Exploring software product management decision problems with constraint solving – opportunities for prioritization and release planning

Regnell, Björn; Kuchcinski, Krzysztof

Published in:
2011 Fifth International Workshop on Software Product Management (IWSPM)

DOI:
10.1109/IWSPM.2011.6046203

2011

Document Version:
Early version, also known as pre-print

Exploring Software Product Management Decision Problems with Constraint Solving – Opportunities for Prioritization and Release Planning

Björn Regnell, Krzysztof Kuchcinski
Dept. of Computer Science
Lund University
Sweden
Email: bjorn.regnell@cs.lth.se

***Abstract*—Decision-making is central to Software Product Management (SPM) and includes deciding on requirements priorities and the content of coming releases. Several algorithms for prioritization and release planning have been proposed, where humans with or without machine support enact a series of steps to produce a decision outcome. Instead of applying some specific algorithm to find an acceptable solution to a decision problem, we propose to model SPM decision-making as a Constraint Satisfaction Problem (CSP), where relative and absolute priorities, interdependencies, and other constraints are expressed as relations among variables representing entities such as feature priorities, stakeholder preferences, and resource constraints. The solution space is then explored with the help of a constraint solver without humans needing to care about specific algorithms. This paper discusses advantages and limitations of CSP modeling in SPM and gives principal examples as a proof-of-concept of CSP modeling in requirements prioritization and release planning. A discussion of further research on constraint solving in SPM is also given.**

***Keywords*-software product management; requirements engineering; release planning; prioritization; constraint programming;**

I. Introduction

Software Product Management (SPM) involves decision-making that is critical to the success or failure of software products [1]. In particular, a product manager needs to establish requirements priorities and determine release plans in dialog with internal and external stakeholders [2], [3]. Several approaches have been proposed for these decisions problems, e.g. binary search trees for requirements prioritization [4], [5], and, e.g. optimization by linear programming [6], [7], [8] or genetic algorithms [9] for release planning. These approaches stipulate a specific algorithm to be enacted by humans only, or by humans using a supporting computer program.

In this paper we propose a different approach, namely to support SPM by modeling a decision problem independently of a solution algorithm as an integer finite domain Constraint Satisfaction Problem (CSP) [10] and then use a integer finite domain CSP solver such as JaCoP [11], [12] to explore the set of solutions that satisfy the constraints. There are several potential benefits of this approach, including:

- *Flexible specification.* CSP specification of SPM decision problems is flexible in the sense that it is possible to combine many different types of constraints on a problem in one single model without changing the underlying method for solving it.
- *Interactive exploration.* It is possible to get a single solution as well as deriving all solutions, when multiple solutions exist. Thus, stakeholders can investigate the solution space as well as the consequences of changing the constraints to better understand what decision parameters that influence properties of the solution. In cases when the set of all solutions is too large to be practically feasible to investigate exhaustively, adding further constraints incrementally can help reduce the search space.
- *Optimization support.* CSP enables optimization to obtain a solution that maximizes or minimizes desired properties. Existing CSP solvers have a large set of pre-defined constraints providing efficient algorithms for finding solutions to optimization problems, which can be used concisely to model SPM decision problems. By restricting the problem specification to constraints over integer finite domains, we avoid many problems of algorithmic complexity. As subsequently illustrated, integer value approximations of real values are sufficient for many SPM problems.

The contribution of this paper is threefold: (1) a demonstration of how to apply a CSP approach to SPM using examples from requirements prioritization and release planning, together with (2) a discussion of benefits and limitation of the approach, as well as (3) a set of issues of further research into how CSP tools can be developed to provide powerful support for decision-making in SPM.

With these contributions we do not claim to escape the inherent difficulties of real-world prioritization and release planning. Our aim is instead to add an alternative to existing algorithmic approaches, such as the Analytical Hierarchy Process [4] for prioritization or Integer Linear Programming [7], [31] for release planning – an alternative that can provide a powerful, general and concise toolbox for flexible investigation of the nature of real-world SPM decision problems (rather than their algorithmic solution) by combining standard, high-level, declarative languages, such as MiniZinc [14], with ready-made optimization algorithms in CSP solvers, such as JaCoP [12].

The paper is organized as follows. Section II provides a short introduction to constraint satisfaction and gives

examples of existing algorithmic approaches to requirements prioritization and release planning. In Section III a constraint-driven approach to requirements prioritization is illustrated through examples. Section IV demonstrates how a previously published release planning example can be concisely modeled as a CSP. In Section V we discuss benefits and limitations of the proposed approach, and Section VI concludes the paper with a list of issues of further research in order to forward the utility of CSP as a vehicle for decision-making in SPM.

II. Related Work

This section provides a brief introduction to constraint satisfaction. Examples of related work from two areas relevant to SPM are given, namely requirements prioritization and release planning, to give a basis for the subsequent sections where CSP modeling is proposed as a new approach to support SPM in deciding priorities and release plans.

A. Constraint Satisfaction

We here provide a short introduction to constraint programming. A more thorough discussion on constraint programming and its application can e.g. be found in Rissi et al [13] or Tsang [10].

Formally, a *constraint satisfaction problem* (CSP) is defined as a 3-tuple $\mathcal{S} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$ is a *set of variables*, $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$ is a set of *finite domains*, and $\mathcal{C}$ is a set of *constraints*. Finite domain variables are defined by their domains, i.e. the range of values that can be assigned to the variables. A finite domain is usually expressed using integers, for example $x :: 1..7$, meaning that x can take integer values ranging from 1 to 7. A constraint $c(x_1, x_2, \ldots, x_n) \in \mathcal{C}$ among variables of $\mathcal{V}$ is a subset of $D_1 \times D_2 \times \ldots \times D_n$ that restricts which combinations of values the variables can simultaneously take. Equations, inequalities and even programs can define a constraint.

