A Scala Embedded DSL for Combinatorial Optimization in Software Requirements Engineering

Regnell, Björn; Kuchcinski, Krzysztof

2013

In proceedings of COSpeL: First Workshop on Domain Specific Languages in Combinatorial Optimization, Uppsala, Sweden, Sept. 16, pp. 19-34, 2013

A Scala Embedded DSL for Combinatorial Optimization in Software Requirements Engineering

Björn Regnell and Krzysztof Kuchcinski

Dept. of Computer Science, Lund University, Sweden
{bjorn.regnell|krzysztof.kuchcinski}@cs.lth.se

Abstract. The goal of the presented work is to provide support for software requirements engineering domain experts in modeling combinatorial optimization problems that arise in requirements prioritization and release planning. A Domain-Specific Language (DSL), called reqT/CSP, is presented that integrates constraints modeling with requirements modeling. The DSL is embedded in the object-functional Scala programming language. The DSL is demonstrated using principal examples of priority ranking and release planning. Benefits, limitations and future work are discussed.

Keywords: domain-specific language, combinatorial optimization, software engineering, requirements engineering, release planning, prioritization, embedded DSL, solver, constraint satisfaction programming, CSP, JaCoP, Scala

1 Introduction

This paper presents on-going work on a Domain-Specific Language (DSL) for combinatorial optimization in software Requirements Engineering (RE). The presented language for expressing Constraint Satisfaction Problems is called reqT/CSP, which is a sub-DSL of a larger requirements modeling DSL called reqT that aims to provide a semi-formal, open and scalable tool for RE [33].

Combinatorial problems are intrinsic to several sub-disciplines of RE, such as prioritization [12, 20], product line modeling [10, 38], and release planning [7, 40]. The presented work aims to provide a DSL for combinatorial optimization that integrates requirements modeling and constraint modeling in the same language. The presented research is guided by this research question:

> **Research Question**: How to support domain experts in modeling and solving combinatorial optimization problems in Requirements Engineering?

We embark on the above quest through software tool implementation and DSL development based on reqT [1] and the JaCoP open source solver [2]. The reqT tool provides a Domain-Specific *Embedded* Language (DSEL, a.k.a EDSL or internal DSL) that is implemented "inside" its host language, to take advantage of the available host infrastructure [19, 41]. The host language of both reqT and its sub-language reqT/CSP

is *Scala*, an object-functional, general purpose programming language running on the Java Virtual Machine [29]. Scala was chosen for its flexible syntax suitable for DSL embedding [39]. Furthermore, with Scala's Read-Evaluate-Print-Loop (REPL)[1], users of reqT/CSP can interactively and incrementally model and investigate constraint optimization problems as an integrated part of their requirements modeling, while utilizing the power of Scala and its comprehensive libraries.

The paper is organized as follows. Section 2 relates the presented research to previous work and provides background information on the reqT DSL and the JaCoP solver. In Section 3, the main contribution of this paper is explained through examples of usage of the reqT/CSP language and selected details of its implementation. Section 4 concludes the paper with a discussion of benefits and limitations of the proposed approach and an outlook on future research directions.

2 Background and Related work

Requirements Engineering (RE) is a research discipline recognized within Software Engineering already in the 1970's [9] and focuses on the intentional level of software development and the decision-making regarding what software to build. RE involves intertwined sub-processes such as elicitation, specification, validation and prioritization [12, 20, 24], as well as market-oriented activities [34] related to software product management [13, 16] including software release planning [7, 18]. Combinatorial optimization problems in RE occur in several areas [31], including:

- *prioritization*, where a set of requirements are assessed based on criteria, such as benefit, cost and risk, to find the requirements of highest priority, while balancing the views of selected stakeholders [12, 32];
- *release planning*, where a set of requirements are scheduled for subsequent development based on their priorities as well as resource constraints and scheduling constraints such as precedence and coupling, while trying to optimize aspects such as stakeholder benefit versus implementation cost [7, 15, 40];
- *product line modeling*, where a set of variation points may include different combinations of optional features depending on various constraints offering a family of products, while trying to optimize aspects such as reuse and time-to-market [10,38].

This paper focuses on constraint solving for the first two of the above areas (for the third area see e.g. Salinesi et al. [36–38]). Combinatorial optimization in release planning have been studied using linear programming by Ruhe et al. [18, 28, 35] and Akker et al. [6, 25, 26], while we in this paper focus on constraint solving for release planning aiming to utilize the potential modeling flexibility of CSP [30, 31] compared to other optimization approaches.

Languages specifically tailored for constraint solving include e.g. MiniZinc [3] and AMPL [17]. They offer mathematical-like notations to define combinatorial problems and can be used as a common input notation for many solvers. These languages are general and can be used to specify combinatorial optimization problems of any domain.

[1] The REPL is a textual user interface for incremental compilation and execution of Scala code.

