f,$res") #== 0)) ++
  (for (res <- resources; rel <- releases) yield
   Sum(features.map(f => Var(s"cost($rel,$f,$res"))), Var(s"totCost($rel,$res")))++) ++
  (for (res <- resources; rel <- releases) yield
   XlteqY(Var(s"totCost($rel,$res")), Var(res!rel!Capacity))) ++
  (for (rel <- releases) yield
   Sum(resources.map(res => Var(s"totCost($rel,$res"))), Var(s"totCost($rel")))
)
val constraints =
    assignValuesFromModel(m) ++
    releasePlanningConstraints
val utility = Var("totBenefit(Release(a))")
val (m2, r) =
    Model().impose(constraints).solve(Maximize(utility))

reqT> val allocationModel = m2 / Feature
   allocationModel: reqt.Model =
   Model(
       Feature("F3") has Order(1),
       Feature("F1") has Order(2),
       Feature("F2") has Order(2)
   )

reqT> val cost = r.lastSolution(Var("totCost(Release(a))"))
   cost: Int = 70
**Coupling:** Two features must be in the same release:

\[(\text{Feature("F1")!Order}) \#== (\text{Feature("F2")!Order})\]

**Precedence:**
One feature must be implemented before another feature:

\[(\text{Feature("F2")!Order}) \#< (\text{Feature("F3")!Order})\]
Conclusions and Discussion

ReqT integrates RE with CSP using an object-functional embedded DSL to solve decision and resource allocation problems in software engineering.

- Some results so far:
  - Basic DSL in place using immutable Scala case classes
  - Hiding: variable – store dependencies, mutability etc.
  - Integration with JaCoP search parameters, including time-out and solutions limit

- Outlook on future work:
  - More complete implementation of JaCoP 4.0 constraints
  - GUI support (MSc Thesis: Oskar Präntare & Joel Johansson)
  - Soft constraints
  - Stochastic constraints

Any feedback, question, input etc. welcome!!

http://reqT.org