A *solution to a CSP* is the assignment of a value from a variable's domain, for each variable so that all constraints are satisfied. For a specific problem we may want *just one solution*, *all solutions* or a (close to) *optimal solution* given some objective function defined in terms of the variables.

A CSP solver is built using consistency methods and systematic search procedures. *Consistency methods* try to remove inconsistent values from the domains in order to reach a set of pruned domains such that their combinations are valid solutions. Each time a value is removed from a finite domain, all the constraints that contain that variable are revised. Most consistency methods are not complete and the solver needs to explore the remaining domains for a solution using search.

Solutions to a CSP are usually found by systematically assigning values taken from the variables' domains to the variables. This assignment is implemented as a *depth-first-search*. The consistency method is called as soon as the domains of the variables for a given constraint are pruned. If a partial solution violates any of the constraints, backtracking will take place, reducing the size of the remaining search space. The solvers can also find an optimal solution that is defined by combining depth-first-search with a *branch-and-bound* algorithm.

An example search for a single solution is depicted in Figure 1. The example problem has four finite domain variables and five inequality constraints. The search assigns value one to variable x_0 and the solver propagates this decision by pruning the domains of variables x_1 and x_2. Finally, assignment to variable x_2 triggers solver propagation and a solution is obtained. Solver consistency and propagation methods makes it possible in this case to find the solution based on two variables instead of four.

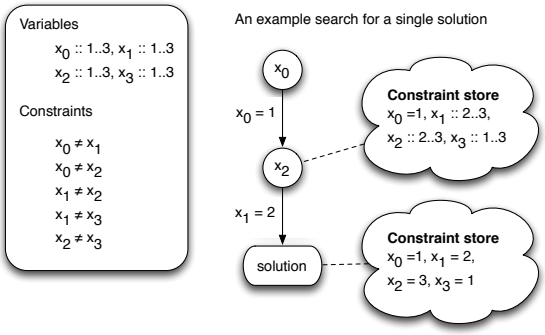

Figure 1. An example depth-first-search for a single solution.

For the purpose of illustrating how constraint solving can be applied to SPM we use our JaCoP solver [11] but our methods can be used together with other available solvers.

MiniZinc [14] is standard language for specifying finite domain CSP models. MiniZinc is high-level enough to express most constraint problems, but low-level enough to be mapped onto existing solvers. MiniZinc is a subset of the more general Zinc language. MiniZinc includes constructs for specifying variables, their domains and constraints over variable using predicates. MiniZinc specifications are compiled to FlatZinc and executed using some CSP solver such as JaCoP [11]. In subsequent sections we use MiniZinc specifications to illustrate the application of CSP modeling to SPM.

CSP has been applied in a variety of areas [13], [10], including e.g. feature modeling [15], program comprehension [16], testing [17], and hardware synthesis [11]. In particular, CSP has also been applied in requirements engineering by Salinesi et al [18], [19] where constraint solving is used in a product line engineering context. Salinesi et al. use constraints to express feature models and apply constraint solving to find a set of features that satisfies the feature model, with respect to selected or excluded features according to the variation points of the feature model.

This paper proposes to extend the application of CSP to the SPM-relevant areas of prioritization and release planning, which we believe are particularly suitable for CSP modeling. The subsequent subsections provide some relevant examples of related work in these two areas.

B. Requirements Prioritization

Requirements prioritization involves identifying high-priority requirements to reflect the opinions of the stakeholders of a system and its future evolution [20], [21], [4], [22]. SPM is carried out in a market-driven context [23], and prioritization in software product development typically involves product managers that gather estimates of the future business value of candidate features of coming software releases and balancing this information against product investment costs in terms of e.g. implementation effort [24], [2].

Requirements prioritization research has for more than a decade produced new knowledge into specific methods, see e.g. the systematic review by Herrmann et al [25] identifying 15 different requirements prioritization methods. Common to these methods are that they propose a specific algorithm to be enacted by human, potentially with support of a computer tool. Examples include priority ranking using for example a binary search tree algorithm [5], priority grouping [22], cumulative voting [26], and pair-wise comparison on a ratio scale as input to a modification of the AHP algorithm [20]. The output of these methods include a numerical assignment that maps the set of requirements to be prioritized to a scale e.g. ordinal or ratio [22]. Prioritization methods have also been empirically investigated exemplified by Lehtola et al [27] finding several problems of practice, including that stakeholders sometimes find it difficult to estimate how much more valuable one requirement is than another on a ratio scale. Practitioners also sometimes find it difficult to estimate which number in a priority grouping to give to factors.

In this paper we propose to enable stakeholders to mix different ways doing prioritization in a flexible manner enabled by a constraint-based specification of priorities, as demonstrated subsequently. This approach represents a different way of looking at prioritization, that may help address some of the problems identified in empirical research and moves focus from specific algorithms to problem formulation.

C. Release Planning

The term (software) release planning is here used to denote the activity of deciding how to assign releasable product characteristics (features, requirements) to a planned sequence of releases of an evolving software product. Prioritization can be viewed as a sub-problem of release planning, where the latter not only involves assigning priorities according to a set of criteria reflecting the views of a set of stakeholders, but also includes scheduling, resource planning and taking into account requirements inter-dependencies [28]. Release planning is far from trivial and depending on the complexity of the problem it can be very challenging to find a good balance among the factors that impact on the goodness of a release plan, determined e.g. by its business value, risk, investment costs, technical feasibility and last but not least stakeholder satisfaction [8].

Carlshamre [29] discusses release planning in market-driven software product development and propose the use of tool support for exploring different release plans in terms of their value to different stakeholders. The systematic review by Svahnberg et al [30] on strategic release planning models identifies two research groups with major contributions on tool support for release planning: (1) Ruhe et al [8] with the EVOLVE-family of models [9], [31] supported by the commercially available ReleasePlanner tool [32]; and (2) van den Akker et al [33], [6], [7] with a research prototype for release planning implemented based on the ILOG CPLEX Optimizer [34] with a spreadsheet program as a user interface. Both these tools use Integer Linear Programming (also proposed by Bagnall [35]) as underlying technology for finding (near-) optimal solutions to the release planning problem modeled as a set of linear inequalities over integer-valued variables.