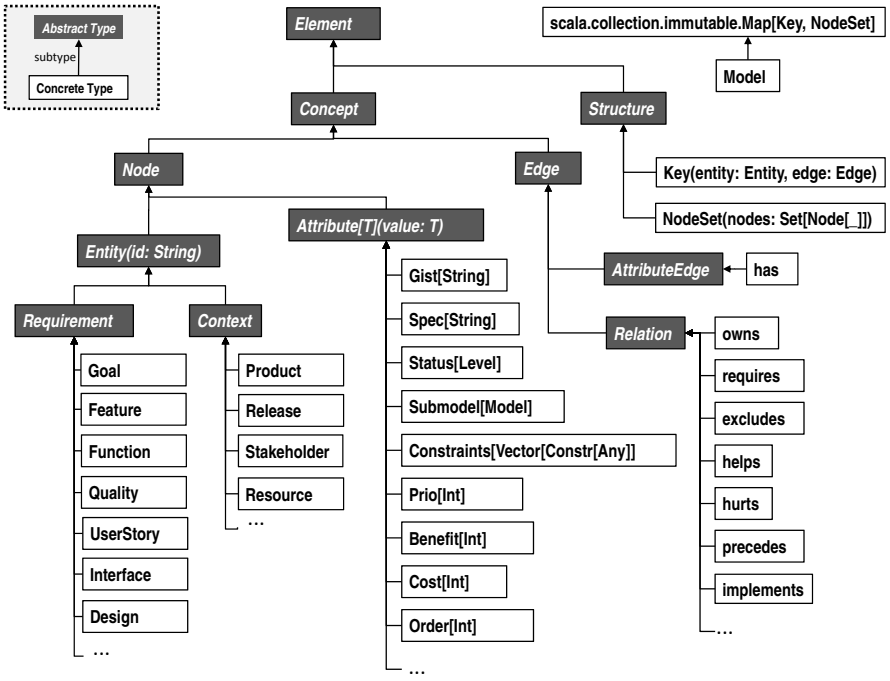

Fig. 1. Parts of the reqT metamodel.

However, languages such as MiniZinc [3] and AMPL [17] do not offer specialized abstractions for RE problems and they are typically compiled separately rather than integrated with an embedded DSL.

Kaplan [21] is a language where constraints can be expressed as lambdas using a subset of Scala, called PureScala. Kaplan is based on compile-time transformations using a compiler plug-in, while reqT/CSP is a library of "simple" case classes that the library implementation uses to automatically construct internal objects required by the solver back-end.

2.1 The reqT DSL

The reqT tool includes a DSL for requirements modeling [1, 33], allowing domain experts to specify and analyze requirements. The metamodel of the reqT DSL aims to provide RE-specific concepts that give flexible expressiveness to domain experts by allowing a mix of informal natural language text and graph-oriented formalizations of typed requirements entities, attributes and relations. The use of a DSL allows requirements to be represented as textual, computational entities that can be stored together with production code and test scripts in a common version management system.

By embedding the DSL [41] into Scala, reqT can utilize Scala's collection library. This enables domain experts to combine their model specifications with Scala scripts

that manipulate their models using existing collection operations. Models can also be traversed for various semantical checks, e.g. to investigate cycles and specific combinations of attributes etc.

The metamodel of reqT is shown in Figure 1 (only a selection of elements is shown for space reasons). A reqT Model is an immutable Map with a collection of pairs of Key and NodeSet, allowing fast direct indexing through any Key in the collection.

A Model can be constructed through a sequence of triplets
`<Entity> <Edge> <NodeSet>`
as exemplified in Listing 1.

Listing 1. An example of a reqT Model.

```
1 Model(
2   Feature("F1") has (Spec("The system shall..."), Status(IMPLEMENTED)),
3   UserStory("US1") has (Gist("As a user I want..."), Status(ELICITED)),
4   UserStory("US1") requires Feature("F1")
5 )
```

A model can be visualized using GraphViz [4]. When calling the method `toGraphViz` on a model, a dot language [4] export of the model is generated that can be rendered using GraphViz as shown in Figure 2. The Model has one `requires` relation between the two entities UserStory US1 and Feature F1, where the former has the attributes Gist and Status, while the latter has Spec and Status attributes. Entities, relations and attributes can be flexibly connected using the concepts of the reqT metamodel, partly depicted in Figure 1.

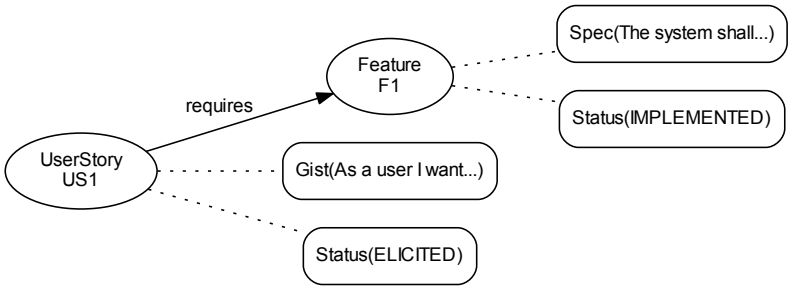

Fig. 2. The corresponding graph of the example in Listing 1. Oval nodes represent entities. Rounded rectangles represent attributes. Solid arrowed lines represent typed relation edges. Dashed lines represent edges to attributes.