In this paper we propose an alternative to the linear programming approach through constraint satisfaction, with the potential of a more powerful problem modeling.

III. Constraint-driven Prioritization

In order to specify requirements prioritization as a constraint satisfaction problem, we here define prioritization as the problem of creating a mapping from a set of n requirements entities, $\{r_i | 1 \leq i \leq n\}$ to a set of numerical values $\{p_i | 1 \leq i \leq n\}$ that reflects the requirements priorities according to a specific criterion and some stakeholders' views. The entities can be e.g. features, user stories, or quality requirements. Criteria can be e.g. market value, development cost, urgency, risk etc. The range of each numerical value p_i belongs to $[1 \ldots m]$. Thus, the interpretation of such a mapping is that p_i represents the priority of requirements r_i with respect to a certain criteria and a certain (set of) stakeholder(s) on m different integer values. In general, requirements prioritization could of course include mappings to any numerical range, but in order to simplify the constraint solving we here restrict the problem to finite sets of integers. Some particular prioritization methods, such as AHP [4] have a range of real valued numbers in $[0 \ldots 1]$, but this can e.g. be approximated by the range of integers $[0 \ldots 100]$ in percent, to keep the range to a finite set.

We demonstrate prioritisation as a CSP for three example approaches from literature: priority ranking, priority grouping and cumulative voting, [22], [4], [26].

In *priority ranking* the range of the priority mapping is the same as the number of requirements, $m = n$. Thus the set of requirements are prioritized on an ordinal scale [4]. Specification 1 provides an example of a CSP priority ranking model in MiniZinc.

In Specification 1, the first constraint is general and provides the property of a true ranking saying that all ranks are different, while the subsequent constraints represent the specific relative priority judgments of a hypothetical stakeholder. Stakeholders can express the priorities using different types of relative constraints including greater than, less than, greater than or equal, etc. Thus, the stake-

Specification 1 A priority ranking example.

```
int: n = 5;
array[1..n] of var 1..n: P;
constraint
  alldifferent(P);
constraint P[1] > P[2];
constraint P[2] > P[3];
constraint P[3] < P[4];
constraint
  forall ( i in 1..n)
    ( P[5] >= P[i] );
solve satisfy;
output [show(P)];
```

holders can choose order of the pairwise comparisons, as well as how the pairwise comparisons are expressed. Further, we can see that the final constraint in Specification 1 is used to pinpoint that one feature has a higher rank than all other features, which illustrates that CSP modeling offers the expressive power of first order predicates. While implementations of AHP for requirements prioritisation often decides the order of pairwise comparisons together with a stipulated scale on how to compare the requirements that is embodied in the specific algorithm of AHP [21], the approach in Specification 1 offer a more general and flexible way to express priorities that gives the control over the way the prioritisation is carried out to the stakeholder in terms of both which steps are taken in which order and the actual formulation of the relative priorities.

There are 3 solutions to Specification 1, as provided by the JaCoP solver output:

```
P = array1d(1..5, [4, 3, 1, 2, 5])
P = array1d(1..5, [4, 2, 1, 3, 5])
P = array1d(1..5, [3, 2, 1, 4, 5])
```

This set of three solutions gives the priorities of $r_5 = 5$ and $r_3 = 1$ in all solutions. However, for r_1, r_2 and r_4 the different solutions give different priorities, providing different options to the stakeholder that are structurally different while still satisfying the constraints. As demonstrated in this example, the stakeholder does not need to be exhaustive in providing ratio scale priorities for each pair-wise comparison of requirements, but can provide those constraints that are known. If the problem is under-determined due to incomplete information then the solver can give a set of all solutions to be assessed. If the solution set is small then the stakeholders can select the preferred option, or if the solution set is very large then the stakeholders may consider adding additional constraints to reduce the solution space. If the problem includes inconsistent constraints then no solution is provided and the set of conflicting constraints can be highlighted by the solver and revisited by the stakeholder.

In general, the problem of identifying minimal subsets of inconsistent constraints is difficult, but a solver can report the constraint that caused inconsistency. Dependencies between these constraints and other constraints can give hints to stakeholders of where to look to resolve an inconsistency.

In *priority grouping* the prioritization involves a mapping to m groups, representing different priority levels, often with an interpretation such as "very high", "high", "medium", "low", "very low"; this typical example has $m = 5$ groups, modeled using MiniZinc as shown in Specification 2.

Specification 2 A priority grouping example.

```
int: n = 7;
array[1..n] of var 1..5: P;
constraint
  forall ( i in {1,2,3})
    ( P[i] = 5 );
constraint P[4] > 3 /\ P[5] < P[4];
constraint P[6] =  P[4] + 1;
constraint P[7] >= P[5] + 3;
solve satisfy;
output [show(P)];
```

In Specification 2 the first constraint is used by the hypothetical stakeholder to express that three specific requirements have top priority, illustrating the ability to set absolute priorities of groups of requirements. Subsequent constraints apply different types of relative priority decisions. The slash-backslash syntax in the second constraints illustrates the use of logical conjunction. This example shows the flexibility provided by CSP specification to enable a combination of both absolute and relative priority judgments.

Prioritisation with *cumulative voting* [26], [22] (also known as the 100$-test) can be modeled similar to Specification 2, modified with that P is specified to be in the range of $0 \ldots 100$, together with the following constraint to ensure the cumulative property:

```
constraint sum (i in 1..n) (P[i]) = 100;
```

Cumulative voting is similar to AHP in that the priorities are given on a scale that can be interpreted as if there is a limited pool of 100 priority points or percentages available. In addition, the AHP algorithm gives a consistency index based on the redundant information given if each requirements is involved in more than one pairwise comparison, while the cumulative voting in general lacks this consistency checking ability. The constraint solving approach to requirements prioritization proposed here provides inherent consistency support as the solver detects if the circular relationships of an over-determined problem specification includes inconsistent constraints.

The constraint-based prioritization models above can be generalized to include several criteria and several stakeholders in a similar way as is illustrated with the release planning example in Specification 3, further explained in the subsequent Section IV.

In summary, the above examples demonstrate that requirements prioritization can be modeled as a constraint satisfaction problem, using a CSP language such as MiniZinc. We argue that the specifications concise and flexible based on the facts that the models are expressed at a high-level and can combine different types of relative and

absolute priority judgments at the discretion of the human stakeholders. Constraint-based requirements prioritization is different compared to previous approaches in that the actual algorithm for finding the total set of ranks, groups or cumulative priority points is left to the machine, while humans can concentrate on deciding relative and absolute priorities in a consistent way.

IV. Release Planning as a CSP

We demonstrate in this section how the release planning problem can be modeled as a constraint satisfaction problem using the same example as Ruhe et al [8]. (This is a small example used for illustration purposes, a larger example is given in [31].) The example project includes 15 features to be planned for either one of the next 2 releases or postponed. There are 2 stakeholders that have different views on each feature's value and urgency. The value of a feature is modeled as an integer in the range $(1..9)$, with 9 representing the highest value. Urgency is modeled as a tuple with an integer for each release (plus the postponed case) in the range $(0..9)$, where 9 represents highest urgency and 0 represents no urgency. The example also includes 4 limited resources including a certain number of person hours of analysts, developers, testers and a monetary budget. For each feature and resource there is an estimate of how much of that resource that is consumed if that feature is selected. There is also a maximum available total capacity for each release. Resource consumption and capacity is represented by non-negative integers. We refer to Ruhe et al [8] for a more elaborate definition of the example. In Specification 3 this release planning example is re-modeled as a CSP using the MiniZinc language.

Input variables. The input variables of the CSP model follow the notation in Ruhe et al [8], where $N = 15$, $K = 2$, $R = 4$, and $S = 2$ represent number of features, releases, resources and stakeholders respectively. The array input variables are: r representing the resource consumption for each feature and resource; $value$ representing feature value for for each stakeholder and feature; $urgency$ representing the urgency for each stakeholder, feature and release; C and P representing coupling and precedence relations among features respectively; $lambda$ representing the relative importance of each stakeholder; ksi representing the relative importance of each release; and finally Cap representing the available resources for each release and resource.

Output variables. The output variables of the CSP model in Specification 3 are: the WAS function [8] representing the weighted average satisfaction of all stakeholders for all features weighted with $lambda$, $value$ and $urgency$ for each release; the K array representing a release plan solution in terms of which feature is chosen to be implemented in which release number or zero if not chosen at all; and F representing the objective function to be maximized, subsequently defined as a constraint over WAS.

Constraints. There are three constraints of this CSP model: The first constraint expresses coupling and prece-

Specification 3 The release planning example by Ruhe et al [8] modeled as a constraint satisfaction problem.

```
int: N;  % number of features
int: K;  % number of releases
int: R;  % number of resources
int: S;  % number of stakeholders
array[1..N] of string: feature_id;
array[1..N, 1..R] of int: r;
array[1..S, 1..N] of int: value;
array[1..S, 1..N, 1..3] of int: urgency;
array[1..3, 1..2] of int: C; % coupling
array[1..5, 1..2] of int: P; % precedence
array[1..S] of int: lambda; % stakeholder importance
array[1..K] of int: ksi; % release importance
% ksi is multiplied by 10 to have integers
array[1..K, 1..R] of int: Cap; %capacity
array[1..N, 1..K] of var int:
  WAS = array2d(1..N, 1..K,
    [ ksi[k] * sum (j in 1..S)
      (lambda[j] * value[j,i] * urgency[j,i,k])
        | i in 1..N, k in 1..K]);

% Variables  ==========
array[1..N] of var 1..K+1: x;   % feature release number
var int: F;  % objective function

% Constraints =========
constraint % dependency constraints
  forall ( i in index_set_1of2(C) )
    ( x[C[i,1]] = x[C[i,2]] )
  /\
  forall ( i in index_set_1of2(P))
    ( x[P[i,1]] <= x[P[i,2]] );
constraint % resource constraints
  forall ( k in 1..K, j in 1..R)
    ( sum (i in 1..N)
    ( r[i,j] * bool2int( x[i] = k) ) <= Cap[k, j] );
constraint % objective function
  F = sum (k in 1..K, i in 1..N )
    ( WAS[i, k] * bool2int(x[i] = k ) );

solve maximize F;
output[ "x = "++show(x)++"\n"++"F = "++show(F)];
```

The listing below is an excerpt of the code to specify the input data that is specific to this particular instance as given by Table 1 in Ruhe et al [8]. Only initial numbers for the input array variables `r`, `value` and `urgency` are shown here.

```
N = 15; K = 2; R = 4; S = 2;
lambda = [ 4, 6 ]; ksi = [ 7, 3 ];
% ksi is multiplied by 10 to have integer values
C = [|7, 8, |9, 12, |13, 14, |];
P = [|2, 1, |5, 6, |3, 11, |8, 9, |13, 15, |];
Cap = [| 1300, 1450, 158, 2200,
       | 1046, 1300,  65, 1750, |];
r = [| 150, 120, 20, 1000, |  75,  10,  8,  200,
       % ... etcetera
value = [| 6, 7, 9, 5,
       % ... etcetera
urgency = array3d(1..S, 1..N, 1..3, [
       % stakeholder 1
       5, 4, 0,
       5, 0, 4,
       9, 0, 0,
       % ... etcetera
```

dence with a predicate saying: (1) for all feature that are part of the set of couplings (in this example there are 3 coupled feature pairs): the features should be in the same release, AND (2) for all features: if that feature is second in a precedence pair relation (in this example there are 5 precedence relations) then it should be implemented in a later release than its preceding feature or not chosen at all. The `/\` and `\/` syntax elements mean logical conjunction (AND) and logical disjunction (OR) respectively.