Model extraction and analysis. Users of reqT can carve out parts of models with special operators, including the restrict `/` and exclude `\` operators [33]. Given a Model `m`, the expression `m / Feature` evaluates to a new Model restricted to keys only containing entities of type Feature. The exclude operator used in the complementary expression `m \ Feature` yields a new Model with all keys of m that do *not* have a Feature entity. It is also possible to use the restrict and exclude operators over attributes and relations.

There are several other operators for, e.g., aggregation of models and for extracting parts of a model using depth first search by following relation edges. For further information of available operators and examples of their usage, see the reqT home page [1].

Models inside models. Any entity in a Model can contain a Submodel attribute that in turn can contain a Model, hence enabling a hierarchy of models in a recursive tree structure. A hierarchical modeling approach can be used for scalability reasons when there is a need to modularize large models, but also for expressing models where e.g. different stakeholder have different priorities for the same set of features, as is shown in Listing 8 in Section 3.2. References to values of an attribute of a certain entity is created using the bang operator, e.g. the expression `(Feature("x")!Prio)` constructs a reference to the `Prio` attribute value of `Feature("x")`. If models are contained inside models, references including submodel paths of arbitrary lengths can be constructed by successive application of the bang operator.

For example, the expression `(Stakeholder("s")!Feature("x")!Prio)` refers to the submodel of `Stakeholder("s")` and the `Prio` attribute value of `Feature("x")` in that submodel.

2.2 The JaCoP Solver

The solver back-end of reqT/CSP is implemented using JaCoP [23]. JaCoP is a general purpose constraint programming library implemented in Java. The solver implements both finite domain constraint for integer variables as well as set variables, and offers a rich set of global constraints [2].

There are two available front-ends for JaCoP for specification of variables, constraints and search methods: a MiniZinc front-end and a Scala API. The Scala DSEL presented in this paper is aimed at a higher abstraction level compared to the existing Scala API, and constraints and variables in reqT/CSP are exclusively based on immutable case classes [29] and integrated with the reqT metamodel, as the Constraint attribute can be attached to any entity of a reqT model.

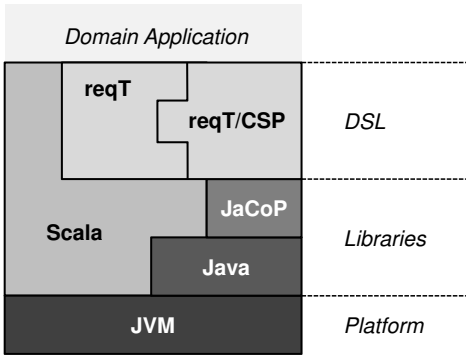

Fig. 3. Architecture layers of reqT/CSP. A horizontal border between layers represents a direct dependency, where the above layer is dependent on the layer below.

JaCoP has been used in several different domains for solving embedded combinatorial problems, e.g. in design automation for resource assignment and scheduling as well as computational pattern identification and selection [22, 27]. In the area of RE, JaCoP has been used for product line modeling [11]. JaCoP has been awarded a silver medal in the MiniZinc Challenge of 2010, 2011 and 2012 in the fixed search category [5].

3 The reqT/CSP DSL

The reqT/CSP DSL aims to integrate requirements modeling with constraint modeling while providing a high-level interface to the solver back-end that can wrap the search details when appropriate, but still allows access to search parameters when more detailed control over the solution space search is needed.

Architecture layers. Figure 3 shows the layers of the reqT architecture where the dependencies are shown by horizontal borders between components. The reqT/CSP DSL is integrated with reqT via the `Constraints` type that holds a collection of the base type of the reqT/CSP constraint type hierarchy, called `Constr`, as shown in the metamodel in Figure 1. The reqT/CSP DSL wraps the JaCoP solver back-end, that in turn depends on Java libraries. The user can create domain applications by combining constraint models in reqT/CSP with requirements models in reqT and general Scala scripting. Both reqT and reqT/CSP depend on Scala libraries.

Constraints can be expressed using variables that refer to values of integer attributes of reqT models, as exemplified in Listing 6 in Section 3.1. This enables integration of constraint modeling with requirements modeling, as constraints on requirements models can be stored inside requirements models, and solutions that may fulfill the constraints and optimize some variables can be searched for using the high-level interface to the solver back-end.

Listing 2. Some key parts of the implementation of reqT/CSP

```
1  case class Var[+T](ref: T) extends ... {
2    def #==[B >:T](that: Var[B]) = XeqY(this, that)
3    ...
4  }
5  case class Interval(min: Int, max: Int) extends ... {
6    def ::[T](v: Var[T]) = Bounds(Seq(v), Seq(this))
7    def ::[T](vs: Seq[Var[T]]) = Bounds(vs, Seq(this))
8  }
9  case class Bounds[+T](seq1: Seq[Var[T]], domain: Seq[Interval]) ...
10 case class SumBuilder[+T](vs: Seq[Var[T]]) {
11   def #==[B >:T](that: Var[B]) = SumEq(vs, that)
12 }
13 object Sum {
14   def apply[T](vs: Seq[Var[T]]) = SumBuilder(vs)
15 }
```

Implementation. Listing 2 shows excerpts of key classes of the reqT/CSP DSL, including the `Var` case class that represents an integer finite domain variable.[2] The identifying name of a variable is given by the `toString` value of the `ref` field. The `Var` case class has a series of operators for constructionof constraints, e.g. the `#==` operator that constructs the `XeqY` constraint. This allow users to write expressions such as `Var("x") #== Var("y")` to constrain two variables to the same value.[3]

[2] Scala's case classes provide automatic structural equality and pattern matching abilities.