The second constraint expresses the resource constraints with a predicate saying that for all releases and resources the sum of the resource consumption of all chosen features should be less than or equal to the available capacity for that release and resource. The `bool2int` function converts a Boolean expression to zero if it is false and to one if it is true. It is used her to only sum up the resource values if the feature is chosen, i.e. its release number is non-zero.

The third constraint includes the specification of the objective function F defined as the sum over all releases and all features of the weighted average stakeholder satisfaction for the chosen features of a particular solution.

The statement `solve maximize F` instructs the solver to search for an optimal solution with respect to F. The output after compiling and executing this code includes the x array with the feature allocations to releases:

```
x = [3, 1, 1, 1, 3, 3, 1, 1, 2, 3, 3, 2, 1, 1, 3]
F = 20222
```

This means that the optimal solution allocates features 2, 3, 4, 7, 8, 13 and 14 to release 1. Features 9 and 12 are allocated to release 2, while features 1, 5, 6, 10, 11, and 15 are postponed.

The maximum value of the objective function is output as `F = 20222` (with the `ksi` array a factor 10 higher to give integer values). [1]

We can use the solver to find alternative solutions, e.g. by adding the constraint that the objective function should be more than 95% of the maximum objective function and changing the solver instruction to a satisfaction instead of an optimization:

```
constraint F > 19210;
solve satisfy;
```

If we run solver with the `all-solutions` option, we then get an output of, in this case, 12 different near-optimal solutions that our hypothetical product manager can scrutinize. Some of the solutions are structurally different from others in that different sets of features are shifted among releases [8].

The MiniZinc CSP model in Specification 3 follows the original definitions by Ruhe et al [8], but the optimization problem can actually be specified more computationally efficiently with a so called *global constraint* that makes the solving process more efficient. In the case of the release planning problem example above we can model it as a bin packing problem and use the `bin_packing_capa` global constraint [36] as follows.

```
forall (j in 1..R)(
  let {
    array[1..K+1] of int: capacity =
         array1d(1..K+1,
         [Cap[k, j] | k in 1..K] ++ [10000])
  } in
  bin_packing_capa(capacity,x,[r[n,j]|n in 1..N])
);
```

In the specification above, each resource is modeled as one global bin packing constraint and each release is modeled as a bin that can be filled with features. Bins 1 and 2 are designated for releases 1 and 2, and have respective capacities as defined in the original formulation. Bin 3 is used to pack all features that are not allocated neither to the first nor the second release and is given a high capacity of `10000` to make it fit any postponed feature.

The formulation of release planning as a binpacking problem is not novel [37], but with a CSP problem formulation in a high-level language such as MiniZinc we can reuse the standard algorithms implemented in solvers such as JaCoP. We are thereby taking a step away from algorithm implementation details that previous work is focused on [37], [35], so that we can keep a high-level view of the problem modeling. However, a human problem formulator such as a product manager needs to be aware of and realize that binpacking is a suitable global constraint for certain types of decision problems, and may need support by tailored tools that can point to such opportunities in particular cases when the computational solution search needs faster computation.

V. Discussion: Benefits and Limitations

Benefits. We propose in this paper to model the SPM decision-making problems of prioritization and release planning as constraint satisfaction problems, and we argue that this approach can complement existing approaches, providing several potential benefits as discussed below.

1) *General, powerful, flexible and concise decision problem specification.* The specification of SPM decision-making problems such as prioritization and release planning as a constraint satisfaction problem offers the generality and expressive power of predicate logic constraints when using high-level CSP languages such as MiniZinc [14] that includes many built-in operators, predicates and functions such as comparisons (e.g. `<`, `==`), arithmetic operations (e.g. `+`, `*`, `sum`, `min`), logical operations (e.g. `and`, `xor`, `forall`), set operations (e.g. `union`, `subset`, `in`, `card`), array operations (e.g. `length`, `index_set`), coercions (e.g. `round`, `int2float`, `bool2int`), and bounds operations (`ub`, `lb`, `dom`). In Section III we illustrate how different ways of expressing priority relations (absolute, relative, etc.) can be combined in a flexible way. The problem specifications can be very concise; a previously published non-trivial release planning problem can be specified using a

[1] Our solver found an inconsistency in the original solution in Ruhe et al [8]. Solution $x2$ where $F = 17080$ uses 75 units of resource 3 and is thus violating the capacity limit of 65 for release 2. This error may explain discrepancies between our solution and the previously published.

small set of constraints using less than 40 lines of MiniZinc code.

2) *Problem focus, with no specific algorithm stipulated.* As pointed out in Section II, previous work on requirements prioritization and release planning suggests that product managers engage in the enactment of a specific algorithm [5] or focus researchers attention on algorithm implementation details [37]. With a constraint-based approach to SPM, the machine takes care of the solution finding algorithms, while humans can focus on the problem formulation including the understanding of the constraints and the specification of relations among requirements, stakeholders and resources in the application domain.
3) *Computational scalability.* For small prioritization and release planning problems a CSP solver typically provides solutions within milliseconds. For large and complex problems the algorithmic complexity may yield longer execution times that even may be unreasonably long, but available solvers such as JaCoP embodies much of the existing knowledge on heuristic for efficient search for solutions, e.g. through the use of global constraints as discussed in Section IV. Benchmarking with existing optimization solutions on larger problems including the complexity introduced with task scheduling [31] would be interesting for further work.
4) *Inspiration from priority options and alternative release plans.* Constraint satisfaction provides the opportunity of investigating a problem with many solutions, as solvers with a simple option can provide all solutions that satisfy a set of general constraints. A product manager can iteratively adjust the high-level constraints and explore the consequences, simply by changing the high-level constraint specification. Thus, the SPM decision-making can be informed and inspired by computer-generated solution options given as direct feedback on formal decision problem specifications.
5) *Incomplete SPM decision problem specifications.* In SPM practice, decisions are often based on incomplete and inaccurate information. As the constraint satisfaction solvers can provide many solutions, a product manager can start with an incomplete specification that only includes a small set of constraints, and then generate a family of solutions that may be acceptable under a given uncertainty, and if needed the product manager can iteratively provide more details by adding more constraints or adjusting the constraints so that a harder limit on the solution space is specified.
6) *Detecting inconsistent priorities or impossible release plans.* In practice, relations among requirements such as relative priority relations may include circular references that are gives an inconsistent specification. A CSP solver can, often within short execution time, detect if there is no solution to a constraint-based priority specification or a release planning problem. The product manager can revisit the inconsistent specification and relax or otherwise update the constraints given.
7) *Open constraint solver API.* Our examples in Sections III and IV we used the MiniZinc language [14] to illustrate the power of a high-level CSP language in SPM problem specification. It is, however, also possible to use the API of a constraint solver, such as the JaCoP API [12], directly in Java to provide database connectivity, Internet access, graphical user interfaces, etc.

Some of the above benefits, in particular benefit 3 and 4, also holds for Integer Linear Programming (ILP) that has been applied in previous approaches to release planning, e.g. [8], [33], but we believe that these benefits are even further emphasized if combined with the other benefits of a CSP approach, e.g. a powerful problem specification language and iterative exploration of the solution space. CSP is a complementary method to ILP, and these approaches can be combined to utilize the best abilities of each solving strategy in a specific situation. However, the rich set of CSP constraints, exemplified in benefit 1 above, goes beyond linear inequalities and also include arithmetical and specialized global constraints not only for integers but also for sets. This can potentially make problem specification easier for product managers, although this conjecture needs to be verified empirically. CSP specification also provides more flexibility in the choice of solving strategy, compared to ILP, as CSP can address both optimization and satisfaction problems.

Limitations. There are several potential limitations of the constraint solving approach to SPM that are outlined subsequently:

1) *Constraint solving competence needed.* Product managers need a degree of understanding of constraint solving, and it can be questioned if the average product manager is capable of using a constraint specification language. Some product managers have engineering background and may very well be able to utilize the expressive power of languages such as MiniZinc. Other product managers may get access to the power of constraint solving techniques if provided with a GUI front end that can hide some of the details.
2) *Scalability in output analysis by humans.* The strength of enabling incomplete specifications can also be a problem if the solution space turns out to be very large. As the number of features of a release plan increases, a product manager may be overloaded with solution options, resulting in a scalability threat to the proposed approach. However, it may be possible to develop heuristics for assisting product managers in the exploration of the solution space.
3) *No SPM-specific constraints.* As SPM currently is no major application area of constraint solving, the algorithms implemented are perhaps better tailored to

other areas than SPM. There may be unimplemented constraint types or search heuristics that is needed to better support SPM, that may be discovered if constraint solving is applied in empirical studies and industrial SPM practice.

4) *No GUI tailored for SPM.* Currently, to the best of our knowledge, no user interface front-end to constraint solvers exists that is tailored specifically to SPM with user-friendly packaging of SPM-relevant decision problems.
5) *Further development in constraints solving technology may be needed.* The industrial practice of SPM often involves great uncertainty in estimates of e.g. business value and implementation effort and information on inter-dependencies may be lacking. A problem that includes inconsistent constraints may need "softer" satisfaction search to satisfy a relevant subset of soft constraints [38]. Thus, constraint solving technology may not be mature enough for SPM to handle a reality of ill-structured problems, frequent inconsistencies and confidence intervals with a stochastic nature.

Based on the above limitations and the demonstration of the idea to apply constraint solving to SPM decision-making problems in previous sections, we propose in Section VI a set of issues of further research that may help to realize potential benefits and address some of the potential limitations.

VI. Conclusion and Further Research

In summary, this paper makes the following claims of contribution, novelty and utility:

- *Contribution.* We demonstrate how constraint satisfaction can be applied to software product management decision problems, such as requirements prioritization and release planning. This approach is discussed in terms of benefits and limitations leading to a set of issues for further research.
- *Novelty.* Previous approaches have been devoted to specific algorithms for solving the decision problems of prioritization and release planning. The approach of constraint satisfaction has to the best of our knowledge not been proposed previously, being the first approach that may help humans to focus on the problem specification while enabling the machine to combine standard implementation of many different optimization and satisfaction algorithms.
- *Utility.* Our conjecture is that constraint satisfaction problem specification for SPM can provide a more general, powerful, flexible and concise way of expressing prioritization and release planning problems with problem understanding rather than algorithmic implementation in focus. We demonstrate using examples how these benefits may be realized and discuss potential limitations. The discussion on benefits and limitations can be a basis both for future empirical work in SPM to verify the potential benefits, as well as for research into new solver capabilities, user interfaces and methodological support to address limitations.