[3] Scala's flexible syntax allows one-argument object method calls to be simplified, so that `Var("x").#==(Var("y"))` can be written `Var("x") #== Var("y")`

The `Interval` case class has a right-associative operator `::` that, via an implicit conversion between `Interval` types and Scala's `Range` type, allows for the construction of `Bounds` constraints with expressions such as `Var("x")::{1 to 10}` representing that `x` can hold values between 1 and 10.

`SumBuilder` has the `#==` operator that together with the `Sum` factory object allow expressions such as `Sum(Var("x"), Var("y"), Var("z")) #== Var("sum")` for constraining the sum of a sequence of variables to be equal to another variable's value.

Listing 3 shows the definition of four functions that generate sequences of variables or constraints. The `vars` generator is used at the second line of Listing 4 to construct a value f referring to a vector with 5 variables `Var("f0")` to `Var("f4")`. A bounding constraint on vectors or other sequences of variables can be constructed with the `::` operator as shown on line 4 of Listing 4.

The forAll functions takes a sequence of objects of some type T and applies a function that takes an object of type T and returns a constraint. This can be used to construct a sequence of constraints, e.g. as shown on line 9 in Listing 4. [4] Similarly, the sumForAll function generates a sequence of variables of a SumBuilder that in turn, with the `#==` operator, can be used to create a `SumEq` constraint, as exemplified in Listing 9 on line 11.

Listing 3. Functions for generating variables and constraints.

```
1 def vars[T](vs: T *): Vector[Var[T]] = vs.map(Var(_)).toVector
2 def vars(n: Int, prefix: String): Vector[Var[String]] =
3   (for (i <- 0 until n) yield Var(s"$prefix$i")).toVector
4 def forAll[T](xs:Seq[T])(f: T => Constr[_]) = Constraints(xs.map(f(_)))
5 def sumForAll[T](xs:Seq[T])(f: T => Var[_]) = SumBuilder(xs.map(f(_)))
```

3.1 Prioritization Example

Many different methods have been proposed for requirements prioritization, often implying that a specific algorithm is used to calculate priorities while requiring a specific form of input data from stakeholders [12]. One simple method for requirements prioritization is priority ranking [12], where requriements are assigned priorities on an ordinal scale. This method can be expressed using variables bounds to the interval `{1 to n}` where n is the number of requirements together with the `AllDifferent` constraint.

By also allowing general constraints over priority ranks, domain experts can express domain-specific concerns, e.g. that one specific requirement should have a higher rank than *all* other requirements, or that one specific requirements should have a higher rank than some another specific requirement. Thus, constraints add modeling flexibility for domain experts compared to if the ranks were derived using some specific sorting method such as bubble sort or binary search tree algorithms [8, 20, 31].

Listing 4 shows a priority ranking example using constraints, both the general constraints inherent to the ranking method, `f::{1 to n}` and `AllDifferent(f)`, as well as some additional, domain-specific constraints. The level of abstraction is comparable to the corresponding MiniZinc model shown in Listing 5.

[4] Scala's string interpolator `s` is used to insert values of identifiers into strings by the `$` sign.

We can also call `solve(FindAll)` on a sequence of constraints to find all possible solutions, which given the constraints of the example in Listing 4 would yield 3 different solutions.

Listing 4. Prio ranking in reqT/CSP

```
1  val n = 5
2  val f = vars(n, "f")
3  Constraints(
4    f::{1 to n},
5    AllDifferent(f),
6    f(0) #> f(1),
7    f(1) #>= f(2),
8    f(2) #< f(3),
9    forAll(0 until n)
10     { f(4) #>= f(_) }
11 ).solve(Satisfy)
```

Listing 5. Corresponding MiniZinc model

```
1  int: n = 5;
2  array[1..n] of var 1..n: f;
3  constraint
4    alldifferent(f);
5  constraint f[1] > f[2];
6  constraint f[2] > f[3];
7  constraint f[3] < f[4];
8  constraint
9    forall ( i in 1..n)
10     ( f[5] >= f[i] );
11 solve satisfy;
```

The example in Listing 4 demonstrates how reqT/CSP can be used as a general constraint programming DSL without explicit reference to reqT models. A major objective with reqT/CSP is to integrate constraint modeling with requirements modeling, avoiding the need for modeling in separate languages. This integration is demonstrated in Listing 6, where a set of reqT features in a requirements model `m` is combined with a sequence of constraints `cs`. The vector `f` includes references to `Prio` attributes of all entities of `m`, constructed with a lambda using the bang operator, as shown on line 8. The constraints in `cs` are then imposed on the model m and the solution to the combinatorial optimization issued by line 17 will make the resulting priorities to be inserted in a new model `m2` for each feature of `m`. The resulting model `m2` is shown in Listing 7. The search demonstrates combinatorial optimization by calling `solve` with the argument `Maximize(Feature("a")!Prio)`.

Listing 6. Prio ranking integrated with a requirements model in reqT

```
1  val m = Model(
2    Feature("a") has Spec("..."),
3    Feature("b") has Spec("..."),
4    Feature("c") has Spec("..."),
5    Feature("d") has Spec("..."),
6    Feature("e") has Spec("...")
7  )
8  val f = m.entityVector.map(_!Prio)
9  val cs = Constraints(
10     f::{1 to n},
11     AllDifferent(f),
12     (Feature("a")!Prio) #> (Feature("b")!Prio),
13     (Feature("b")!Prio) #>= (Feature("c")!Prio),
14     (Feature("c")!Prio) #< (Feature("d")!Prio),
15     forAll(0 until n) { (Feature("e")!Prio) #>= f(_) }
16   )
17 val (m2, result) = m.impose(cs).solve(Maximize(Feature("a")!Prio))
```

Listing 7. Resulting model of priority ranking in Listing 6

```
m2: reqt.Model =
Model(
  Feature("a") has (Spec("..."), Prio(4)),
  Feature("b") has (Spec("..."), Prio(3)),
  Feature("c") has (Spec("..."), Prio(1)),
  Feature("d") has (Spec("..."), Prio(2)),
  Feature("e") has (Spec("..."), Prio(5))
)
```

Listing 8. A reqT model with input parameters (prio, benefit, cost, capacity) to a principal release planning problem, as well as precedence and coupling constraints by stakeholders.

```
val m = Model(
  Stakeholder("Martin") has (Prio(10),
    Submodel(
      Feature("F1") has Benefit(20),
      Feature("F2") has Benefit(20),
      Feature("F3") has Benefit(20)
    ),
    Constraints((Feature("F2")!Order) #< (Feature("F3")!Order))
  ),
  Stakeholder("Anna") has (Prio(20),
    Submodel(
      Feature("F1") has Benefit(5),
      Feature("F2") has Benefit(15),
      Feature("F3") has Benefit(35)
    ),
    Constraints((Feature("F1")!Order) #== (Feature("F2")!Order))
  ),
  Resource("DevTeam") has Submodel(
    Release("Alfa") has Capacity(100),
    Release("Beta") has Capacity(100),
    Feature("F1") has Cost(10),
    Feature("F2") has Cost(70),
    Feature("F3") has Cost(20)
  ),
  Resource("TestTeam") has Submodel(
    Release("Alfa") has Capacity(100),
    Release("Beta") has Capacity(100),
    Feature("F1") has Cost(40),
    Feature("F2") has Cost(10),
    Feature("F3") has Cost(50)
  ),
  Release("Alfa") has Order(1),
  Release("Beta") has Order(2)
)
```

3.2 Release Planning Example

Listing 8 shows how reqT/CSP is used to express a principal example of input parameters to a fictitious release planning problem. Model `m` includes two stakeholders (Anna and Martin), three features (F1, F2 and F3), two resources (DevTeam and TestTeam), and two releases (Alfa and Beta).

The preferences of Anna has double priority compared to Martin. Stakeholder preferences are expressed using the `Benefit` attribute. Each resource has a `Capacity` for each release and an estimated `Cost` for each feature. The temporal ordering of releases are modeled using the `Order` attribute.

The stakeholders also have their own specific constraints on precedence and coupling, where Martin wants F2 to be implemented before F1, while Anna wants F1 to be in the same release as F2.

Listing 9. Generating release planning constraints based on an input model

```
def releasePlanningConstraints(m: Model): Constraints = {
  val features = (m.flatten / Feature).sourceVector
  val releases = (m / Release).sourceVector
  val resources = (m / Resource).sourceVector
  val stakeholders = (m / Stakeholder).sourceVector
  m.constraints ++ Constraints(
    forAll(features) { f => (f!Order)::{1 to releases.size} } ++
    forAll(stakeholders, features) { (s, f) =>
      (s!f!Benefit) * (s!Prio) #== Var(s"benefit($s,$f)") } ++
    forAll(features) { f =>
      sumForAll(stakeholders)(s => Var(s"benefit($s,$f)")) #==
        Var(s"benefit($f)") } ++
    forAll(releases, features) { (r, f) =>
      IfThenElse((f!Order) #== (r!Order),
        Var(s"benefit($r,$f)") #== Var(s"benefit($f)"),
        Var(s"benefit($r,$f)") #== 0) } ++
    forAll(releases) { r =>
      sumForAll(features)(f => Var(s"benefit($r,$f)")) #==
        Var(s"totBenefit($r)") } ++
    forAll(releases,features, resources) { (rel, f, res) =>
      IfThenElse((f!Order) #== (rel!Order),
        Var(s"cost($rel,$f,$res)") #== (res!f!Cost),
        Var(s"cost($rel,$f,$res)") #== 0) } ++
    forAll(resources, releases) { (res, rel) =>
      sumForAll(features)(f => Var(s"cost($rel,$f,$res)")) #==
        Var(s"totCost($rel,$res)") } ++
    forAll(resources,releases) { (res, rel) =>
      Var(s"totCost($rel,$res)") #<= (res!rel!Capacity) } ++
    forAll(releases) { rel =>
      sumForAll(resources)(res => Var(s"totCost($rel,$res)")) #==
        Var(s"totCost($rel)") }
  )
}
```

There are many ways of modeling the release planning problem, depending on e.g. (1) what type of information that domain experts want to take into account, (2) which input parameters are meaningful to estimate given uncertainties of future prediction, (3) what type of constraints are important in a specific case, etc.