Research agenda. Based on the provided demonstration and discussion of a constraint satisfaction approach to SPM decision-making, we conclude by giving a list of issues that can act as a part of a research agenda for the communities of SPM and CSP researchers to take on in future research:

1) *Empirical studies.* There is a need to verify the proposed approach by empirical investigation of domain-specific constraints in prioritization and release planning. In particular, it is important to further understand what types of constraints that are most important in practice, and how practicing product managers and other SPM stakeholders think when they decide on priorities and release plans. A major conjecture to be investigated is the utility of high-level CSP specification languages such as MiniZinc in use by practitioners as a tool for understanding and investigation of SPM decision problems.
2) *Scalability.* It is important to investigate the scalability of the proposed approach both in terms of computational efficiency and human usability aspects. It would be interesting to compare runtime of existing approaches to CSP solvers. In general, the bin packing and knap sack problems are NP-complete, thus ILP in combination with other optimization approaches such as genetic algorithms may yield faster execution in particular cases. Benchmarking the computational efficiency for finding optimal and near optimal solutions would thus be interesting. Perhaps even more important is to investigate how humans can handle large-scale solutions spaces, and how support can be given to humans in their navigation among many different potential decisions. This is related to the issue of visualization.
3) *Visualization.* In order to help product managers in their decision-making it would be valuable to investigate graphical user interfaces for solution exploration including new ways of visualization of priorities and their inter-relationships as well as depicting feature allocations to releases and resources. By defining measures of distance between solutions and providing support for the clustering of similar solution, useful visualizations may be discovered. This is related to solution robustness analysis.
4) *Solution robustness analysis.* In release planning it is very interesting to do what-if-analysis [33] to investigate the effects of changed constraints on release plans. Product managers may prefer solutions that are not so sensitive to changes in feature allocations. Robustness of release plans expressed using constraints may be investigated to see if and how the CSP approach can support robustness analysis beyond existing approaches.
5) *Domain-specific languages for CSP in SPM.* General constraint satisfaction languages such as MiniZinc lack some capabilities that would make specifica-

tions easier for product managers to interpret. In parallel with further development of general CSP languages, researchers can also investigate special purpose languages for specification of SPM decision problems. The balance between generality and specificity is an interesting topic for further research.

6) *CSP solver API-development for SPM-tooling.* By adapting solver Application Programming Interfaces such as the JaCoP API [12], more tailored support for SPM tooling may be provided. Such work can be a basis for combing existing requirements database applications and feature tracking tools with the constraint solving approach.
7) *Heuristics for global constraints.* CSP solves provide a comprehensive toolbox for expressing constraint problems using efficient global constraints as discussed in Section IV. However, modeling of SPM decision-making problems with constraints require specific competence. Thus, it would be interesting to develop guidelines and heuristics that map SPM decision problems to constraint specifications, and investigate if this can help practitioners in utilizing the potential benefits of CSP applications in SPM.
8) *Analysis of inconsistent constraints.* In general, the problem of identifying minimal subsets of inconsistent constraints can be computationally hard for large problems. However, better support for tracing the solution search process beyond a line number in a specification would be interesting, in particular pin-pointing constraints involved in an circular inconsistencies through the common variables involved in a set of relations.
9) *Soft and stochastic constraints for SPM.* The decision-making in real-world SPM practice involves great uncertainties and incomplete information. Soft constraint approaches [38] and stochastic constraint programming [39] can be investigated to see if these extensions to CSP can provide added benefits to the modeling of SPM problems.

The above potential items of a SPM-CSP research agenda are given in non-prioritized order. However, we believe that further empirical understanding of SPM decision-making is of highest priority when selecting research directions. Otherwise we risk focus on small details with little relevance to future SPM practice.

ACKNOWLEDGMENT

The authors would like to give special thanks to Dietmar Pfahl for valuable comments on a draft version of this paper. Many thanks also to the anonymous reviewers that provided many constructive improvement proposals. The project is partly funded by the Swedish Foundation for Strategic Research and VINNOVA (The Swedish Governmental Agency for Innovation Systems) within the EASE Industrial Excellence Center: http://ease.cs.lth.se/

REFERENCES

[1] S. Brinkkemper, C. Ebert, and J. Versendaal, "Proceedings of the first international workshop on software product management," in *Software Product Management, 2006. IWSPM '06. International Workshop on*, sept. 2006, pp. 1 –2.

[2] I. van de Weerd, S. Brinkkemper, R. Nieuwenhuis, J. Versendaal, and L. Bijlsma, "On the creation of a reference framework for software product management: Validation and tool support," *2006 International Workshop on Software Product Management (IWSPM'06)*, pp. 3–12, 2006.

[3] C. Ebert, "The impacts of software product management," *Journal of Systems and Software*, vol. 80, no. 6, pp. 850 – 861, 2007.

[4] J. Karlsson, C. Wohlin, and B. Regnell, "An evaluation of methods for prioritizing software requirements," *Information and Software Technology*, vol. 39, no. 14-15, pp. 939–947, 1998.

[5] T. Bebensee, I. van de Weerd, and S. Brinkkemper, "Binary priority list for prioritizing software requirements," in *Requirements Engineering: Foundation for Software Quality*, ser. Lecture Notes in Computer Science, R. Wieringa and A. Persson, Eds. Springer Berlin / Heidelberg, 2010, vol. 6182, pp. 67–78.

[6] C. Li, J. van den Akker, S. Brinkkemper, and G. Diepen, "Integrated requirement selection and scheduling for the release planning of a software product," in *Requirements Engineering: Foundation for Software Quality*, ser. Lecture Notes in Computer Science, P. Sawyer, B. Paech, and P. Heymans, Eds. Springer Berlin / Heidelberg, 2007, vol. 4542, pp. 93–108.

[7] C. Li, M. van den Akker, S. Brinkkemper, and G. Diepen, "An integrated approach for requirement selection and scheduling in software release planning," *Requirements Engineering*, vol. 15, pp. 375–396, 2010.

[8] G. Ruhe and M. Saliu, "The art and science of software release planning," *Software, IEEE*, vol. 22, no. 6, pp. 47 – 53, nov.-dec. 2005.

[9] D. Greer and G. Ruhe, "Software release planning: an evolutionary and iterative approach," *Information and Software Technology*, vol. 46, no. 4, pp. 243 – 253, 2004.

[10] E. Tsang, *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1993. [Online]. Available: http://www.bracil.net/edward/FCS.html

[11] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, no. 3, pp. 355–383, Jul. 2003.