In Listing 9, a function named `releasePlanningConstraints` is defined that demonstrates *one* particular way of modeling the release planning problem. This function returns Constraints, which can be used in subsequent searches for optimal solutions, and in what-if analysis [6]. Firstly, four vectors of features, releases, resources and stakeholders are extracted from the input model `m` via the restrict operator and the `sourceVector` method. As features are stored in submodels, the model is flattened to pull all features to the top level. The expression `m.constraints` yields all constraints in model `m`, e.g. collecting all precedence and coupling constraints of all stakeholders.

The `++` operator is used to concatenate vectors of constraints. In Listing 9 concatenation is applied to avoid the need of flattening, as `forAll` returns a vector of constraints.

There are other ways of expressing the release planning problem. The problem can, for example, be re-modeled with extra binary variables and the `Reified` constraint and multiplication constraints instead of `IfThenElse`, or the `Binpacking` global constraint can be used.

Listing 10 shows how to issue a search for a solution that optimizes the total benefit of the first release. The `impose` method on a `Model` combines the constraints in that model with additional constraints and returns an intermediate object than can be used to parameterize and issue a solve process taking all constraints into account.

A new model `m2` is computed with the result of imposing the release planning constraints on model `m`, thus taking the stakeholders' constraints from Listing 8 into account. Should the stakeholders' constraints be inconsistent, no solution can be found unless the constraints that are responsible for the inconsistency first are excluded. The resulting feature allocation to releases via the `Order` attribute is extracted from `m2` together with the total cost of the first release, with output as shown in Listing 11.

Listing 10. Searching for an optimal solution given an input model.

```
1 val utility = Var("totBenefit(Release(Alfa))")
2 val (m2, result) =
3   m.impose(releasePlanningConstraints(m)).solve(Maximize(utility))
4 val featureAllocation = m2 / Feature
5 val costOfAlfa = result.lastSolution(Var("totCost(Release(Alfa))"))
```

Listing 11. Output results of the release planning problem in Listings 8–10

```
1 featureAllocation: reqt.Model =
2   Model(
3     Feature("F3") has Order(2),
4     Feature("F1") has Order(1),
5     Feature("F2") has Order(1)
6   )
7 costOfAlfa: Int = 130
```

3.3 Search Parameters and Result

A simple, black-box constraint solving can be issued by calling `solve()`, thereby utilizing all default values of the available search parameters to the `solve` methods. However, if the solution is hard for the solver to find within reasonable time, it may be necessary to open provide more control over the pruning of the solution space. The solve method has a number of parameters that are fed to the solver back-end, as shown in Listing 12.

Listing 12. Parameters to the solve method and their default values.

```
def solve[T](
      searchType: SearchType = Satisfy,
      timeOutOption: Option[Long] = None,
      solutionLimitOption: Option[Int] = None,
      valueSelection: ValueSelection = IndomainRandom,
      variableSelection: VariableSelection = InputOrder,
      assignOption: Option[Seq[Var[T]]] = None
  ): Result[T] = ...
```

The default search type is `Satisfy`. Other options are `CountAll` and `FindAll` to search for multiple solutions, while counting or recording all found solutions. A search for an optimal value for some variable `v` can be issued with the search types `Minimize(v)` or `Maximize(v)`.

The default value selection method is `IndomainRandom`. Also `IndomainMax`, `IndomainMedian`, `IndomainMiddle`, and `IndomainMin` can be used.

The default variable selection method is `InputOrder`. Other available selection methods are: `MaxRegret`, `SmallestDomain`, `LargestDomain`, `LargestMin`, `LargestMax`, `SmallestMin`, `SmallestMax`, `MostConstrainedDynamic`, and `MostConstrainedStatic`. (For further explanation of selection methods, see the JaCoP documentation [2].)

Listing 13. The Result case class that holds solution data.

```
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
)

trait Solutions[T] {
  def nSolutions: Int
  def solutionMap(solutionIndex: Int): Map[Var[T], Int]
  def valueVector(v: Var[T]): Vector[Int]
  def printSolutions: Unit
  ...
}
```

Listing 13 shows the type of objects that are returned as a result of calling the `solve` method on `Constraints`. The type `Conclusion` is the base type of the case objects named `SolutionFound`, `SolutionNotFound`, `InconsistencyFound` that, to-

gether with the case class `SearchFailed(msg: String)`, are possible conclusions of a search result.

The values `Some(SearchTimeOut)` or `Some(SolutionLimitReached)` are given if a search was interrupted. If a solution was found the `solutionOption` value includes a optional instance of the `Solutions` trait that allows for iteration over solutions using the `solutionMap` as defined at line 11 in Listing 13.

4 Discussion and Conclusion

This paper presents the reqT/CSP DSL and demonstrates how it can be used to express combinatorial optimization problems in requirements engineering, and how a solver back-end can be invoked to search the solution space. This section concludes the presentation by discussing benefits, limitations and future work.

4.1 Benefits

The presented DSL provides an interface to the back-end solver at a level of abstraction comparable to MiniZinc (see Listing 5). The integration with reqT enables domain experts to combine requirements and constraints modeling. Embedding a requirements engineering constraint modeling language in Scala and connecting it to a full-featured constraint solver such as JaCoP makes it possible to use the full power of the host language for scripting and custom extensions in concert with the solver capabilities.

Moreover, a general constraint solver offers a rich catalog of constraints that can be used for modeling combinatorial optimization problems in different ways and for controlling the search with respect to e.g. value and variable selection.

Compared to the existing Scala API for JaCoP [2], the reqT/CSP DSL has a higher abstraction level and enables more concise search parameterization.

4.2 Limitations

The presented work is still on-going and reqT and its CSP sub-language is still experimental and has not yet been validated in real-world software engineering. The interface to the back-end solver is still incomplete and only a part of its available capabilities is exposed. The wrapper also introduces a layer of indirection that may have performance penalties.

The current implementation of reqT/CSP is in some aspects specific to JaCoP, and the DSL exposes JaCoP-specific search parameters in cases where users choose to go beyond default parameters. Also, some case class names are based on JaCoP classes, e.g. `XeqY`, chosen in absence of a globally standardized constraint nomenclature (alternative operator notation, e.g. `#==` in line with the Prolog tradition is, however, also available). In principal it would be possible to change the solver back-end from JaCoP to any other solver that has a Java-based API, but some (probably minor) adjustments to the DSL and (probably localized) refactoring of the implementation would then be necessary.

4.3 Future work

In general, the *scalability* of the approach needs to be investigated. Also, the *usability* of the presented DSL would be interesting to assess in empirical studies with domain experts, and it remains to be seen if reqT can effectively solve real world combinatorial optimization problems in requirements engineering.

Furthermore, in terms of DSL *expressiveness*, it would be interesting to investigate how the inherent uncertainties of software requirements can be expressed, as requirements in practice often contain volatile information and rough estimations of different parameters. For example, the future benefit and cost of implementing a given feature can be estimated differently by different stakeholders. In this case, constraint programming with *soft constraints* and/or *stochastic constraints* may be useful. Soft constraints enable specification of situations were constraints sometimes may be violated, e.g. to some specified cost. Stochastic constraints make it possible to assign probabilities to different values and calculate the most probable outcome or an expected value for a given parameter [14].

From *implementation generalization* perspective, further work may include a more complete exposure of the available back-end solver capabilities. It would also be interesting to investigate the pros and cons of generalizing the back-end solver wrapper by removing all explicit dependencies to a specific solver, trading off the potential lack of implementation-specific search process control with the potential gains of increased flexibility in choice of solver technology.

In conclusion, although this work demonstrate potential utility of the proposed DSL for combinatorial optimization in requirements engineering, future extensions and more validation are needed.

Acknowledgments. *This work is partly funded by VINNOVA within the EASE project.*

References

1. reqT web page, `http://reqT.org/`, visited June 2013.
2. JaCoP web page, `http://www.jacop.eu/`, visited June 2013.
3. MiniZinc web page, `http://www.minizinc.org`, visited June 2013.
4. GraphViz web page, `http://www.graphviz.org/`, visited June 2013.
5. MiniZinc Challenge web page, `http://www.minizinc.org/challenge.html`, visited June 2013.
6. van den Akker, M., Brinkkemper, S., Diepen, G., Versendaal, J.: Software product release planning through optimization and what-if analysis. Information and Software Technology 50(1-2), 101 – 111 (2008)
7. Bagnall, A., Rayward-Smith, V., Whittley, I.: The next release problem. Information and Software Technology 43(14), 883 – 90 (2001)
8. Bebensee, T., van de Weerd, I., Brinkkemper, S.: Binary priority list for prioritizing software requirements. In: Wieringa, R., Persson, A. (eds.) Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science, vol. 6182, pp. 67–78. Springer Berlin / Heidelberg (2010)