[12] JaCoP web page, visited May 2010. [Online]. Available: http://www.jacop.eu/

[13] F. Rossi, P. van Beek, and T. Walsh, Eds., *Handbook of Constraint Programming*. Amsterdam, The Netherlands: Elsevier Science Publishers, 2006.

[14] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack, "Minizinc: Towards a standard cp modelling language," in *Principles and Practice of Constraint Programming (CP2007)*, ser. Lecture Notes in Computer Science, C. Bessiere, Ed. Springer Berlin / Heidelberg, 2007, vol. 4741, pp. 529–543.

[15] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: a literature review," *Information Systems*, vol. 35, no. 6, 2010. [Online]. Available: http://dx.doi.org/10.1016/j.is.2010.01.001

[16] S. Woods and Q. Yang, "Program understanding as constraint satisfaction: representation and reasoning techniques," *Automated Software Engineering*, vol. 5, no. 2, pp. 147 – 81, 1998/04/.

[17] R. DeMilli and A. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900 – 10, Sept. 1991.

[18] C. Salinesi, R. Mazo, D. Diaz, and O. Djebbi, "Using integer constraint solving in reuse based requirements engineering," *18th IEEE International Requirements Engineering Conference (RE'10)*, pp. 243–251, 2010.

[19] C. Salinesi, D. Diaz, O. Djebbi, R. Mazo, and C. Rolland, "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.

[20] J. Karlsson, "Software requirements prioritizing," *Requirements Engineering, 1996., Proceedings of the Second International Conference on*, pp. 110–116, 1996.

[21] J. Karlsson, S. Olsson, and K. Ryan, "Improved practical support for large-scale requirements prioritising," *Requirements Engineering*, vol. 2, pp. 51–60, 1997.

[22] P. Berander and A. Andrews, "Requirements prioritization," in *Engineering and Managing Software Requirements*, A. Aurum and C. Wohlin, Eds. Springer Berlin Heidelberg, 2005, pp. 69–94.

[23] B. Regnell and S. Brinkkemper, "Market-driven requirements engineering for software products," in *Engineering and Managing Software Requirements*, A. Aurum and C. Wohlin, Eds. Springer Berlin Heidelberg, 2005, pp. 287–308.

[24] I. van de Weerd, S. Brinkkemper, R. Nieuwenhuis, J. Versendaal, and L. Bijlsma, "Towards a reference framework for software product management," *14th IEEE International Requirements Engineering Conference (RE'06)*, pp. 319–322, 2006.

[25] A. Herrmann and M. Daneva, "Requirements prioritization based on benefit and cost prediction: An agenda for future research," in *International Requirements Engineering, 2008. RE '08. 16th IEEE*, sept. 2008, pp. 125 –134.

[26] B. Regnell, M. Host, J. Natt och Dag, P. Beremark, and T. Hjelm, "An industrial case study on distributed prioritisation in market-driven requirements engineering for packaged software," *Requirements Engineering*, vol. 6, no. 1, pp. 51–62, 2001.

[27] L. Lehtola and M. Kauppinen, "Empirical evaluation of two requirements prioritization methods in product development projects," in *Software Process Improvement*, ser. Lecture Notes in Computer Science, T. Dingsøyr, Ed. Springer Berlin / Heidelberg, 2004, vol. 3281, pp. 161–170.

[28] P. Carlshamre, K. Sandahl, M. Lindvall, B. Regnell, and J. Natt och Dag, "An industrial survey of requirements interdependencies in software product release planning," 2001, pp. 84 – 91.

[29] P. Carlshamre, "Release planning in market-driven software product development: Provoking an understanding," *Requirements Engineering*, vol. 7, no. 3, pp. 139–151, 2002.

[30] M. Svahnberg, T. Gorschek, R. Feldt, R. Torkar, S. B. Saleem, and M. U. Shafique, "A systematic review on strategic release planning models," *Information and Software Technology*, vol. 52, no. 3, pp. 237 – 248, 2010.

[31] A. Ngo-The and G. Ruhe, "Optimized resource allocation for software release planning," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 109 – 23, 2009.

[32] Release Planner web page, visited May 2010. [Online]. Available: https://www.releaseplanner.com/product.htm

[33] M. van den Akker, S. Brinkkemper, G. Diepen, and J. Versendaal, "Software product release planning through optimization and what-if analysis," *Information and Software Technology*, vol. 50, no. 1-2, pp. 101 – 111, 2008.

[34] IBM ILOG CPLEX Optimizer web page, visited May 2010. [Online]. Available: http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/

[35] A. Bagnall, V. Rayward-Smith, and I. Whittley, "The next release problem," *Information and Software Technology*, vol. 43, no. 14, pp. 883 – 90, 2001/12/15.

[36] P. Shaw, "A constraint for bin packing," in *Principles and Practice of Constraint Programming (CP 2004)*, ser. Lecture Notes in Computer Science, M. Wallace, Ed. Springer Berlin / Heidelberg, 2004, vol. 3258, pp. 648–662.

[37] A. Szőke, "Bin-packing-based planning of agile releases," in *Evaluation of Novel Approaches to Software Engineering*, ser. Communications in Computer and Information Science, L. A. Maciaszek, C. Gonzalez-Perez, and S. Jablonski, Eds. Springer Berlin Heidelberg, 2010, vol. 69, pp. 133–146.

[38] P. Meseguer, F. Rossi, and T. Schiex, "Chapter 9 soft constraints," in *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence, P. v. B. Francesca Rossi and T. Walsh, Eds. Elsevier, 2006, vol. 2, pp. 281 – 328.

[39] T. Walsh, "Stochastic constraint programming," in *15th European Conference on Artificial Intelligence*, F. van Harmelen, Ed. IOS Press, Amsterdam, 2002, pp. 111–115.