9. Bell, T.E., Thayer, T.: Software requirements: Are they really a problem? In: Proceedings of the 2nd international conference on Software engineering. pp. 61–68. IEEE Computer Society Press (1976)
10. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Information Systems 35(6) (2010)
11. Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A.: Fama: Tooling a framework for the automated analysis of feature models. In: Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS). pp. 129–134 (2007)
12. Berander, P., Andrews, A.: Requirements prioritization. In: Aurum, A., Wohlin, C. (eds.) Engineering and Managing Software Requirements, pp. 69–94. Springer Berlin Heidelberg (2005)
13. Brinkkemper, S., Ebert, C., Versendaal, J.: Proceedings of the first international workshop on software product management. In: Software Product Management, 2006. IWSPM '06. International Workshop on. pp. 1 –2 (sept 2006)
14. Buddha, S.K.: Reasoning under uncertainty: Stochastic constraint programming using jacop. Tech. rep., Ecole polytechnique fédérale de Lausanne, Laboratory of Articial Intelligence (LIA) (2011)
15. Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B., Natt och Dag, J.: An industrial survey of requirements interdependencies in software product release planning. pp. 84 – 91 (2001)
16. Ebert, C.: The impacts of software product management. Journal of Systems and Software 80(6), 850 – 861 (2007)
17. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. Duxbury Press / Brooks/Cole Publishing Company (2003), http://www.ampl.org
18. Greer, D., Ruhe, G.: Software release planning: an evolutionary and iterative approach. Information and Software Technology 46(4), 243 – 253 (2004)
19. Hudak, P.: Modular domain specific languages and tools. In: Software Reuse, 1998. Proceedings. Fifth International Conference on. pp. 134–142. IEEE (1998)
20. Karlsson, J., Wohlin, C., Regnell, B.: An evaluation of methods for prioritizing software requirements. Information and Software Technology 39(14-15), 939–947 (1998)
21. Köksal, A.S., Kuncak, V., Suter, P.: Constraints as control. ACM SIGPLAN Notices 47(1), 151–164 (2012)
22. Kuchcinski, K.: Constraints-driven scheduling and resource assignment. ACM Transactions on Design Automation of Electronic Systems (TODAES) 8(3), 355–383 (Jul 2003)
23. Kuchcinski, K., Szymanek, R.: JaCoP Library. User's Guide. http://www.jacop.eu (2013)
24. Lauesen, S.: Software Requirements - Styles and Techniques. Addison-Wesley (2002)
25. Li, C., van den Akker, J., Brinkkemper, S., Diepen, G.: Integrated requirement selection and scheduling for the release planning of a software product. In: Sawyer, P., Paech, B., Heymans, P. (eds.) Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science, vol. 4542, pp. 93–108. Springer Berlin / Heidelberg (2007)
26. Li, C., van den Akker, M., Brinkkemper, S., Diepen, G.: An integrated approach for requirement selection and scheduling in software release planning. Requirements Engineering 15, 375–396 (2010)
27. Martin, K., Wolinski, C., Kuchcinski, K., Floch, A., Charot, F.: Constraint programming approach to reconfigurable processor extension generation and application compilation. ACM Trans. on Reconfigurable Technology and Systems (TRETS) 5(2), 10:1–10:38 (Jun 2012), http://doi.acm.org/10.1145/2209285.2209289
28. Ngo-The, A., Ruhe, G.: Optimized resource allocation for software release planning. IEEE Transactions on Software Engineering 35(1), 109 – 23 (2009)
29. Odersky, M., et al.: An overview of the Scala programming language. Tech. rep. (2004), http://lampwww.epfl.ch/~odersky/papers/ScalaOverview.html

30. Przepiora, M., Karimpour, R., Ruhe, G.: A hybrid release planning method and its empirical justification. In: Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement. pp. 115–118. ESEM '12 (2012)
31. Regnell, B., Kuchcinski, K.: Exploring software product management decision problems with constraint solving - opportunities for prioritization and release planning. In: Software Product Management (IWSPM), 2011 Fifth International Workshop on. pp. 47 –56 (2011)
32. Regnell, B., Host, M., Natt och Dag, J., Beremark, P., Hjelm, T.: An industrial case study on distributed prioritisation in market-driven requirements engineering for packaged software. Requirements Engineering 6(1), 51–62 (2001)
33. Regnell, B.: Reqt.org – towards a semi-formal, open and scalable requirements modeling tool. In: Doerr, J., Opdahl, A. (eds.) Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science, vol. 7830, pp. 112–118. Springer Berlin Heidelberg (2013)
34. Regnell, B., Brinkkemper, S.: Market-driven requirements engineering for software products. In: Aurum, A., Wohlin, C. (eds.) Engineering and Managing Software Requirements, pp. 287–308. Springer Berlin Heidelberg (2005)
35. Ruhe, G., Saliu, M.: The art and science of software release planning. Software, IEEE 22(6), 47 – 53 (nov-dec 2005)
36. Salinesi, C., Diaz, D., Djebbi, O., Mazo, R., Rolland, C.: Exploiting the versatility of constraint programming over finite domains to integrate product line models. 17th IEEE International Requirements Engineering Conference (RE'09) pp. 375–376 (2009)
37. Salinesi, C., Mazo, R., Diaz, D., Djebbi, O.: Using integer constraint solving in reuse based requirements engineering. 18th IEEE International Requirements Engineering Conference (RE'10) pp. 243–251 (2010)
38. Salinesi, C., Mazo, R., Djebbi, O., Diaz, D., Lora-Michiels, A.: Constraints: The core of product line engineering. In: Research Challenges in Information Science (RCIS), 2011 Fifth International Conference on. pp. 1–10. IEEE (2011)
39. Sujeeth, A.K., Rompf, T., Brown, K.J., Lee, H., Chafi, H., Popic, V., Wu, M., Prokopec, A., Jovanovic, V., Odersky, M., et al.: Composition and reuse with compiled domain-specific languages. In: Proceedings of ECOOP (2013)
40. Svahnberg, M., Gorschek, T., Feldt, R., Torkar, R., Saleem, S.B., Shafique, M.U.: A systematic review on strategic release planning models. Information and Software Technology 52(3), 237 – 248 (2010)
41. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM Sigplan Notices 35(6), 26–36 (2000)